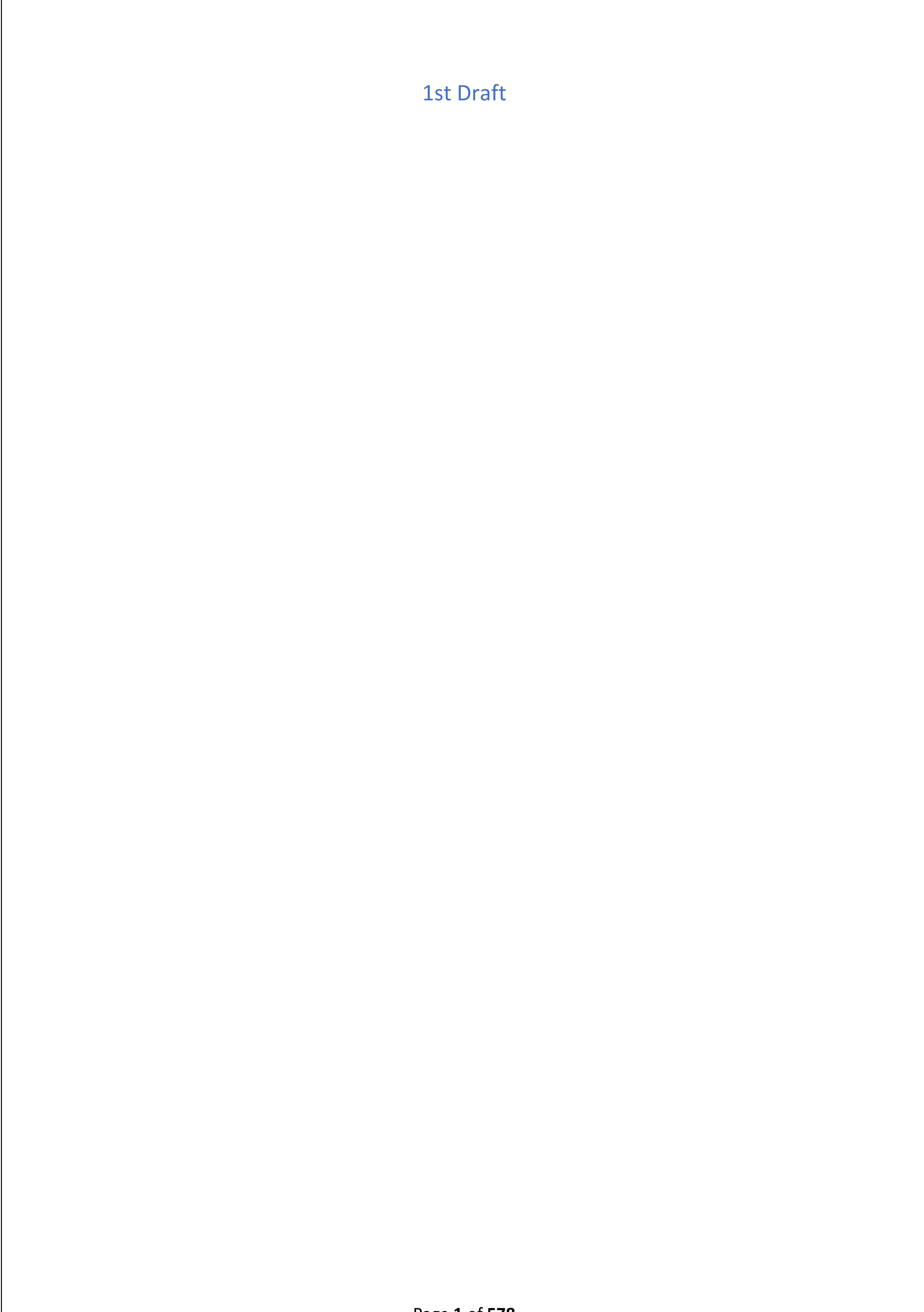
IT Industry Readiness Bootcamp Program

Mobile Application Development

Course Manual



1st Draft

Contents

Week 1: Introduction to Mobile App Development Basics 11

Day 1: INTRODUCTION 11

Understanding Mobile App Development course 11

What is mobile app development? 11

Importance and relevance of mobile apps in today's world. 11

Role and Importance of Mobile Apps in Today's World 11

App Development Life Cycle 11

1. Discovery Phase & Planning 12

2. Design 12

3. Development 12

4. Testing & Quality Assurance Phase 12

5. Release Phase 13

5. Deployment 13

6. Maintenance 13

What are the 6 Application Development Life Cycle Models? 13

Waterfall Model 13

Agile Model 13

V-Shaped Model 14

Iterative Model 14

Spiral Model 14

DevOps Model 15

Discover the difference between Consumer and Enterprise Mobile Apps 16

7 Key Steps of Mobile App Development Process 16

Discover the Main Differences Between iOS and Android Apps 17

Day 2: Overview of iOS and Android Platforms 17

History and significance of iOS. 17

Overview of iOS development environment. 18

History and significance of Android. 19

What are the security and privacy features of Apple iOS? 19

Apple iOS version history 19

Overview of Android development environment. 21

Day 3: Setting up Integrated Development Environment 22

Using Command Line for Kotlin Development 22

Creating a Kotlin Project from the Command Line 22
Introduction to IDE's 22
Setting up IntelliJ IDEA for Kotlin Development 25
Configuring Android Studio for Kotlin 26
Day 4: Introduction to programming 28
Definition of Programming 28
Importance of Programming 28
Day 5: Basic Concepts in Programming 30
Introduction to Kotlin 30
Variables and Data Types 30
Week-2 – Introduction to Programming Concepts 35
Day 1: Introduction to Programming Concepts 35
Definition of variables and data types. 35
Creating different programs on different data types. 35
Examples in the Kotlin language. 36
Example 2: Check whether a number is even or odd using if...else expression 41
Day 2: Control Structures 43
Conditional Statements (if-else) 43
Handling Multiple Conditions: 45
Ternary Operator (Conditional Expression): 45
Learning loop structures for repetitive tasks. 45
Day 3: Programming Logic and Algorithms 50
Logic Building 50
Developing logical thinking for problem-solving 50
Introduction to Algorithms 50
Understanding basic algorithms and their importance. 50
Day 4: Introduction to the Kotlin Programming Language 51
Kotlin Basics 51
Basic syntax and structure. 51
Variables and Data Types in the Chosen Language 51
Practice using variables and different data types. 51
Day 5: Functions or Methods 51
Purpose of Functions 51
Parameters and Return Values 51
Writing different types of functions 52
Examples: 52

Project Discussion 53
Week-3 Data Structures and Algorithms in Kotlin 54
Introduction to Data Structures 54
Day 1: Introduction to Kotlin Arrays 54
Arrays in Kotlin: Declaration, Initialization 58
Array Operations: Push, Pop 59
Practice Exercise 59
Day 2: Lists and Basic Algorithms 59
Lists in Kotlin: Mutable List vs. List 73
List Operations: Insertion, Deletion, Searching 73
Linked Lists in Kotlin 74
Day 3: Maps (Dictionaries) 75
Maps (HashMap’s) in Kotlin: Declaration, Insertion, Retrieval 75
Day 4: Complete Revision of Kotlin 80
Day 5: Practice Session & Working on Projects 80
Week 4- User Interface design for Mobile Apps 81
Day 1: Introduction to UI/UX Design 81
Principles of UI/UX Design 81
What is the difference between UX and UI? 81
Day 2: Designing for Mobile Apps 122
Day 3: Visual Design Basics 124
Day 4: Layout and Composition 126
Why is the Hierarchy of Information Important? 134
Day 5: Interaction Design 134
Understanding User Flows 134
Creating intuitive navigation paths. 134
What is Micro Animation? 134
The Role of Animation in Enhancing User Experience 139
Week 5- Building User Interface 142
Day 1: Implementing App Layouts and Views 142
Layout Structure 142
Adding Components 143
Supported Components 143
Labels 143
Checkboxes 144
Radio Buttons 144

Text Fields ... 144
Combo Boxes ... 145
Spinners ... 145
Link Label ... 145
Separators ... 145
Explanatory Text ... 145
Integrating Panels with Property Bindings ... 146
Dialogs ... 146
Configurables ... 146
Enabling and Disabling Controls ... 146
Understanding different layout types ... 219
a) Linear Layout ... 219
b) Relative Layout: ... 220
c) Constraint Layout: ... 221
d)Frame Layout ... 222
e) Table Layout: ... 223
How layouts affect the appearance of your app. ... 224
Creating Views: ... 224
Adding UI Elements: ... 224
Buttons: ... 224
Text Fields: ... 224
Labels (TextViews): ... 225
Images (ImageViews): ... 225
Lists (ListView or RecyclerView): ... 225
Checkboxes and Radio Buttons: ... 225
Spinners: ... 225
. ... 226
Day 3: Working with Navigation and Menus ... 226
Navigation Basics: ... 226
Setting up fragment managers (Android). ... 226
Navigating between different screens or views. ... 226
Creating Menus: ... 227
Implementing menus for actions or navigation options. ... 227
Contextual menus for specific views or elements. ... 227
Navigation Basics: ... 256
Setting up Fragment Managers (Android): ... 256

Navigating between different screens or views. 256
Creating Menus: 256
Day 4: Advanced UI Components (Optional) 258
Using Advanced UI Components: 258
Implementing features like image galleries, maps, etc. 258
Advanced UI components to enhance 263
RecyclerView: 263
DatePickerDialog: 263
TimePickerDialog: 264
Implementing features like image galleries, maps, etc. 264
Day 5: Practice and Review 266
Project Activity: 266
Building a small app with various UI components, input handling, and navigation. 266
Week 6 – Working with Data 267
Week 6 – Working with Data 267
Day 1: Introduction to Data Handling 267
Understanding Data in Apps: 267
Different types of data (e.g., text, images, user inputs) 268
Importance of efficient data management. 268
Basic Data Storage: 269
Introduction to key-value pairs, file storage, etc. 269
Day 2: Introduction to Databases 270
Database Basics: 270
Importance of structured data. 271
Types of Databases: 272
Overview of different database options (SQLite, Firebase, etc.). 272
Pros and cons of each. 274
Day 3: Working with SQLite (or chosen Database) 276
Setting Up SQLite: 276
Integrating SQLite into your mobile app project. 277
Day 4: Working with Firebase (or chosen Database) 281
Setting Up Firebase: 281
Day 5 Uploading, retrieving, and managing data with Firebase Firestore 283
Week 7- Handling Features and Sensors 283
Day 1: Introduction to Device Features 283
Overview of Device Features: 283

Understanding the capabilities of mobile devices (camera, GPS, accelerometer, etc.). 284
Camera: 284
GPS (Global Positioning System): 285
Accelerometer: 285
Gyroscope: 285
Magnetometer (Compass): 285
Proximity Sensor: 285
Ambient Light Sensor: 285
Fingerprint Sensor/Biometric Sensors: 285
How these features can enhance app functionality. 286
Day 2: Working with the Camera 286
Camera Integration: 286
Setting up camera access in your app. 286
Capturing photos and videos. 287
Processing Media: 287
Editing, filtering, or applying effects to captured media. 287
Day 3: Implementing GPS and Location Services 288
Location Services Integration: 288
Accessing GPS data in your app. 288
Day 4: Utilizing the Accelerometer and Sensors 290
Understanding Sensors: 290
How to access and utilize sensor data. 290
Implementing Motion-Based Interactions: 291
Day 5: Device Permissions and Security 293
Requesting Permissions: 293
Why You Need the Permission: 293
Handle Denied Permission: 293
Day 1: Introduction to Device Features 294
Overview of Device Features: 294
Understanding the capabilities of mobile devices (camera, GPS, accelerometer, etc.). 294
How these features can enhance app functionality. 294
Choosing Relevant Features: 294
Identifying which device features are applicable to your app's purpose. 294
Day 2: Working with the Camera 294
Camera Integration: 294
Setting up camera access in your app. 294

Capturing photos and videos. 294
Processing Media: 294
Editing, filtering, or applying effects to captured media. 294
Day 3: Implementing GPS and Location Services 294
Location Services Integration: 294
Accessing GPS data in your app. 294
Retrieving current location and tracking movement. 294
Day 4: Utilizing the Accelerometer and Sensors 294
Understanding Sensors: 294
Introduction to different sensors (accelerometer, gyroscope, etc.) 294
How to access and utilize sensor data. 294
Implementing Motion-Based Interactions: 294
Using sensor data for interactive elements in your app. 294
Day 5: Device Permissions and Security 294
Requesting Permissions: 294
How to ask for user consent to access device features. 294
Best practices for handling permissions. 294
Security Considerations: 294
Ensuring secure handling of sensitive data and device features. 294
Week 8- Mobile Application Development Tools and Frameworks 295
Day 1: Introduction to Mobile App Development Frameworks 295
Understanding Mobile App Development Frameworks 295
Overview of popular cross-platform frameworks (e.g., React Native) 295
React Native: 295
Flutter: 296
Pros and cons of using cross-platform frameworks. 296
Choosing the Right Framework: 298
Day 2: Exploring React Native 299
Setting Up the Development Environment: 300
Installing React Native and required dependencies 300
Creating a New React Native Project 300
Setting up a new project. 300
Building a Simple App: 300
Creating a basic app using React Native. 300
Understanding component-based architecture. 300
Day 3: Exploring Flutter 302

Setting Up the Development Environment: 302
Get the Flutter SDK. 302
Setting up a new project. 303
Building a Simple App: 303
Understanding the widget-based architecture. 303
. 304
Day 4: Introduction to App Development Tools 304
Overview of Integrated Development Environments (IDEs): 304
Introduction to VS code for cross-platform development. 304
Understanding the features and functionalities of each IDE. 304
Day 5: Testing and Debugging Mobile Apps 305
Unit Testing: 305
Writing and executing unit tests for your app. 305
Manual Testing: 305
Exploratory testing and user acceptance testing. 305
Debugging Tools and Techniques: 306
Week -9 App Deployment and Distribution 306
Debugging and Profiling 353
Week-10 App Performance Optimization 391
Day 1: Introduction to App Performance Optimization 391
Why Performance Matters: 391
Understanding the impact of performance on user experience and app success. 391
Real-world examples of poorly performing apps. 391
Key Performance Metrics: 391
Identifying important metrics like response time, CPU usage, memory consumption, etc 391
Day 2: Memory Management and Resource Optimization 391
Memory Management Techniques: 391
Understanding memory allocation and deallocation. 391
Implementing best practices for efficient memory usage. 391
Resource Optimization: 391
Techniques for optimizing resource-intensive operations (e.g., image loading, network requests). 391
Day 3: Performance Profiling and Debugging 392
Profiling Tools: 392
Introduction to performance profiling tools (e.g., Instruments for iOS, Android Profiler for Android). 392

Using these tools to identify performance bottlenecks. 392
Debugging Techniques: 392
Troubleshooting common performance issues. 392
Using profiling data to pinpoint areas for improvement. 393
Day 4: UI/UX Impact on Performance 393
Optimizing UI Rendering: 393
Strategies for efficient rendering of UI components. 393
Avoiding unnecessary layout recalculations. 393
Reducing Network Latency: 393
Techniques for minimizing network requests and optimizing data retrieval. 393
Day 5: Caching and Data Management 393
Implementing Caching: 393
Utilizing caching mechanisms to store and retrieve data. 393
Choosing the right caching strategy for your app. 393
Data Management Best Practices: 393
Strategies for handling large datasets efficiently. 393
Week-11 Mobile App Security and Privacy 394
1 Introduction to the Crypto Library 429
Configuring SSL or TLS certificates 442
Procedure 442
Week -12 Project Development 497
What is monetization? 517
Monetization, startups and the product lifecycle 518
What are in-app purchases (IAPs)? 521
Why are IAPs important? 522

Week 1: Introduction to Mobile App Development Basics

Day 1: INTRODUCTION

Understanding Mobile App Development course

Welcome to the "Mobile App Development Basics" course! In this course, we will provide you with a comprehensive understanding of mobile app development. You'll learn how to create applications for various mobile platforms, gaining the skills needed to bring your app ideas to life.

What is mobile app development?

Mobile app development refers to the process of creating software applications that are designed to run on mobile devices such as smartphones and tablets. These applications are tailored to take advantage of the unique capabilities of mobile hardware and provide valuable services to users.

Mobile application development is the process of making software for smartphones, tablets and digital assistants, most commonly for the Android and iOS operating systems. The software can be preinstalled on the device, downloaded from a mobile app store or accessed through a mobile web browser. The programming and markup languages used for this kind of software development include Java, Swift, C# and HTML5.

Mobile app development is rapidly growing. From retail, telecommunications and e-commerce to insurance, healthcare and government, organizations across industries must meet user expectations for real-time, convenient ways to conduct transactions and access information. Today, mobile devices—and the mobile applications that unlock their value—are the most popular way for people and businesses to connect to the internet. To stay relevant, responsive and successful, organizations need to develop the mobile applications that their customers, partners and employees demand.

Yet mobile application development might seem daunting. Once you've selected the OS platform or platforms, you need to overcome the limitations of mobile devices and usher your app all the way past the potential hurdles of distribution. Fortunately, by following a few basic guidelines and best practices, you can streamline your application development journey.

Importance and relevance of mobile apps in today's world.

Mobile apps have revolutionized the way we interact with technology and have become an integral part of our daily lives. From communication and entertainment to productivity and e-commerce, apps have transformed various industries.

Role and Importance of Mobile Apps in Today's World

The Ecommerce has added value to the mode of shopping. These days online shopping is one of the most significant advantages that people are enjoying with the help of mobile apps. Now it is easier to get your favorite clothes, shoes, home accessories and much more materials online without visiting any shop. It is easy to search the item of choice from the comfort of your home and that too at affordable rates. Online shopping is a type of industry where both the selling and buying process is easy.

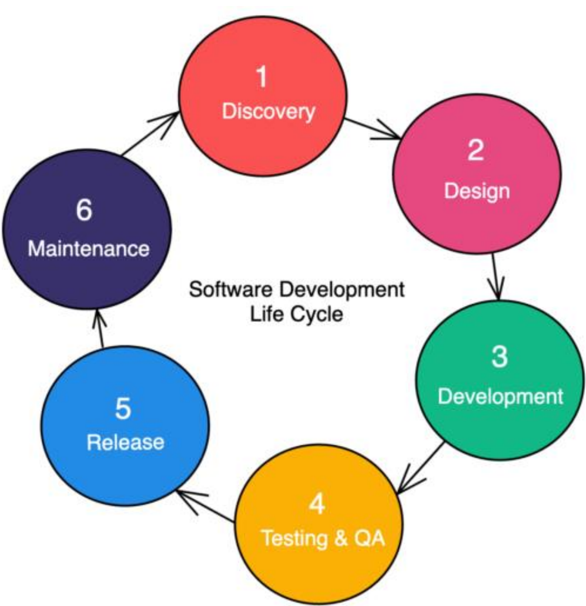
App Development Life Cycle

The discovery phase is the most important phase of the life cycle since it sets the requirements for what to build. During this requirement-gathering phase, you collect feedback from users, find shortcomings in the application, and establish an action plan to address them. Product managers usually do most of the heavy lifting during this portion because this is where they gather important information about the application from stakeholders.

The app development life cycle is a structured process that ensures the creation of high-quality, reliable, and user-friendly applications. It consists of several distinct phases, each crucial for the success of the app:

1. Discovery Phase & Planning

During the planning phase, the project's scope, objectives, and requirements are defined. It involves conducting market research to understand the target audience and competition. Additionally, feature specifications are outlined to guide the development process.



2. Design

The design phase involves drafting solutions to problems found during the discovery phase. UX/UI designers are the most active participants during this phase since they work with the team to create potential solutions from the design point of view. UX/UI designers will use mediums like blueprints, wireframes, and mock-ups to display these solutions and alert the team to potential blockers ahead of time.

3. Development

In the development phase, the actual code of the application is written. Developers choose the development platform (native or cross-platform) and integrate any necessary backend components for data storage and retrieval.

4. Testing & Quality Assurance Phase

Once the development team builds the necessary features, the application is ready for testing. At this point, the team sets up one or more environments to reflect the production version of the app so that they can test its functionality (either manually or by using automated integration testing pipelines). QA professionals and security engineers are involved in this phase, with the singular goal of finding issues that will prevent the app from working as it should. Problems encountered during this phase, like bugs and security and compliance issues, are reported to the development team so they can address them. Because of limited time, the team will likely prioritize issues in order of severity and decide whether to fix some or all of them.

The QA team is usually the last line of defense because they decide whether certain features are ready for users.

Testing is a critical phase where the application is thoroughly examined to identify and fix any issues. This includes unit testing, integration testing, and user acceptance testing (UAT) to ensure the app meets quality standards.

5. Release Phase

Once the QA team vets the application for accuracy, the app is ready for customer use. This step usually involves bumping up the app version to reflect the latest release and sending updates to users that newer versions of the app are available for download.

5. Deployment

During deployment, the app is prepared and submitted to app stores for distribution. This phase involves packaging the app, writing descriptions, and creating necessary promotional materials.

6. Maintenance

After deployment, the app requires ongoing maintenance. This includes bug fixing, performance optimization, and the addition of new features to keep the app relevant and functional.

What are the 6 Application Development Life Cycle Models?

Although most organizations go with the waterfall or agile approach, you can use many different models (or methodologies) to bring your application to life. Let's review some examples of those below.

Waterfall Model

The waterfall model is the most traditional and straightforward approach to building an application. The model has this name because you display the phases like a waterfall. With this model, you must follow each step sequentially and not overlap. The waterfall model is an excellent choice if the requirements are constant and don't change regularly.

Pros:

1. Clear and well-defined steps make it easy to release quality software at the end of the cycle.
2. Easy to understand and follow, the learning curve is negligible.

Cons:

1. Since the model is rigid, you can't incorporate new changes during an ongoing cycle.
2. Testing and QA are only done at the very end of the cycle, which makes it difficult to account for unexpected bugs.

Agile Model

The agile model is one of the most popular and widely-used approaches to building applications. Agile methodology states that you should deliver a product's value in stages. Compared to the waterfall model, where each step is followed one by one without any overlap, the agile model encourages and expects phases to overlap.

Pros:

1. It gives you the flexibility to change and adapt to new requirements throughout the development process.
2. Continuous testing and customer feedback help identify and mitigate issues early in the development cycle.

Cons:

1. It requires a high level of communication and coordination among team members.
2. Continuous changes and iterations to requirements can lead to scope creep, which can result in a project that never reaches completion or is not of good quality.

V-Shaped Model

The V-shaped model considers testing during each phase of application development. It looks very similar to the waterfall method but includes corresponding testing stages as additional visual elements.

Pros:

1. Testing is integrated into each stage of the development process, reducing the likelihood of errors or bugs in the final product.

Cons:

1. It’s very rigid, making it hard to modify requirements once you complete a phase.
2. Testing can be time-consuming and expensive, particularly for larger projects.

Iterative Model

In an iterative model, the development team usually starts with vague requirements to build the initial version of the app and then improves upon it after each release.

Pros:

1. It’s good for applications that may require changes in the future, i.e., big projects.
2. Testing and releasing the application is easier because you can deliver it in iterations.

Cons:

1. It’s not ideal for small projects or projects with a limited budget.
2. The design will frequently change because of incomplete requirements.

Spiral Model

The spiral model incorporates risk when delivering your application to customers, and it involves a series of steps that quantify risk analysis, prototyping, experimentation, and evaluation.

Pros:

1. It helps identify and mitigate risks early in the development phase of the cycle.
2. It incorporates testing and feedback during each phase of the cycle to improve the overall quality of the final product.

Cons:

1. It requires a high level of expertise to perform accurate risk analysis, which may only be available to some development teams.

DevOps Model

Unlike the models mentioned above, the DevOps model is relatively new in the industry. It emphasizes that delivering software is a joint effort between Development (Dev) and IT Operations (Ops) teams, and its ultimate goal is to improve the speed and quality of the final product.

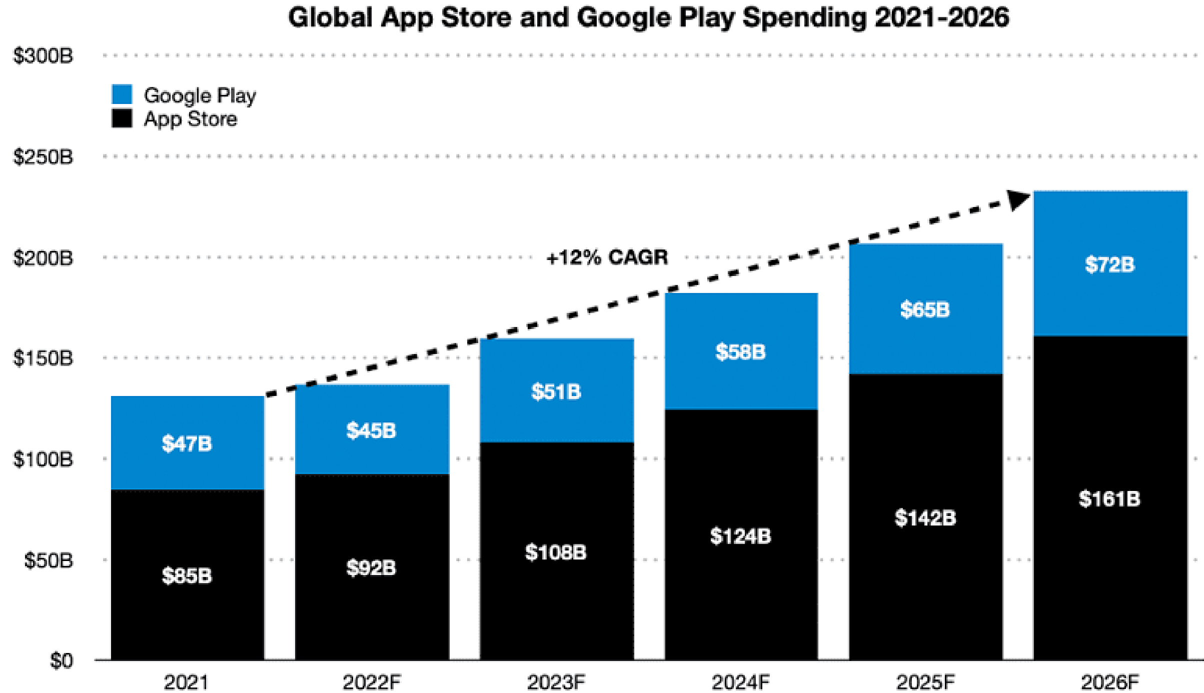
Pros:

1. It encourages better collaboration and communication between development and IT operations teams, which can improve efficiency and reduce errors.
2. Manual processes involved in software delivery are automated, which can also improve efficiency and reduce errors.

Cons:

1. It's challenging to implement for large-scale organizations with legacy operations and teams.
2. It requires investment in new developer tools and infrastructure, which can be costly.

Every day, the number of apps released to the market grows. Plus, right now, there are more mobile devices in the world than there are people. No wonder that by 2026 mobile applications are projected to generate $233 billion in revenue.



With such promising forecasts, it's easy to see why many companies are looking into mobile app development.

Some businesses are interested in creating the next TikTok or Zoom. Thus, focusing on consumer-facing projects and the revenue they can generate. Others, want to optimize internal processes and choose to invest in enterprise solutions that can reduce expenses and improve efficiencies.

Discover the difference between Consumer and Enterprise Mobile Apps

Whatever the case, if developing a mobile solution is of interest to your firm, it's valuable for you to know the process of building an app. Today, we're going to share with you a mobile app creation guide based on our experience and discuss each step of making the app of your dream. Let's get going.

7 Key Steps of Mobile App Development Process

The process of mobile app development can somewhat vary on an individual basis. However, the following seven steps are the ones that you'll likely have to go through:

- Strategy Development
- Analysis and Planning
- UI/UX Design
- App Development
- Application Testing
- Deployment
- Support and Performance Monitoring

So, without further ado, read on to find out what the seven key stages of the mobile app development process are and take into account a few considerations to ensure a smooth flow before you start.

Step 1: Strategy Development

The very first step of the app development process is, unsurprisingly, defining the strategy. At this point, you've got to start thinking carefully about your future application, its goals, capabilities, and business model.

Identify Objectives

Whether you're building a consumer or enterprise app, you'll likely begin by identifying its main objectives. This can be done by answering the following questions:

- What problem will my application solve?
- Who are its target users?
- What results do I want my app to achieve?

The truth is, you might not have to think too much about these questions because you may already have the initial idea for your app. However, it's still good to note the key objectives down so you can always refer back to the goals you are striving for.

Research Competitors

The next step of strategy planning is looking into your existing or potential competitors.

Are there any apps on the market that serve the same purpose? How are they performing? Do they have many installs and positive reviews?

By researching your competition you will be able to bypass mistakes that may already have been made within your industry. Moreover, you'll get an idea about the current landscape of similar mobile apps and be able to determine how you can differentiate from competitors.

Select Platform

At the strategy development phase of the app creation process, you'll also need to decide for which platform you're building your tool. Will it be a custom Android solution? Or, is an iOS app a better pick? Or, perhaps, it is a cross-platform application that you truly need.

We'll later go into more detail about the various options you have, but it's important to already begin thinking about this area of the process.

Discover the Main Differences Between iOS and Android Apps

Choose Monetization Method

Lastly, choosing the right monetization method is only applicable for those delivering a consumer application. In this case, you want to make direct money from your investment, but there are various options for you to pick from, including:

- In-app advertising
- In-app purchases
- Subscriptions
- Affiliate marketing
- Paid apps

The monetization method depends on your goals and the type of an application that you're making. For example, if you're developing a dating app, it's probably not a good idea to charge for downloads. Instead, you'll likely go for in-app purchases and subscriptions.

On the contrary, if you choose to build a game app, chances are you'll leverage in-app advertising along with in-app purchases and subscriptions.

As you can see, everything depends on the purpose of your application. However, it's worth mentioning that the subscription-based monetization model seems to be gaining widespread approval.

Day 2: Overview of iOS and Android Platforms

History and significance of iOS.

iOS, developed by Apple Inc., is a mobile operating system that revolutionized the smartphone industry. Apple iOS is a proprietary mobile operating system that runs on mobile devices such as the iPhone and iPad. It was first introduced in 2007 with the launch of the original iPhone. Over the years, iOS has undergone significant updates, introducing features like the App Store, Siri, and an ever-evolving user interface.

Apple iOS is a proprietary mobile operating system that runs on mobile devices such as the iPhone and iPad.

The significance of iOS lies in its intuitive user experience, seamless integration with other Apple devices, and a robust security framework. This has led to a dedicated user base that values both the hardware and software offerings of Apple.

Apple iOS market share

As of 2023, the Apple iOS market share was 19.9% worldwide, according to IDC. The 2023 iOS market share marks a multi-year high for Apple, surpassing the 18.8% share reported in 2022 and significantly higher than it was in 2019 when IDC reported iOS as having a 13.4% market share.

Apple iOS features

- Wi-Fi, Bluetooth and cellular connectivity, along with VPN support.
- Integrated search support, which enables simultaneous search through files, media, applications and email.
- Gesture recognition supports -- for example, shaking the device to undo the most recent action.
- Push email.
- Safari mobile browser.
- Integrated front- and rear-facing cameras with video capabilities.
- Direct access to the Apple App Store and the iTunes catalog of music, podcasts, television shows and movies available to rent or purchase. iOS is also designed to work with Apple TV.
- Compatibility with Apple's cloud service, iCloud.
- Siri is Apple's virtual assistant that can set reminders, offer suggestions or interact with certain third-party apps. Siri's voice has been modified recently to make it sound more natural.
- Cross-platform communications between Apple devices through AirDrop.
- Support for Apple Watch, runs watchOS 9 but requires iPhone 8 or later running iOS 16 or later.
- Apple Pay, which stores users' credit card data and allows them to pay for goods and services directly with an iOS device.
- CarPlay allows users to interact with an iOS device while driving. CarPlay supports Siri voice controls, and users can access apps through a connected vehicle's touchscreen. CarPlay provides access to maps, phone, calendar, messaging, and music apps.
- The HomePod feature allows Siri to identify family members by voice, giving everyone a personalized experience. HomePod's handoff feature allows users to hand off music, podcasts and phone calls so that they can listen on another device.
- HomeKit allows iOS to be used as a tool for controlling home automation. HomeKit accessories include routers, lights, security cameras and more. The Home app allows you to control these devices from iOS.

Overview of iOS development environment.

The iOS development environment centres around Xcode, Apple's integrated development environment (IDE). Xcode provides a suite of tools for building iOS applications. It includes a code editor, graphical user interface designer, performance analyser, and a simulator for testing.

Developers primarily use the Swift programming language, which is known for its readability and safety features. Swift has become the preferred language for iOS development due to its modern syntax and extensive standard library.

History and significance of Android.

Android, developed by Android Inc. and later acquired by Google, is the world's most widely used mobile operating system. It made its debut in 2008 and has since become the backbone of millions of devices worldwide. Android's open-source nature, customizable interface, and broad device compatibility contribute to its popularity.

The significance of Android lies in its accessibility, allowing a wide range of manufacturers to use and adapt it for their devices. This diversity provides users with a multitude of options in terms of hardware, form factors, and price points.

What are the security and privacy features of Apple iOS?

iOS includes the following security features:

- **Apple ID support.** Users can sign into websites and apps using their existing Apple ID. Additionally, iOS supports signing in using Face ID or Touch ID, which use biometric authentication methods. Apple IDs are protected with two-factor authentication.

- **Privacy and security.** iOS supports fine-grained controls that prevent apps from gaining location information or accepting AirDrop content from unknown senders. Apps can also be blocked from using Wi-Fi or Bluetooth without users' permission. Additionally, iOS devices use a secure boot chain to ensure that only trusted (signed) code is executed during the boot process. This allows iOS devices to verify the integrity of any code running on the device.

- **Secure Enclave Support.** Secure Enclave is a hardware-based feature that stores cryptographic keys in an isolated location to prevent those keys from being compromised. Secure Enclave is not exclusive to iOS devices. It also works with Apple TV, Apple Watch, Mac computers and other Apple products.

Apple iOS version history

Apple iOS was originally known as iPhone OS. The company released three versions of the mobile OS under that name before iOS 4 debuted in June 2010. Apple released iOS 2 on July 11, 2008. It premiered alongside Apple's iPhone 3G. This operating system was followed on June 17, 2009 by iOS 3. The fourth version of iOS was released on June 21, 2010, along with the iPhone 4.

Introduction:

- **Year:** iOS was introduced on June 29, 2007, with the launch of the first iPhone.
- **Significance:** It marked a revolutionary moment in the smartphone industry, combining a touch-friendly interface with powerful features.

Evolution of iOS:

- **Updates:** iOS has undergone numerous updates, each introducing new features, enhancements, and optimizations.
- **Stability:** Apple's closed ecosystem allows for better control over hardware and software, contributing to the system's stability.

App Store:

- **Launch:** The App Store was introduced in 2008.
- **Impact:** It transformed the way software is distributed and consumed, creating a thriving ecosystem for developers and users alike.

User Interface:

- **Intuitive Design:** iOS is renowned for its user-friendly interface, characterized by simplicity, elegance, and a focus on user experience.
- **Influence:** Many competitors adopted elements of iOS design principles.

Security and Privacy:

- **Emphasis:** iOS places a strong emphasis on security and privacy.
- **End-to-End Encryption:** Features like iMessage are end-to-end encrypted, enhancing user privacy.

Integration with Apple Ecosystem:

- **Synergy:** iOS seamlessly integrates with other Apple devices and services, creating a unified ecosystem.
- **Cross-Device Functionality:** Features like Handoff and Continuity allow users to transition between devices seamlessly.

Developer Community:

- **Robust:** iOS has a thriving developer community, contributing to a vast array of high-quality apps.
- **Monetization:** The App Store's success has provided developers with a platform for monetization.

iOS Devices:

- **Expansion:** iOS powers not only iPhones but also iPads, iPod Touch, and Apple Watch.
- **Consistency:** The consistent user experience across devices contributes to brand loyalty.

iOS in Enterprise:

- **Adoption:** iOS has gained traction in enterprise environments due to its security features and business-friendly applications.
- **Mobile Device Management (MDM):** Businesses can deploy and manage iOS devices efficiently.

Global Impact: - **Cultural Impact:** iOS devices have become cultural phenomena, influencing how people communicate, work, and entertain themselves. - **Market Dominance:** iPhones are among the most popular smartphones globally.

Overview of Android development environment.

Android development primarily takes place in Android Studio, the official IDE for Android development. It offers a powerful suite of tools for designing, coding, testing, and debugging Android applications. This includes a code editor, layout editor, emulator for testing, and a wealth of libraries and frameworks.

Java and Kotlin are the primary programming languages used for Android development. Kotlin, introduced by JetBrains, has gained popularity for its conciseness and interoperability with existing Java code.

On the following days, we'll delve deeper into the development environments, languages, and best practices for both iOS and Android platforms.

Day 3: Setting up Integrated Development Environment

Using Command Line for Kotlin Development

Before we dive into Integrated Development Environments (IDEs), let's familiarize ourselves with the basics of Kotlin development using the command line. This approach provides a foundational understanding of how Kotlin projects are set up and managed.

Creating a Kotlin Project from the Command Line

Introduction to IDE's

Integrated Development Environments (IDEs) are sophisticated software applications designed to facilitate and streamline the process of software development. They provide a comprehensive suite of tools and features that are essential for efficient coding, testing, debugging, and deploying applications.

Why are IDEs important?

You can use any text editor to write code. However, most integrated development environments (IDEs) include functionality that goes beyond text editing. They provide a central interface for common developer tools, making the software development process much more efficient. Developers can start programming new applications quickly instead of manually integrating and configuring different software. They also don't have to learn about all the tools and can instead focus on just one application. The following are some reasons why developers use IDEs:

Code editing automation

Programming languages have rules for how statements must be structured. Because an IDE knows these rules, it contains many intelligent features for automatically writing or editing the source code.

Syntax highlighting

An IDE can format the written text by automatically making some words bold or italic, or by using different font colors. These visual cues make the source code more readable and give instant feedback about accidental syntax errors.

Intelligent code completion

Various search terms show up when you start typing words in a search engine. Similarly, an IDE can make suggestions to complete a code statement when the developer begins typing.

Refactoring support

Code refactoring is the process of restructuring the source code to make it more efficient and readable without changing its core functionality. IDEs can auto-refactor to some extent, allowing developers to improve their code quickly and easily. Other team members understand readable code faster, which supports collaboration within the team.

Local build automation

IDEs increase programmer productivity by performing repeatable development tasks that are typically part of every code change. The following are some examples of regular coding tasks that an IDE carries out.

Compilation

An IDE compiles or converts the code into a simplified language that the operating system can understand. Some programming languages implement just-in-time compiling, in which the IDE converts human-readable code into machine code from within the application.

Testing

The IDE allows developers to automate unit tests locally before the software is integrated with other developers' code and more complex integration tests are run.

Debugging

Debugging is the process of fixing any errors or bugs that testing reveals. One of the biggest values of an IDE for debugging purposes is that you can step through the code, line by line, as it runs and inspect code behavior. IDEs also integrate several debugging tools that highlight bugs caused by human error in real time, even as the developer is typing.

What are the types of IDEs?

Integrated development environments (IDEs) can be broadly classified into several different categories, depending on the application development they support and how they work. However, many IDE software applications can fit into multiple categories. The following are some types of IDEs:

Local IDEs

Developers install and run local IDEs directly on their local machines. They also have to download and install various additional libraries depending on their coding preferences, project requirements, and development language. While local IDEs are customizable and do not require an internet connection once installed, they present several challenges:

- They can be time consuming and difficult to set up.
- They consume local machine resources and can slow down machine performance significantly.
- Configuration differences between the local machine and the production environment can give rise to software errors.

Cloud IDEs

Developers use cloud IDEs to write, edit, and compile code directly in the browser so that they don't need to download software on their local machines. Cloud-based IDEs have several advantages over traditional IDEs. The following are some of these advantages:

Standardized development environment

Software development teams can centrally configure a cloud-based IDE to create a standard development environment. This method helps them avoid errors that might occur due to local machine configuration differences.

Platform independence

Cloud IDEs work on the browser and are independent of local development environments. This means they connect directly to the cloud vendor's platform, and developers can use them from any machine.

Better performance

Building and compiling functions in an IDE requires a lot of memory and can slow down the developer's computer. The cloud IDE uses compute resources from the cloud and frees up the local machine's resources.

How should I choose an IDE?

You can find many modern integrated development environments (IDEs) on the market with a range of features and different price points. Many IDEs are open source, or free to use and configure. The following are some criteria to consider when choosing an IDE:

The programming language

The programming language you want to code in often dictates the choice of an IDE. Dedicated IDEs have automation features that particularly suit the syntax of specific languages. On the other hand, multi-language IDEs support multiple languages.

The operating system

While most IDEs have multiple versions for different operating systems, they might work better on specific platforms. For example, some IDEs can perform optimally on the Linux platform but might be slow or difficult to use on other platforms.

Automation features

The three common features in most IDES are the source code editor, build automation, and debugger. Additional features may vary and can include the following:

- Code editor UI enhancements
- Automated testing features
- Code deployment support via plugin integration
- Code refactoring support
- Application packaging support

IDE customization

Some IDEs include the ability to customize workflows to match a developer's needs and preferences. You can download and use plugins, extensions, and add-ons to customize your programming experience.

Why is Android development Kotlin-first?

We reviewed feedback that came directly from developers at conferences, our Customer Advisory Board (CAB), Google Developer Experts (GDE), and through our developer research. Many developers already enjoy using Kotlin, and the request for more Kotlin support was clear. Here's what developers appreciate about writing in Kotlin:

- **Expressive and concise:** You can do more with less. Express your ideas and reduce the amount of boilerplate code. 67% of professional developers who use Kotlin say Kotlin has increased their productivity.
- **Safer code:** Kotlin has many language features to help you avoid common programming mistakes such as null pointer exceptions. Android apps that contain Kotlin code are 20% less likely to crash.

- **Interoperable:** Call Java-based code from Kotlin, or call Kotlin from Java-based code. Kotlin is 100% interoperable with the Java programming language, so you can have as little or as much of Kotlin in your project as you want.
- **Structured Concurrency:** Kotlin coroutines make asynchronous code as easy to work with as blocking code. Coroutines dramatically simplify background task management for everything from network calls to accessing local data.

What does Kotlin-first mean?

When building new Android development tools and content, such as Jetpack libraries, samples, documentation, and training content, we will design them with Kotlin users in mind while continuing to provide support for using our APIs from the Java programming language.

	Java language	**Kotlin**
Platform SDK support	Yes	Yes
Android Studio support	Yes	Yes
Lint	Yes	Yes
Guided docs support	Yes	Yes
API docs support	Yes	Yes
AndroidX support	Yes	Yes
AndroidX Kotlin-specific APIs (KTX, coroutines, and so on)	N/A	Yes
Online training	Best effort	Yes
Samples	Best effort	Yes
Multi-platform projects	No	Yes
Jetpack Compose	No	Yes
Compiler plugin support	No	Yes -

The Kotlin Symbol Processing API was created by Google to develop lightweight compiler plugins.

Setting up IntelliJ IDEA for Kotlin Development

Setting up IntelliJ IDEA for Kotlin Development

IntelliJ IDEA is a powerful IDE that provides a range of features to enhance your Kotlin development experience. Here are some additional steps for setting up IntelliJ IDEA:

Install Plugins: Explore the available plugins in IntelliJ IDEA's marketplace. Numerous plugins are available for different purposes, from version control to code analysis.

Customize Code Styles: Configure code styles and formatting options according to your preferences. Consistent code style improves code readability and maintainability.

Configuring Android Studio for Kotlin

Android Studio, as the official IDE for Android development, seamlessly supports Kotlin. Here are the detailed steps to set up Android Studio for Kotlin development:

Using Kotlin in Existing Projects

Open an Existing Android Project:

Launch Android Studio.

Click "Open an existing Android Studio project" in the Welcome screen or go to File > Open.

Locate and Select Your Project:

Browse your file system and select the folder containing your existing Android project.

Click "OK" to open the project.

Convert Java Files to Kotlin:

Once your project is open, navigate to a Java file that you want to convert to Kotlin.

Right-click on the file, and from the context menu, select Convert Java File to Kotlin File.

Convert Java to Kotlin

Android Studio will perform the conversion and create a Kotlin version of the file.

Kotlin File Created

Review the generated Kotlin code to ensure accuracy.

Setting up Emulators

Android emulators are essential for testing applications on virtual devices before deploying them to real devices. Here's how you can set up and configure emulators in Android Studio:

Launch Android Studio:

Open the AVD Manager:

Click on the "AVD Manager" icon in the toolbar or go to View > Tool Windows > AVD Manager.

AVD Manager

Create a New Virtual Device:

In the AVD Manager window, click on the "Create Virtual Device" button.

Create Virtual Device

Select a Device Definition:

Choose a device definition from the list. For example, you can select a Pixel device.

Select Device Definition

Select a System Image:

Choose a system image (Android version) that you want to use for the emulator.

Select System Image

Configure AVD Options:

Customize additional settings for the virtual device if needed, then click "Finish".

Configure AVD Options

Launch the Emulator:

In the AVD Manager, select your newly created virtual device and click on the "Play" button.

Launch Emulator

The emulator will start up, allowing you to test your applications on various virtual devices.

Day 4: Introduction to programming

Definition of Programming

Programming is the process of giving precise instructions to a computer to perform specific tasks or solve problems. These instructions are written in a specific programming language, which serves as a medium of communication between humans and computers.

Programming refers to a technological process for telling a computer which tasks to perform in order to solve problems. You can think of programming as a collaboration between humans and computers, in which humans create instructions for a computer to follow (code) in a language computers can understand.

Programming enables so many things in our lives. Here are some examples:

- When you browse a website to find information, contact a service provider, or make a purchase, programming allows you to interact with the site's on-page elements, such as sign-up or purchase buttons, contact forms, and drop-down menus.
- The programming behind a mobile app can make it possible for you to order food, book a rideshare service, track your fitness, access media, and more with ease.
- Programming helps businesses operate more efficiently through different software for file storage and automation and video conferencing tools to connect people globally, among other things.
- Space exploration is made possible through programming.

1. *Giving instructions to a computer*

Programming involves creating a set of step-by-step instructions that a computer can understand and execute. These instructions can range from simple tasks like adding two numbers to complex operations like simulating real-world phenomena.

2. *Written in a specific language.*

Programming languages are designed to facilitate human-computer interaction. Each language has its own syntax and rules, allowing programmers to express their ideas and solutions in a structured and understandable manner. Examples of popular programming languages include Python, Java, C++, and JavaScript.

Importance of Programming

Programming plays a crucial role in various aspects of modern life, and its significance continues to grow in our technologically driven world.

1. *Problem-solving*

programming is like being a detective. You're given a puzzle, and you must figure out the best way to solve it. This could be anything from creating a simple calculator to predicting the weather.

For instance, think about those weather apps. They crunch tons of data and give us accurate forecasts. All thanks to programming!

2. Automation

Ever wished you could have a personal assistant to handle all those boring, repetitive tasks? That's where programming comes in. It lets us create systems that do the boring stuff for us.
Think about it, in online stores, there are systems that manage orders, handle payments, and update inventory. All automatic!

3 Creativity and Innovation

With programming, you can let your imagination run wild! You can create games, websites, apps - the possibilities are endless. It's like being an artist, but with lines of code!

Imagine all those fun mobile apps we use daily. From social media to fitness trackers, they're all born out of someone's creative coding.

4. Job Opportunities

Knowing how to program opens a whole world of job opportunities. There's a huge demand for folks who can code, and it's only growing. You can land gigs as a software developer, data scientist, and more. Just think, tech companies are always on the lookout for skilled programmers. So, not only is it a cool skill, but it can also land you some sweet jobs.

Day 5: Basic Concepts in Programming

Introduction to Kotlin

Kotlin is a versatile programming language that's gaining popularity for its concise syntax and powerful features. Whether you're a beginner or an experienced programmer, Kotlin is a great language to learn.

Kotlin is a modern, trending programming language that was released in 2016 by JetBrains.

It has become very popular since it is compatible with Java (one of the most popular programming languages out there), which means that Java code (and libraries) can be used in Kotlin programs.

Kotlin is used for:

- Mobile applications (specially Android apps)
- Web development
- Server side applications
- Data science
- And much, much more!

Why Use Kotlin?

- Kotlin is fully compatible with Java
- Kotlin works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)
- Kotlin is concise and safe
- Kotlin is easy to learn, especially if you already know Java
- Kotlin is free to use

Big community/supp

Variables and Data Types

variables and data types. These are like the building blocks of any program. They help us store and manipulate information.

1. *Definition of Variables*

Think of it as a container that holds information. This information can be anything - numbers, text, true/false statements, you name it. The cool thing is you can change what's inside this container as many times as you want.

For example, imagine you have a variable called name, and it holds the value "Ahmed". Later, you can change it to "Ali" if you want. That's the power of variables!

2. *Different Data Types*

Now, not all information is the same. We categorize them into different types, depending on what kind of data they represent. Here are some common data types:

Integers (Int): These are whole numbers without decimal points. For instance, 1, 10, -5.

Floating-Point Numbers (Float, Double): These include numbers with decimal points. Like 3.14 or 2.71828.

Booleans (Boolean): This one's easy - it's either true or false. It's used for decision-making in programs.
Strings (String): These are sequences of characters, like words or sentences. "Hello, World!" is an example.
Characters (Char): This type stores a single character, like 'a' or '$'.
Arrays: They're like collections of variables. You can have an array of numbers, or an array of names.
Null: Sometimes, a variable might not have any value. In those cases, we use null to represent the absence of data.
Let's say you're working on a program to calculate grades. You might use integers for the scores, strings for the names, and Booleans to check if a student passed or failed.
Remember, picking the right data type is crucial. It helps save memory and ensures your program runs efficiently.

In Kotlin, the type of a variable is decided by its value:

Example

```
val myNum = 5           // Int

val myDoubleNum = 5.99    // Double

val myLetter = 'D'       // Char

val myBoolean = true      // Boolean

val myText = "Hello"      // String
```

Data types are divided into different groups:

- Numbers
- Characters
- Booleans
- Strings
- Arrays

Numbers

Number types are divided into two groups:

Integer types store whole numbers, positive or negative (such as 123 or -456), without decimals. Valid types are Byte, Short, Int and Long.

Floating point types represent numbers with a fractional part, containing one or more decimals. There are two types: Float and Double.

If you don't specify the type for a numeric variable, it is most often returned as Int for whole numbers and Double for floating point numbers.

Integer Types

Byte

The Byte data type can store whole numbers from -128 to 127. This can be used instead of Int or other integer types to save memory when you are certain that the value will be within -128 and 127:

Example

```
val myNum: Byte = 100
println(myNum)
```

Short

The Short data type can store whole numbers from -32768 to 32767:

Example

```
val myNum: Short = 5000
println(myNum)
```

Int

The Int data type can store whole numbers from -2147483648 to 2147483647:

Example

```
val myNum: Int = 100000
println(myNum)
```

Long

The Long data type can store whole numbers from -9223372036854775807 to 9223372036854775807. This is used when Int is not large enough to store the value. Optionally, you can end the value with an "L":

Example

```
val myNum: Long = 15000000000L
println(myNum)
```

Difference Between Int and Long

A whole number is an Int as long as it is up to 2147483647. If it goes beyond that, it is defined as Long:

Example

```
val myNum1 = 2147483647  // Int
val myNum2 = 2147483648  // Long
```

Floating Point Types

Floating point types represent numbers with a decimal, such as 9.99 or 3.14515.

The Float and Double data types can store fractional numbers:

Float Example

```
val myNum: Float = 5.75F
println(myNum)
```

Double Example

```
val myNum: Double = 19.99
println(myNum)
```

Use Float or Double?

The **precision** of a floating point value indicates how many digits the value can have after the decimal point. The precision of Float is only six or seven decimal digits, while Double variables have a precision of about 15 digits. Therefore it is safer to use Double for most calculations.

Also note that you should end the value of a Float type with an "F".

Scientific Numbers

A floating point number can also be a scientific number with an "e" or "E" to indicate the power of 10:

Example

```
val myNum1: Float = 35E3F
val myNum2: Double = 12E4
println(myNum1)
println(myNum2)
```

Booleans

The Boolean data type and can only take the values true or false:

Example

```
val isKotlinFun: Boolean = true
val isFishTasty: Boolean = false
println(isKotlinFun)   // Outputs true
println(isFishTasty)   // Outputs false
```

Boolean values are mostly used for conditional testing, which you will learn more about in a later chapter.

Characters

The Char data type is used to store a **single** character. A char value must be surrounded by **single** quotes, like 'A' or 'c':

Example

```
val myGrade: Char = 'B'

println(myGrade)
```

Unlike Java, you cannot use ASCII values to display certain characters. The value 66 would output a "B" in Java, but will generate an error in Kotlin:

Example

```
val myLetter: Char = 66

println(myLetter) // Error
```

Strings

The String data type is used to store a sequence of characters (text). String values must be surrounded by **double** quotes:

Example

```
val myText: String = "Hello World"

println(myText)
```

You will learn more about strings in the Strings chapter.

Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

You will learn more about arrays in the Arrays chapter.

Type Conversion

Type conversion is when you convert the value of one data type to another type.

In Kotlin, numeric type conversion is different from Java. For example, it is not possible to convert an Int type to a Long type with the following code:

Example

```
val x: Int = 5

val y: Long = x

println(y) // Error: Type mismatch
```

To convert a numeric data type to another type, you must use one of the following functions: toByte(), toShort(), toInt(), toLong(), toFloat(), toDouble() or toChar():

Example

```
val x: Int = 5

val y: Long = x.toLong()

println(y)
```

Week-2 – Introduction to Programming Concepts

Day 1: Introduction to Programming Concepts

Definition of variables and data types.

In programming, a variable is a storage location identified by a memory address and an associated symbolic name (an identifier), which contains some known or unknown quantity of information referred to as a value. Variables are used to store and manipulate data in a program.

Creating different programs on different data types.

Data Types

Data types define the type of data a variable can hold. They also determine what operations can be performed on the data. In Kotlin, there are several basic data types:

Integers (Whole numbers without a decimal point):

Byte: 8-bit signed integer (-128 to 127)

Short: 16-bit signed integer (-32768 to 32767)

Int: 32-bit signed integer (-2^31 to 2^31-1)

Long: 64-bit signed integer (-2^63 to 2^63-1)

Floating-point Numbers (Numbers with a decimal point):

Float: 32-bit floating point (6-7 decimal digits precision)

Double: 64-bit floating point (15 decimal digits precision)

Characters (A single Unicode character):

Char: 16-bit Unicode character

Boolean (Represents true or false values):

Boolean: true or false

Strings (A sequence of characters):

String: A collection of characters

Examples in the Kotlin language.

```
fun main() {
    // Integers
    val myByte: Byte = 127
    val myShort: Short = 32767
    val myInt: Int = 2147483647
    val myLong: Long = 9223372036854775807L

    // Floating-point numbers
    val myFloat: Float = 3.14f
    val myDouble: Double = 3.141592653589793

    // Characters
    val myChar: Char = 'A'

    // Boolean
    val myBoolean: Boolean = true

    // Strings
    val myString: String = "Hello, World!"

    println("Byte: $myByte")
    println("Short: $myShort")
    println("Int: $myInt")
    println("Long: $myLong")
    println("Float: $myFloat")
    println("Double: $myDouble")
    println("Char: $myChar")
    println("Boolean: $myBoolean")
    println("String: $myString")
}
```

Kotlin Program to Print an Integer (Entered by the User)

Example 1: How to Print an Integer entered by an user in Kotlin using Scanner

```
import java.util.Scanner

fun main(args: Array<String>) {

    // Creates a reader instance which takes
    // input from standard input - keyboard
    val reader = Scanner(System.`in`)
    print("Enter a number: ")

    // nextInt() reads the next integer from the keyboard
    var integer:Int = reader.nextInt()

    // println() prints the following line to the output screen
    println("You entered: $integer")
```

```
}
```

Output:

```
Enter a number: 10
You entered: 10
```

Example 2: How to Print an Integer without using Scanner

```
fun main(args: Array<String>) {

    print("Enter a number: ")

    // reads line from standard input - keyboard
    // and !! operator ensures the input is not null
    val stringInput = readLine()!!

    // converts the string input to integer
    var integer:Int = stringInput.toInt()

    // println() prints the following line to the output screen
    println("You entered: $integer")
}
```

Output:

```
Enter a number: 10
You entered: 10
```

Kotlin Program to Add Two Integers

Example: Kotlin Program to Add Two Integers

```
fun main(args: Array<String>) {

    val first: Int = 10
    val second: Int = 20

    val sum = first + second

    println("The sum is: $sum")
```

```
}
```

When you run the program, the output will be:

```
The sum is: 30
```

Kotlin Program to Multiply two Floating Point Numbers

Example: Multiply Two Floating Point Numbers

```
fun main(args: Array<String>) {

    val first = 1.5f
    val second = 2.0f

    val product = first * second

    println("The product is: $product")
}
```

When you run the program, the output will be:

```
The product is: 3.0
```

Kotlin Program to Find ASCII value of a character

Example: Find ASCII value of a character

```
fun main(args: Array) {

    val c = 'a'
    val ascii = c.toInt()

    println("The ASCII value of $c is: $ascii")
}
```

Output:

```
The ASCII value of a is: 97
```

Kotlin Program to Compute Quotient and Remainder

Example: Compute Quotient and Remainder

```
fun main(args: Array<String>) {

    val dividend = 25
    val divisor = 4

    val quotient = dividend / divisor
    val remainder = dividend % divisor

    println("Quotient = $quotient")
    println("Remainder = $remainder")
}
```

When you run the program, the output will be:

```
Quotient = 6
Remainder = 1
```

Kotlin Program to Swap Two Numbers

Example 1: Swap two numbers using temporary variable

```
fun main(args: Array<String>) {

    var first = 1.20f
    var second = 2.45f

    println("--Before swap--")
    println("First number = $first")
    println("Second number = $second")

    // Value of first is assigned to temporary
    val temporary = first

    // Value of second is assigned to first
    first = second
```

```
    // Value of temporary (which contains the initial value of first) is assigned to second
    second = temporary

    println("--After swap--")
    println("First number = $first")
    println("Second number = $second")
}
```

When you run the program, the output will be:

```
--Before swap--
First number = 1.2
Second number = 2.45
--After swap--
First number = 2.45
Second number = 1.2
```

Kotlin Program to Check Whether a Number is Even or Odd

Example 1: Check whether a number is even or odd using if...else statement

```
import java.util.*

fun main(args: Array<String>) {

    val reader = Scanner(System.`in`)

    print("Enter a number: ")
    val num = reader.nextInt()

    if (num % 2 == 0)
        println("$num is even")
    else
        println("$num is odd")
}
```

When you run the program, the output will be:

```
Enter a number: 12
```

```
12 is even
```

Example 2: Check whether a number is even or odd using if...else expression

```
import java.util.*

fun main(args: Array<String>) {

    val reader = Scanner(System.`in`)

    print("Enter a number: ")
    val num = reader.nextInt()

    val evenOdd = if (num % 2 == 0) "even" else "odd"

    println("$num is $evenOdd")
}
```

When you run the program, the output will be:

```
Enter a number: 13
13 is odd
```

Kotlin Program to Find the Frequency of Character in a String

Example: Find Frequency of Character

```
fun main(args: Array<String>) {
    val str = "This website is awesome."
    val ch = 'e'
    var frequency = 0

    for (i in 0..str.length - 1) {
        if (ch == str[i]) {
            ++frequency
        }
    }

    println("Frequency of $ch = $frequency")
}
```

When you run the program, the output will be:

```
Frequency of e = 4
```

Kotlin Program to Convert Character to String and Vice-Versa

Example 1: Convert char to String

```
fun main(args: Array<String>) {
    val ch = 'c'
    val st = Character.toString(ch)
    // Alternatively
    // st = String.valueOf(ch);

    println("The string is: $st")
}
```

When you run the program, the output will be:

```
The string is: c
```

Kotlin Program to Calculate Average Using Arrays

Example: Program to Calculate Average Using Arrays

```
fun main(args: Array<String>) {
    val numArray = doubleArrayOf(45.3, 67.5, -45.6, 20.34, 33.0, 45.6)
    var sum = 0.0

    for (num in numArray) {
        sum += num
    }

    val average = sum / numArray.size
    println("The average is: %.2f".format(average))
}
```

When you run the program, the output will be:

```
The average is: 27.69
```

Kotlin Program to Find Largest Element in an Array

Example: Find largest element in an array

```
fun main(args: Array<String>) {
    val numArray = doubleArrayOf(23.4, -34.5, 50.0, 33.5, 55.5, 43.7, 5.7, -
66.5)
    var largest = numArray[0]

    for (num in numArray) {
        if (largest < num)
            largest = num
    }

    println("Largest element = %.2f".format(largest))
}
```

When you run the program, the output will be:

```
Largest element = 55.50
```

Day 2: Control Structures

Conditional Statements (if-else)

Conditional statements in Kotlin allow you to execute different blocks of code based on certain conditions. They are crucial for controlling the flow of your program.

Kotlin supports the usual logical conditions from mathematics:

- Less than: a < b
- Less than or equal to: a <= b

- Greater than: a > b
- Greater than or equal to: a >= b
- Equal to a == b
- Not Equal to: a != b

You can use these conditions to perform different actions for different decisions.

Kotlin has the following conditionals:

- Use if to specify a block of code to be executed, if a specified condition is true
- Use else to specify a block of code to be executed, if the same condition is false
- Use else if to specify a new condition to test, if the first condition is false
- Use when to specify many alternative blocks of code to be executed

Note: Unlike Java, if..else can be used as a **statement** or as an **expression** (to assign a value to a variable) in Kotlin. See an example at the bottom of the page to better understand it.

Kotlin if

Use if to specify a block of code to be executed if a condition is true.

Syntax

if (*condition*) {

  // block of code to be executed if the condition is true

}

Note that if is in lowercase letters. Uppercase letters (If or IF) will generate an error.

In the example below, we test two values to find out if 20 is greater than 18. If the condition is true, print some text:

Example

if (20 > 18) {

  println("20 is greater than 18")

}

Basic **if-else**:

```
val num = 10

if (num > 5) {
    println("The number is greater than 5")
} else {
    println("The number is less than or equal to 5")
}
```

Handling Multiple Conditions:

```
val grade = 75

if (grade >= 90) {
    println("A")
} else if (grade >= 80) {
    println("B")
} else if (grade >= 70) {
    println("C")
} else {
    println("F")
}
```

Ternary Operator (Conditional Expression):

```
val result = if (num > 5) "Greater" else "Less or Equal"
```

Learning loop structures for repetitive tasks.

Loops (for, while)

Loops allow you to execute a block of code repeatedly. Kotlin supports both for and while loops.

For Loop:

The for loop is used when you know in advance how many times you want to repeat a block of code.

```
for (i in 1..5) {
    println("Count: $i")
```

You can also loop through collections:

```
val fruits = listOf("Apple", "Banana", "Cherry")

for (fruit in fruits) {
    println(fruit)
}
```

While Loop:

The while loop is used when you want to execute a block of code as long as a condition is true.

```
var i = 1
while (i <= 5) {
    println("Count: $i")
    i++
}
```

Do-While Loop:

The do-while loop is similar to a while loop, but it guarantees at least one execution of the block.

```
var i = 1
do {
    println("Count: $i")
    i++
} while (i <= 5)
```

Break and Continue:

break is used to exit a loop prematurely.

continue is used to skip the rest of the loop and start the next iteration.

Example (Break):

```
for (i in 1..10) {
    if (i == 5) {
        break
    }
    println("Count: $i")
}
```

Example (Continue):

```
for (i in 1..5) {
    if (i == 3) {
        continue
    }
    println("Count: $i")
}
```

You can have loops within loops:

```
for (i in 1..3) {
    for (j in 1..3) {
        println("$i, $j")
    }
}
```

return - exits the current function.

label - provides a way to name loops and control them.

```
loop@
for (i in 1..3) {
    for (j in 1..3) {
        if (i == 2 && j == 2)
 break@loop
        println("$i, $j")
    }
}
```

When expression

when defines a conditional expression with multiple branches. It is similar to the switch statement in C-like languages. Its simple form looks like this.

```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> {
        print("x is neither 1 nor 2")
    }
}
```

when matches its argument against all branches sequentially until some branch condition is satisfied.

when can be used either as an expression or as a statement. If it is used as an expression, the value of the first matching branch becomes the value of the overall expression. If it is used as a statement, the values of individual branches are ignored. Just like with if, each branch can be a block, and its value is the value of the last expression in the block.

The else branch is evaluated if none of the other branch conditions are satisfied.

If when is used as an expression, the else branch is mandatory, unless the compiler can prove that all possible cases are covered with branch conditions, for example, with enum class entries and sealed class subtypes).

```
enum class Bit {
    ZERO, ONE
}

val numericValue = when (getRandomBit()) {
    Bit.ZERO -> 0
    Bit.ONE -> 1
    // 'else' is not required because all cases are covered
```

```
}
```

In when statements, the else branch is mandatory in the following conditions:

- when has a subject of a Boolean, enum, or sealed type, or their nullable counterparts.
- branches of when don't cover all possible cases for this subject.

```
enum class Color {
    RED, GREEN, BLUE
}

when (getColor()) {
    Color.RED -> println("red")
    Color.GREEN -> println("green")
    Color.BLUE -> println("blue")
    // 'else' is not required because all cases are covered
}

when (getColor()) {
    Color.RED -> println("red") // no branches for GREEN and BLUE
    else -> println("not red") // 'else' is required
}
```

To define a common behavior for multiple cases, combine their conditions in a single line with a comma:

```
when (x) {
    0, 1 -> print("x == 0 or x == 1")
    else -> print("otherwise")
}
```

You can use arbitrary expressions (not only constants) as branch conditions

```
when (x) {
    s.toInt() -> print("s encodes x")
    else -> print("s does not encode x")
}
```

You can also check a value for being in or !in a range or a collection:

```
when (x) {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is outside the range")
    else -> print("none of the above")
}
```

Another option is checking that a value is or !is of a particular type. Note that, due to smart casts, you can access the methods and properties of the type without any extra checks.

```
fun hasPrefix(x: Any) = when(x) {
    is String -> x.startsWith("prefix")
    else -> false
}
```

when can also be used as a replacement for an if-else if chain. If no argument is supplied, the branch conditions are simply boolean expressions, and a branch is executed when its condition is true:

```
when {
    x.isOdd() -> print("x is odd")
    y.isEven() -> print("y is even")
    else -> print("x+y is odd")
}
```

You can capture when subject in a variable using following syntax:

```
fun Request.getBody() =
    when (val response = executeRequest()) {
        is Success -> response.body
        is HttpError -> throw HttpException(response.status)
    }
```

The scope of variable introduced in when subject is restricted to the body of this when.

Day 3: Programming Logic and Algorithms

Logic Building

Logic building is an essential skill for any programmer. It involves breaking down complex problems into smaller, manageable tasks and then systematically solving them.

Developing logical thinking for problem-solving.

Decomposing Problems:

Break down a complex problem into smaller, more manageable parts.

Understand the relationships and dependencies between different components.

Pseudocode:

Write a high-level description of your solution in plain language before implementing it in code.

Helps you outline the steps and logic needed to solve the problem.

Flowcharts:

Visual representations of a program's flow and logic.

Useful for planning and understanding complex algorithms.

Abstraction:

Focus on the high-level solution first before diving into the implementation details.

Avoid getting lost in the code and lose sight of the overall goal.

Pattern Recognition:

Recognize recurring patterns or structures in problems and leverage them for solutions.

Introduction to Algorithms

An algorithm is a step-by-step set of instructions or rules to accomplish a specific task or solve a particular problem. It's a well-defined process that, when followed, leads to a desired outcome. Algorithms are fundamental in computer science and programming as they provide a systematic approach to tackling complex tasks.

Understanding basic algorithms and their importance.

Importance of Algorithms:

Efficient algorithms lead to faster and more resource-effective solutions.

Well-designed algorithms are essential for optimizing code and minimizing resource usage.

Algorithm Analysis:

Evaluate the efficiency of an algorithm in terms of time and space complexity.

Understand Big O notation to quantify an algorithm's performance.

Common Algorithms:

Familiarize yourself with basic algorithms like sorting, searching, and mathematical operations.

Understand their implementation and when to use them.

Data Structures and Algorithms:

Learn how different data structures (e.g., arrays, linked lists, trees) influence algorithm choice and efficiency.

Problem-Solving Techniques:
Understand common techniques like recursion, dynamic programming, and divide-and-conquer.

Day 4: Introduction to the Kotlin Programming Language

Kotlin Basics

Basic syntax and structure.

Variables and Data Types in the Chosen Language

Practice using variables and different data types.

Day 5: Functions or Methods

Purpose of Functions

In programming, functions (or methods, in some languages) are blocks of reusable code that perform a specific task. They encapsulate a set of operations and allow you to execute them by calling the function's name.

Modularity: Functions allow you to break down complex programs into smaller, manageable parts. Each function can focus on a specific task, making your code easier to understand and maintain.

Reusability: Once you define a function, you can use it multiple times in your program. This reduces redundancy and promotes code reuse.

Abstraction: Functions allow you to use high-level logic without getting bogged down by implementation details. You can think of a function as a black box that performs a specific task.

Testing and Debugging: Functions make it easier to test and debug your code. You can isolate specific functionalities and test them independently.

Parameters and Return Values

Parameters:

Functions can accept input values called parameters or arguments. These values provide the necessary data for the function to perform its task.

```
fun greet(name: String) {
    println("Hello, $name!")
}
```

Return Values:
Functions can also return a value after performing their task. This allows you to use the result in other parts of your program.

```
fun add(a: Int, b: Int): Int {
    return a + b
}
```

Writing different types of functions

1. *Functions without Parameters and Return Values*

```
fun sayHello() {
    println("Hello!")
}
```

2.Functions with Parameters and Return Values:

```
fun add(a: Int, b: Int): Int {
    return a + b
}
```

3.Functions with Default Parameters:

```
fun greet(name: String = "User") {
    println("Hello, $name!")
}
```

4. Functions with Named Parameters:

```
fun divide(dividend: Int, divisor: Int) {
    // ...
}
// Calling with named parameters
divide(dividend = 10, divisor = 2)
```

Examples:

```
fun main() {
    sayHello() // Outputs: "Hello!"

    val sum = add(3, 4) // sum = 7

    greet() // Outputs: "Hello, User!"
    greet("Alice") // Outputs: "Hello, Alice!"
```

Project Discussion

Week-3 Data Structures and Algorithms in Kotlin

Introduction to Data Structures

A data structure is a particular way of organising data in a computer so that it can be used effectively. The idea is to reduce the space and time complexities of different tasks.
The choice of a good data structure makes it possible to perform a variety of critical operations effectively. An efficient data structure also uses minimum memory space and execution time to process the structure. A data structure is not only used for organising the data. It is also used for processing, retrieving, and storing data. There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed. So we must have good knowledge of data structures.
Need Of Data Structure:
The structure of the data and the synthesis of the algorithm are relative to each other. Data presentation must be easy to understand so the developer, as well as the user, can make an efficient implementation of the operation.
Data structures provide an easy way of organising, retrieving, managing, and storing data.
Here is a list of the needs for data.

- Data structure modification is easy.
- It requires less time.
- Save storage memory space.
- Data representation is easy.
- Easy access to the large database

Classification/Types of Data Structures:

1. Linear Data Structure
2. Non-Linear Data Structure.

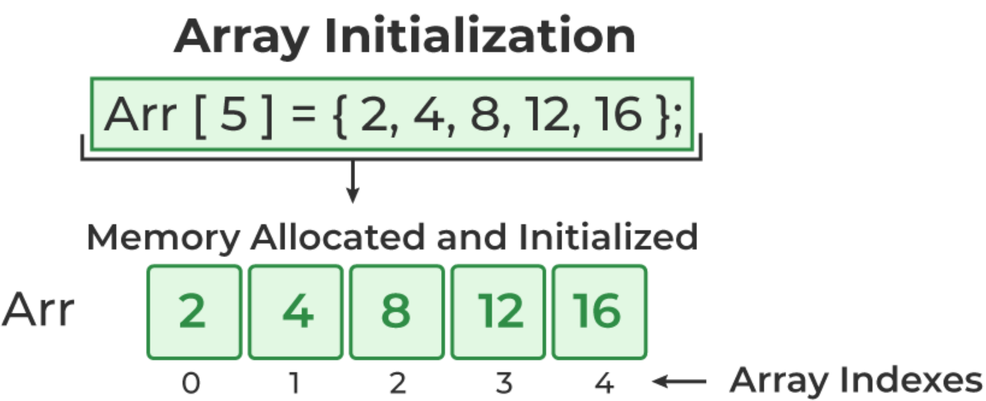
Linear Data Structure:

- Elements are arranged in one dimension ,also known as linear dimension.
- Example: lists, stack, queue, etc.

Non-Linear Data Structure

- Elements are arranged in one-many, many-one and many-many dimensions.
- Example: tree, graph, table, etc.

Day 1: Introduction to Kotlin Arrays



Array is one of the most fundamental data structure in practically all programming languages. The idea behind an array is to store multiple items of the same data-type,such as an *integer* or *string* , under a single variable name.
Arrays are used to organize data in programming so that a related set of values can be easily sorted or searched.
Here are some basic properties of arrays –

They are stored in contiguous memory locations.
They can be accessed programmatically through their indexes (array[1], array[0], etc.)
They are mutable.
Their size is fixed.
To learn more about the array data structure, check out Array tutorials.

Creating an array –

In Kotlin, arrays are not a native data type, but a mutable collection of similar items which are represented by the Array class.
There are **two** ways to define an array in Kotlin.

Using the arrayOf() function –

We can use the library function **arrayOf()** to create an array by passing the values of the elements to the function.

Syntax:

```
val num = arrayOf(1, 2, 3, 4)   //implicit type declaration
val num = arrayOf<Int>(1, 2, 3) //explicit type declaration
```

Kotlin program of creating array using arrayOf() and arrayOf<Int> functions-

Kotlin program of creating array using arrayOf() and arrayOf<Int> functions-

```
fun main()
{
    // declaring an array using arrayOf()
    val arrayname = arrayOf(1, 2, 3, 4, 5)
    for (i in 0..arrayname.size-1)
    {
        print(" "+arrayname[i])
    }
    println()
    // declaring an array using arrayOf<Int>
    val arrayname2 = arrayOf<Int>(10, 20, 30, 40, 50)
    for (i in 0..arrayname2.size-1)
    {
        print(" "+arrayname2[i])
    }
}
```

Output:

```
 1 2 3 4 5
 10 20 30 40 50
```

Using the Array constructor –

Since Array is a class in Kotlin, we can also use the Array constructor to create an array.
The constructor takes **two** parameters:
The size of the array, and
A function which accepts the index of a given element and returns the initial value of that element.

Syntax:

```
 val num = Array(3, {i-> i*1})
```

In the above example, we pass the size of the array as 3 and a lambda expression which initializes the element values from 0 to 9.

Kotlin program of creating array using constructor –

```
fun main()
{
    val arrayname = Array(5, { i -> i * 1 })
    for (i in 0..arrayname.size-1)
```

```
    {
      println(arrayname[i])
    }
}
```

Output:
0
1
2
3
4
Apart from these, Kotlin also has some **built-in** factory methods to create arrays of primitive data types, such as byteArray, intArray, shortArray, etc. These classes do not extend the Array class; however, they implement the same methods and properties.
For example, the factory method to create an integer array is:
val num = intArrayOf(1, 2, 3, 4)
Other factory methods available for creating arrays:
byteArrayOf()
charArrayOf()
shortArrayOf()
longArrayOf()
Accessing and modifying arrays –
So far, we have seen how to create and initialize an array in Kotlin. Now, let's see how to access and modify them.
Again, there are **two** ways of doing this:
Using get() and set() methods –
As you know, an array in Kotlin is basically a **class**. Therefore, we can access the data of a class object via its member functions. The get() and set() functions are said to be member functions.
The get() method takes a single parameter—the index of the element and returns the value of the item at that index.
Syntax:
val x = num.get(0)
The set() method takes 2 parameters: the index of the element and the value to be inserted.
Syntax:
num.set(1, 3)
The above code sets the value of the second element in the array to 3
Using the index operator [ ] –
The **[ ]** operator can be used to access and modify arrays.
To access an array element, the syntax would be:
val x = num[1]
This will assign the value of the second element in *num* to *x*.
To modify an array element, we should do:
num[2] = 5;
This will change the value of the third element in the num array to 5.
Note: Internally, the index operator or [ ] is actually an overloaded operator (see operator overloading) and only stands for calls to the get() and set() member functions.
Here is a working example of Kotlin array manipulation in which we create an array, modify its values, and access a particular element:

```
fun main()
{   // declare an array using arrayOf()
    val num = arrayOf(1, 2, 3, 4, 5)
```

```
    num.set(0, 10)  // set the first element equal to 10
    num.set(1, 6)   // set the secondelement equal to 6

    println(num.get(0)) // print the first element using get()
    println(num[1]) // print the second element using []
}
```

Output:
10
6

Traversing Arrays –

One important property of an array is that it can be traversed programmatically, and each element in the array can be manipulated individually. Kotlin supports few powerful ways to traverse array.
The simplest and most commonly used idiom when it comes to traversing an array is to use the for-loop.

Syntax:

```
for(i in num.indices){
    println(num[i])
  }
```

Kotlin program of array traversal using for loop-

```
// Traversing an array
fun main()
{
    val num = arrayOf<Int>(1, 2, 3, 4, 5)
    num.set(0, 10)
    num.set(1, 6)
    for (i in num.indices)
    {
        println(num[i])
    }
}
```

Output:
10
6
3
4
5

Alternatively, we can use the range to achieve the same effect. In Kotlin, a **range** is an interval between two values (start and end) and can be created using the **(..)** operator. Traversal through the range can be then done using the **in** keyword.

Syntax for range:

```
for (i in 0..10){
   println(i)
}
```

The range of elements in an array is defined from 0 to size-1. So, to traverse an array using range, we run a loop from 0 to size-1 on the array name.

Kotlin program of array traversal using range-

```
// Traversing an array
fun main()
{
    val arrayname = arrayOf<Int>(1, 2, 3, 4, 5)
```

```
    for (i in 0..arrayname.size-1)
    {
        println(arrayname[i])
    }
}
```

Output:
1
2
3
4
5
Another arguably less tedious, way to do the above is using the foreach loop.
Syntax:
arrayname.forEach({index->println(index)})
Kotlin program of array traversal using foreach loop-

```
// Traversing an array
fun main()
{
    val arrayname = arrayOf<Int>(1, 2, 3, 4, 5)
    arrayname.forEach({ index -> println(index) })
}
```

Output:
1
2
3
4
5

Arrays in Kotlin: Declaration, Initialization

An array is a collection of elements of the same type. In Kotlin, arrays are used to store multiple values of similar types.

Declaration and Initialization:

```
// Declaring an array of integers
val numbers: IntArray = intArrayOf(1, 2, 3, 4, 5)

// Declaring an array of strings
val names: Array<String> = arrayOf("Alice", "Bob", "Charlie")
```

Array Operations: Push, Pop

Arrays in Kotlin have a fixed size, so you can't directly push or pop elements like in some other languages. However, you can achieve similar functionality by using specialized functions or by creating a custom data structure.

```
// Adding an element to the end of the array
val newNumbers = numbers.plus(6)

// Removing the last element from the array
val shortenedNumbers = newNumbers.dropLast(1)
```

Practice Exercise

```
fun main() {
    val numbers = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    val shortenedNumbers = numbers.dropLast(1)
    println(shortenedNumbers.joinToString(", "))
}
// output
// 1. 2. 3. 4. 5. 6. 7. 8. 9
```

Day 2: Lists and Basic Algorithms

2. Introduction to Lists

In earlier codelabs, you learned about basic data types in Kotlin such as Int, Double, Boolean, and String. They allow you to store a certain type of value within a variable. But what if you want to store more than one value? That is where having a List data type is useful.

A list is a collection of items with a specific order. There are two types of lists in Kotlin:

- Read-only list: List cannot be modified after you create it.
- Mutable list: MutableList can be modified after you create it, meaning you can add, remove, or update its elements.

When using List or MutableList, you must specify the type of element that it can contain. For example, List<Int> holds a list of integers and List<String> holds a list of Strings. If you define a Car class in your program, you can have a List<Car> that holds a list of Car object instances.

The best way to understand lists is to try them out.

Create a List

1. Open the Kotlin Playground and delete the existing code provided.
2. Add an empty main() function. All of the following code steps will go inside this main() function.

```
fun main() {

}
```

3. Inside main(), create a variable called numbers of type List<Int> because this will contain a read-only list of integers. Create a new List using the Kotlin standard library function listOf(), and pass in the elements of the list as arguments separated by commas. listOf(1, 2, 3, 4, 5, 6) returns a read-only list of integers from 1 through 6.

```
val numbers: List<Int> = listOf(1, 2, 3, 4, 5, 6)
```

4. If the type of the variable can be guessed (or inferred) based on the value on the right hand side of the assignment operator (=), then you can omit the data type of the variable. Hence, you can shorten this line of code to the following:

```
val numbers = listOf(1, 2, 3, 4, 5, 6)
```

5. Use println() to print the numbers list.

```
println("List: $numbers")
```

Remember that putting $ in the string means that what follows is an expression that will be evaluated and added to this string (see string templates). This line of code could also be written as println("List: " + numbers).

6. Retrieve the size of a list using the numbers.size property, and print that out too.

```
println("Size: ${numbers.size}")
```

7. Run your program. The output is a list of all elements of the list and the size of the list. Notice the brackets [], indicating that this is a List. Inside the brackets are the elements of numbers, separated by commas. Also observe that the elements are in the same order as you created them.

```
List: [1, 2, 3, 4, 5, 6]

Size: 6
```

Access list elements

The functionality specific to lists is that you can access each element of a list by its index, which is an integer number that represents the position. This is a diagram of the numbers list we created, showing each element and its corresponding index.

The index is actually an offset from the first element. For example, when you say list[2] you are not asking for the second element of the list, but for the element that is 2 positions offset from the first element. Hence list[0] is the first element (zero offset), list[1] is the second element (offset of 1), list[2] is the third element (offset of 2), and so on.

Add the following code after the existing code in the main() function. Run the code after each step, so you can verify the output is what you expect.

1. Print the first element of the list at index 0. You could call the get() function with the desired index as numbers.get(0) or you can use shorthand syntax with square brackets around the index as numbers[0].

```
println("First element: ${numbers[0]}")
```

2. Next print the second element of the list at index 1.

```
println("Second element: ${numbers[1]}")
```

Valid index values ("indices") of a list go from 0 to the last index, which is the size of the list minus 1. That means for your numbers list, the indices run from 0 to 5.

3. Print the last element of the list, using numbers.size - 1 to calculate its index, which should be 5. Accessing the element at the 5th index should return 6 as the output.

```
println("Last index: ${numbers.size - 1}")
println("Last element: ${numbers[numbers.size - 1]}")
```

4. Kotlin also supports first() and last() operations on a list. Try calling numbers.first() and numbers.last() and see the output.

```
println("First: ${numbers.first()}")
println("Last: ${numbers.last()}")
```

You'll notice that numbers.first() returns the first element of the list and numbers.last() returns the last element of the list.

5. Another useful list operation is the contains() method to find out if a given element is in the list. For example, if you have a list of employee names in a company, you can use the contains() method to find out if a given name is present in the list.

On your numbers list, call the contains() method with one of the integers that is present in the list. numbers.contains(4) would return the value true. Then call the contains() method with an integer that isn't in your list. numbers.contains(7) would return false.

```
println("Contains 4? ${numbers.contains(4)}")
println("Contains 7? ${numbers.contains(7)}")
```

6. Your completed code should look like this. The comments are optional.

```
fun main() {
    val numbers = listOf(1, 2, 3, 4, 5, 6)
    println("List: $numbers")
    println("Size: ${numbers.size}")

    // Access elements of the list
    println("First element: ${numbers[0]}")
    println("Second element: ${numbers[1]}")
    println("Last index: ${numbers.size - 1}")
    println("Last element: ${numbers[numbers.size - 1]}")
    println("First: ${numbers.first()}")
    println("Last: ${numbers.last()}")

    // Use the contains() method
    println("Contains 4? ${numbers.contains(4)}")
    println("Contains 7? ${numbers.contains(7)}")
}
```

7. Run your code. This is the output.

List: [1, 2, 3, 4, 5, 6]

Size: 6

First element: 1

Second element: 2

Last index: 5

Last element: 6

First: 1

Last: 6

Contains 4? true

Contains 7? false

Lists are read-only

1. Delete the code within the Kotlin Playground and replace with the following code. The colors list is initialized to a list of 3 colors represented as Strings.

```
fun main() {
    val colors = listOf("green", "orange", "blue")
}
```

2. Remember that you cannot add or change elements in a read-only List. See what happens if you try to add an item to the list or try to modify an element of the list by setting it equal to a new value.

```
colors.add("purple")
colors[0] = "yellow"
```

3. Run your code and you get several error messages. In essence, the errors say that the add() method does not exist for List, and that you are not allowed to change the value of an element.

```
    colors.add("purple")
    colors[0] = "yellow"
}
```

```
Unresolved reference: add
Unresolved reference. None of the following candidates is applicable
because of receiver type mismatch: public inline operator fun
kotlin.text.StringBuilder /* = java.lang.StringBuilder */.set(index: Int,
value: Char): Unit defined in kotlin.text
No set method providing array access
```

4. Remove the incorrect code.

You've seen firsthand that it's not possible to change a read-only list. However, there are a number of operations on lists that don't change the list, but will return a new list. Two of those are reversed() and sorted(). The reversed() function returns a new list where the elements are in the reverse order, and sorted() returns a new list where the elements are sorted in ascending order.

1. Add code to reverse the colors list. Print the output. This is a new list that contains the elements of colors in reverse order.

2. Add a second line of code to print the original list, so you can see that the original list has not changed.

```
println("Reversed list: ${colors.reversed()}")
println("List: $colors")
```

3. This is the output of the two lists printed.

```
Reversed list: [blue, orange, green]
List: [green, orange, blue]
```

4. Add code to return a sorted version of a List using the sorted() function.

```
println("Sorted list: ${colors.sorted()}")
```

The output is a new list of colors sorted in alphabetical order. Cool!

```
Sorted list: [blue, green, orange]
```

5. You can also try the sorted() function on a list of unsorted numbers.

```
val oddNumbers = listOf(5, 3, 7, 1)
println("List: $oddNumbers")
println("Sorted list: ${oddNumbers.sorted()}")
```

```
List: [5, 3, 7, 1]
Sorted list: [1, 3, 5, 7]
```

Note: Don't worry about having to remember all of the possible list operations. When you are developing apps in Android Studio, as you work with lists and other data types, Android Studio will show you the functions and properties available for them, as for example in the following screenshot.

```
list.
(m) [](index: Int)                                              Int
(m) get(index: Int)                                             Int
(m) indexOf(element: Int)                                       Int
(m) lastIndexOf(element: Int)                                   Int
(m) listIterator()                                 ListIterator<Int>
(v) lastIndex for List<T> in kotlin.collections                 Int
(f) last() for List<T> in kotlin.collections                    Int
(f) last {...} (predicate: (Int) -> Boolean) for List<T> in k…  Int
(m) listIterator(index: Int)                       ListIterator<Int>
(m) subList(fromIndex: Int, toIndex: Int)                List<Int>
(v) size                                                        Int
Press ⏎ to insert, ⇥ to replace
```

By now, you can see the usefulness of being able to create lists. However it would be nice to be able to modify the list after creation, so let's look at mutable lists next.

Linked List Operations: Traverse, Insert and Delete

There are various linked list operations that allow us to perform different actions on linked lists. For example, the insertion operation adds a new element to the linked list.

Here's a list of basic linked list operations that we will cover in this article.

- Traversal - access each element of the linked list
- Insertion - adds a new element to the linked list
- Deletion - removes the existing elements
- Search - find a node in the linked list
- Sort - sort the nodes of the linked list

Before you learn about linked list operations in detail, make sure to know about Linked List first.

Things to Remember about Linked List

- head points to the first node of the linked list
- next pointer of the last node is NULL, so if the next current node is NULL, we have reached the end of the linked list.

In all of the examples, we will assume that the linked list has three nodes 1 --->2 --->3 with node structure as below:

```
struct node {
  int data;
  struct node *next;
};
```

Traverse a Linked List

Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents.

When temp is NULL, we know that we have reached the end of the linked list so we get out of the while loop.

```
struct node *temp = head;
printf("\n\nList elements are - \n");
while(temp != NULL) {
  printf("%d --->",temp->data);
  temp = temp->next;
}
```

The output of this program will be:

List elements are -

1 --->2 --->3 --->

Insert Elements to a Linked List

You can add elements to either the beginning, middle or end of the linked list.

1. Insert at the beginning

- Allocate memory for new node
- Store data
- Change next of new node to point to head
- Change head to point to recently created node

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = head;
head = newNode;
```

2. Insert at the End

- Allocate memory for new node
- Store data
- Traverse to last node
- Change next of last node to recently created node

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = NULL;

struct node *temp = head;
while(temp->next != NULL){
  temp = temp->next;
}

temp->next = newNode;
```

3. Insert at the Middle

- Allocate memory and store data for new node
- Traverse to node just before the required position of new node
- Change next pointers to include new node in between

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;

struct node *temp = head;

for(int i=2; i < position; i++) {
  if(temp->next != NULL) {
    temp = temp->next;
  }
}
newNode->next = temp->next;
temp->next = newNode;
```

Delete from a Linked List

You can delete either from the beginning, end or from a particular position.

1. Delete from beginning

- Point head to the second node

```
head = head->next;
```

2. Delete from end

- Traverse to second last element
- Change its next pointer to null

```
struct node* temp = head;
while(temp->next->next!=NULL){
  temp = temp->next;
}
temp->next = NULL;
```

3. Delete from middle

- Traverse to element before the element to be deleted
- Change next pointers to exclude the node from the chain

```
for(int i=2; i< position; i++) {
  if(temp->next!=NULL) {
    temp = temp->next;
  }
}

temp->next = temp->next->next;
```

Search an Element on a Linked List

You can search an element on a linked list using a loop using the following steps. We are finding item on a linked list.

- Make head as the current node.
- Run a loop until the current node is NULL because the last element points to NULL.
- In each iteration, check if the key of the node is equal to item. If it the key matches the item, return true otherwise return false.

```
// Search a node
bool searchNode(struct Node** head_ref, int key) {
  struct Node* current = *head_ref;

  while (current != NULL) {
    if (current->data == key) return true;
      current = current->next;
  }
  return false;
}
```

Sort Elements of a Linked List

We will use a simple sorting algorithm, Bubble Sort, to sort the elements of a linked list in ascending order below.

1. Make the head as the current node and create another node index for later use.
2. If head is null, return.
3. Else, run a loop till the last node (i.e. NULL).
4. In each iteration, follow the following step 5-6.
5. Store the next node of current in index.
6. Check if the data of the current node is greater than the next node. If it is greater, swap current and index.

Check the article on bubble sort for better understanding of its working.

```
// Sort the linked list
void sortLinkedList(struct Node** head_ref) {
 struct Node *current = *head_ref, *index = NULL;
 int temp;

 if (head_ref == NULL) {
  return;
 } else {
  while (current != NULL) {
   // index points to the node next to current
   index = current->next;

       while (index != NULL) {
    if (current->data > index->data) {
     temp = current->data;
     current->data = index->data;
     index->data = temp;
        }
        index = index->next;
       }
       current = current->next;
  }
 }
```

```
}
```

LinkedList Operations in Python, Java, C, and C++

Python

Java

C

C++

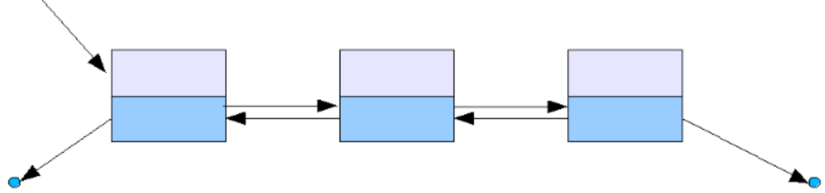
```python
# Linked list operations in Python



# Create a node
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None



class LinkedList:

    def __init__(self):
        self.head = None

    # Insert at the beginning
    def insertAtBeginning(self, new_data):
        new_node = Node(new_data)

        new_node.next = self.head
        self.head = new_node

    # Insert after a node
    def insertAfter(self, prev_node, new_data):
```

```
    if prev_node is None:
        print("The given previous node must inLinkedList.")
        return
```

What is a LinkedList?

Kotlin does not provide a LinkedList implementation. However, you can still use LinkedList because of the Kotlin JVM interoperability with Java. This is because LinkedList is a data collection that is part of the JVM implementation. So, let's understand what a LinkedList is.

In Java, a LinkedList provides a nonsynchronized data collection structure that is based on pointers. A LinkedList does not use an array as backing storage. This means that all its elements are not stored in a contiguous location. Specifically, a LinkedList is a doubly linked list whose elements are linked with a pair of pointers as below:



Doubly linked list is a LinkedList stores its elements in nodes. Each node has a pointer to the previous node and one to the next node in the list. To add an element to the list, the element is placed into a new node, which is then linked to the two adjacent elements in the list. You can declare the type of node elements through a generic.

The LinkedList class has the same methods as the ArrayList class because they both implement the List interface. However, the LinkedList class also implements the Deque interface, which provides the following methods:

- getFirst(): returns the element at the beginning of the list
- getLast(): returns the element at the end of the list
- addFirst(): adds an element at the beginning of the list
- addLast(): adds an element at the end of the list
- removeFirst(): removes the element at the beginning of the list
- removeLast(): removes the element at the end of the list

Why Kotlin does not provide a LinkedList implementation

It seems that Kotlin developers decided not to implement LinkedList because the Java implementation of LinkedList is suboptimal in nearly all cases. Also, considering that originally there was only Kotlin JVM, the Kotlin developers preferred to rely on the Java implementation.

Plus, as you are about to learn, LinkedList gets outperformed by ArrayList in nearly every situation. So, they may have thought there is no real need for a Kotlin implementation of LinkedList.

When to use an ArrayList over a LinkedList

ArrayList is the more efficient solution when it comes to random read access. This is because you can grab any element in constant time. That happens since there is an array behind the scenes. So, the get() method simply has to return the element specified by the index passed as a parameter. On the contrary, a LinkedList only provides sequential access, which is more expensive in terms of performance.

Plus, the ArrayList data structure does not involve overhead. On the contrary, a LinkedList defines two pointers for each element. Therefore, a LinkedList will occupy more elements than an ArrayList. This may easily become a problem when dealing with large lists.

Also, considering that an ArrayList is based on an array, the elements are stored sequentially in memory. So, ArrayLists can also take advantage of the principle of locality. This makes ArrayList more cache-friendly than LinkedList, whose elements are spread all over the RAM.

Thus, with ArrayLists you can experience an additional performance boost when nearby elements are accessed in a short time.

In detail, for an ArrayList<E>:

- get(index : Int) is O(1)
- add(element : E) is O(1) amortized, but O(n) when the underlying array has to be resized
- add(index: Int, element : E) is O(n)
- remove(index : Int) is O(n)
- remove() on the corresponding MutableListIterator object is O(n)
- add(element : E) on the corresponding MutableListIterator object is O(n)

Note the O(1) complexity on the get() method. This is the main benefit of ArrayList over LinkedList. Also, do not forget that you can get the MutableListIterator object from an ArrayList by calling the listIterator() method.

When to use a LinkedList over an ArrayList

LinkedList allows constant-time insertions or removals when inserting an element at the beginning or end of the list. This is true also when the current node is known. In other words, when using a LinkedList with an iterator, you can insert an element in O(1). This is the main benefit of LinkedList over an ArrayList.

Also, when using a LinkedList, you can insert elements indefinitely. On the other hand, the array used by an ArrayList will eventually need to be resized. This is an expensive operation that can be avoided in LinkedList. Plus, considering that this operation may not be executed every time a write operation is performed over the ArrayList, the underlying array may become wastefully empty. In this particular scenario, an ArrayList may have more allocated memory than a LinkedList.

In all other scenarios, LinkedList is worse or equal to ArrayList. Specifically, for a LinkedList<E>

Lists in Kotlin: Mutable List vs. List

In Kotlin, a List is an ordered collection of elements. The key difference between a List and a MutableList is that a MutableList allows for modification of its elements (addition, removal), while a List is immutable and its elements cannot be changed after creation.

Declaration:

```
val immutableList: List<Int> = listOf(1, 2, 3, 4, 5)
val mutableList: MutableList<String> = mutableListOf("apple",
"banana", "cherry")
```

List Operations: Insertion, Deletion, Searching

Insertion

```
// For MutableList
mutableList.add("date") // Adds "date" at the end of the list
mutableList.add(1, "grape") // Adds "grape" at index 1

// For List (Note: It's immutable, so it doesn't support direct
modification)
// To create a new list with an added element:
val newList = immutableList + 6
```

Deletion:

```
// For MutableList
mutableList.remove("banana") // Removes "banana"
mutableList.removeAt(0) // Removes element at index 0

// For List (Again, it's immutable, so create a new list without the
element)
val newList = immutableList.filter { it != 3 }
```

Searching

```
// For MutableList
val index = mutableList.indexOf("cherry") // Finds the index of
"cherry"

// For List
val contains = immutableList.contains(4) // Checks if the list
contains 4
```

Linked Lists in Kotlin

A linked list is a data structure where each element, known as a node, contains both the data and a reference (or link) to the next node. In Kotlin, linked lists are typically implemented using classes.

```
class Node<T>(val data: T) {
    var next: Node<T>? = null
}

class LinkedList<T> {
    var head: Node<T>? = null

    fun add(data: T) {
        val newNode = Node(data)
        newNode.next = head
        head = newNode
    }

    fun print() {
        var temp = head
        while (temp != null) {
            print("${temp.data} -> ")
            temp = temp.next
        }
        println("null")
    }
}

fun main() {
    val linkedList = LinkedList<Int>()
    linkedList.add(3)
    linkedList.add(5)
    linkedList.add(7)
    linkedList.print() // Outputs: 7 -> 5 -> 3 -> null
}
```

Day 3: Maps (Dictionaries)

Maps (HashMap's) in Kotlin: Declaration, Insertion, Retrieval

In Kotlin, maps are collections that store key-value pairs. Each key is associated with a specific value. Maps are often referred to as dictionaries in other programming languages.

Kotlin HashMap is a collection which contains pairs of object. Kotlin Hash Table based implementation of the MutableMap interface. It stores the data in the form of key and value pair. Map keys are unique and the map holds only one value for each key. It is represented as *HashMap<key, value>* or *HashMap<K, V>*. The hash table based implementation of HashMap does not guarantees about the order of specified data of **key**, **value** and **entries** of collections.

Declaration and Initialization:

```
// Immutable Map
val ages = mapOf("Alice" to 30, "Bob" to 25, "Charlie" to 35)

// Mutable Map
val scores = mutableMapOf("Alice" to 85, "Bob" to 90, "Charlie" to
75)
```

Insertion and Modification:

```
// For Mutable Map
scores["David"] = 80 // Adds a new entry
scores["Alice"] = 88 // Modifies the value associated with the
key "Alice"
```

Constructors of Kotlin HashMap class –
Kotlin HashMap provides 4 constructors and access modifier of each is public:
HashMap() : It is the default constructor which constructs an empty HashMap instance.
HashMap(initialCapacity: Int, loadFactor: Float = 0f) : It is used to construct a HashMap of specified capacity. If initialCapacity and loadFactor isn't being used then both will get ignored.
HashMap(initialCapacity: Int) : It is used to construct a HashMap of specified capacity. If initialCapacity isn't being used then it will get ignored.
HashMap(original: Map <out K, V>) : It creates instance of HashMap with same mappings as specified map.
Use of HashMap Functions –
Kotlin program of using functions HashMap(), HashMap(original: Map), Traversing hashmap, HashMap.get() –
java

```
fun main(args: Array<String>) {
    //A simple example of HashMap class define
    // with empty "HashMap of <String, Int>"
    var hashMap : HashMap<String, Int>
            = HashMap<String, Int> ()

    //printing the Empty hashMap
    printHashMap(hashMap)
```

```
    //adding elements to the hashMap using
    // put() function
    hashMap.put("IronMan" , 3000)
    hashMap.put("Thor" , 100)
    hashMap.put("SpiderMan" , 1100)
    hashMap.put("NickFury" , 1200)
    hashMap.put("HawkEye" , 1300)

    //printing the non-Empty hashMap
    printHashMap(hashMap)
    //using the overloaded print function of
    //Kotlin language to get the same results
    println("hashMap : " + hashMap + "\n")

    //hashMap traversal using a for loop
    for(key in hashMap.keys){
        println("Element at key $key : ${hashMap[key]}")
    }

    //creating another hashMap object with
    // the previous version of hashMap object
    var secondHashMap : HashMap<String, Int>
            = HashMap<String, Int> (hashMap)

    println("\n" + "Second HashMap : ")
    for(key in secondHashMap.keys){
        //using hashMap.get() function to fetch the values
        println("Element at key $key : ${hashMap.get(key)}")
    }

    //this will clear the whole map and make it empty
    println("hashMap.clear()")
    hashMap.clear()

    println("After Clearing : " + hashMap)


}

//function to print the hashMap
fun printHashMap(hashMap : HashMap<String, Int>){
    // isEmpty() function to check whether
    // the hashMap is empty or not
    if(hashMap.isEmpty()){
        println("hashMap is empty")
    }else{
        println("hashMap : " + hashMap)
    }
}
```

Output :
hashMap is empty : {}

hashMap : {Thor=100, HawkEye=1300, NickFury=1200, IronMan=3000, SpiderMan=1100}

hashMap : {Thor=100, HawkEye=1300, NickFury=1200, IronMan=3000, SpiderMan=1100}

Element at key Thor : 100
Element at key HawkEye : 1300
Element at key NickFury : 1200
Element at key IronMan : 3000
Element at key SpiderMan : 1100

secondHashMap :
Element at key Thor : 100
Element at key HawkEye : 1300
Element at key IronMan : 3000
Element at key NickFury : 1200
Element at key SpiderMan : 1100

hashMap.clear()
After Clearing : {}

Kotlin program of using HashMap initial capacity, HashMap.size

```
fun main(args: Array<String>) {
    //HashMap can also be initialize
    // with its initial capacity.
    //The capacity can be changed by
    // adding and replacing its element.
    var hashMap : HashMap<String, Int>
            = HashMap<String, Int> (4)

    //adding elements to the hashMap using put() function
    hashMap.put("IronMan" , 3000)
    hashMap.put("Thor" , 100)
    hashMap.put("SpiderMan" , 1100)
    hashMap.put("NickFury" , 1200)

    for(key in hashMap.keys) {
        println("Element at key $key : ${hashMap[key]}")
    }
    //returns the size of hashMap
    println("\n" + "hashMap.size : " + hashMap.size )

    //adding a new element in the hashMap
    hashMap["BlackWidow"] = 1000;
    println("hashMap.size : " + hashMap.size + "\n")

    for(key in hashMap.keys) {
        println("Element at key $key : ${hashMap[key]}")
    }
}
```

Output:
Element at key Thor : 100
Element at key IronMan : 3000
Element at key NickFury : 1200
Element at key SpiderMan : 1100

hashMap.size : 4

hashMap.size : 5

Element at key Thor : 100
Element at key BlackWidow : 1000
Element at key IronMan : 3000
Element at key NickFury : 1200
Element at key SpiderMan : 1100

Kotlin program of using functions HashMap.get(key), HashMap.replace(), HashMap.put() – java

```
fun main(args: Array<String>) {
    var hashMap : HashMap<String, Int>
            = HashMap<String, Int> ()

    //adding elements to the hashMap
    // using put() function
    hashMap.put("IronMan" , 3000)
    hashMap.put("Thor" , 100)
    hashMap.put("SpiderMan" , 1100)
    hashMap.put("Cap" , 1200)

    for(key in hashMap.keys) {
        println("Element at key $key : ${hashMap[key]}")
    }

    //the hashMap's elements can be accessed like this
    println("\nhashMap[\"IronMan\"] : "
            + hashMap["IronMan"])
    hashMap["Thor"] = 2000
    println("hashMap.get(\"Thor\") : "
            + hashMap.get("Thor") + "\n")

    //replacing some values
    hashMap.replace("Cap" , 999);
    hashMap.put("Thor" , 2000);

    println("hashMap.replace(\"Cap\" , 999)" +
            " hashMap.replace(\"Thor\" , 2000)) :")

    for(key in hashMap.keys) {
        println("Element at key $key : ${hashMap[key]}")
    }
}
```

Output:
Element at key Thor : 100
Element at key Cap : 1200
Element at key IronMan : 3000
Element at key SpiderMan : 1100

hashMap["IronMan"] : 3000
hashMap.get("Thor") : 2000

hashMap.replace("Cap", 999) hashMap.replace("Thor", 2000)) :
Element at key Thor : 2000

Element at key Cap : 999
Element at key IronMan : 3000
Element at key SpiderMan : 1100
Time complexity of HashMap –
Kotlin HashMap provides constant time or O(1) complexity for basic operations like get and put, if hash function is properly written and it disperses the elements properly. In case of searching in the HashMap containsKey() is just a get() that throws away the retrieved value, it's O(1) (assuming the hash function works properly).
Some other functions of Kotlin HashMap class –
Boolean containsKey(key: K): It returns true if map contains specifies key.
Boolean containsValue(value: V): It returns true if map maps one of more keys to specified value.
void clear(): It removes all elements from map.
remove(key: K): It removes the specified key and its corresponding value from map
In Kotlin, a HashMap is a collection that stores key-value pairs. Each key in a HashMap must be unique, but values can be duplicated. The HashMap class provides methods for adding, removing, and retrieving elements, as well as checking if a key or value is present in the map.
Here is an example of using HashMap in Kotlin:
Kotlin

```
fun main() {
    // create a new HashMap
    val myMap = hashMapOf<String, Int>()

    // add elements to the HashMap
    myMap.put("apple", 1)
    myMap.put("banana", 2)
    myMap.put("orange", 3)

    // print the HashMap
    println(myMap)

    // remove an element from the HashMap
    myMap.remove("banana")

    // print the updated HashMap
    println(myMap)

    // check if a key is present in the HashMap
    val containsKey = myMap.containsKey("apple")
    println("Contains key 'apple': $containsKey")

    // check if a value is present in the HashMap
    val containsValue = myMap.containsValue(3)
    println("Contains value 3: $containsValue")
}
```

Output:
{apple=1, banana=2, orange=3}
{apple=1, orange=3}
Contains key 'apple': true
Contains value 3: true

In this example, we create a new HashMap with String keys and Int values. We then add three key-value pairs to the HashMap using the put() method. Next, we print the entire HashMap using the println() function. We then remove the key-value pair with the key "banana" using the remove()

method and print the updated HashMap. Finally, we use the containsKey() and containsValue() methods to check if the HashMap contains the key “apple” and the value 3, respectively.
Advantages of HashMap:
HashMap provides a flexible way to store key-value pairs and is easy to use.
HashMap provides efficient O(1) time complexity for basic operations such as adding, removing, and retrieving elements.
HashMap can be used to store a wide variety of data types, including user-defined objects.
Disadvantages of HashMap:
HashMap uses more memory than some other data structures because it stores both keys and values.
HashMap is not thread-safe by default, so concurrent access to a HashMap can cause data corruption or unexpected behavior. If you need to access a HashMap from multiple threads, you should use a thread-safe implementation or use synchronization to ensure thread safety.
The order of elements in a HashMap is not guaranteed, which can be a disadvantage in some cases where element ordering is important. If you need to maintain a specific order of elements, you should use a different data structure such as a LinkedHashMap.

What does sorting mean?

Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order. In general, the organization of different objects in order on the basis of size, color, shape, etc are referred to be sorted.
Example - Sorting of a table on the basis of score

Name	Score
Hannah	93
Edward	79
Miranda	85
William	64
Joanna	81
Collin	85
Mallory	81
Oscar	63
Arturo	79
Annie	72

Name	Score
Oscar	63
William	64
Annie	72
Edward	79
Arturo	79
Joanna	81
Mallory	81
Miranda	85
Collin	85
Hannah	93

Introduction to Sorting Algorithms (e.g., Selection Sort, Insertion Sort)
Selection Sort:
Finds the smallest element in the unsorted portion of the array and swaps it with the first element of the unsorted portion.
Insertion Sort:
Builds the sorted portion of the array one element at a time by repeatedly inserting the next element into its correct position.

Day 4: Complete Revision of Kotlin

Day 5: Practice Session & Working on Projects

Week 4- User Interface design for Mobile Apps

Day 1: Introduction to UI/UX Design

Principles of UI/UX Design

UI stands for User Interface and UX stands for User Experience. Together, they encompass the design process of creating products that are not only visually appealing but also functionally effective and user-friendly.

A common misconception is that UI and UX come down to the same thing and that they are just one discipline, but that is not entirely true. They are separate disciplines that focus on different aspects of the user's journey with a digital product.
However, they overlap in many ways and are so closely connected that they've largely merged into one profession. Let's take a look at the main differences between UX and UI before moving on to discuss what the UI/UX design process involves and why it matters to you.

What are user experience goals?
UX designers are responsible for making sure that the user's interaction with a digital product is as fast, easy and efficient as possible. They are not concerned with visuals; they map out the user journey and strategise to solve users' pain points and meet their needs. The outcome of that is a wireframe – a blueprint of the product.

What are user interface goals?
UI designers pick up where UX designers leave off. Their role is to bring the digital product to life based on the wireframe provided. Like in the furnishing example – you already have all the functionalities mapped out, and now it is time to decorate.
As for UI, the point is to make the interface aesthetically pleasant for users, but also to add to the efficiency and ease that the UX designer has planned out. Additionally, it's important that the product reflect the brand's image and vision properly. So, UI designers are responsible for creating an interface that is:

- True to the business.
- Hierarchical and logical.
- Easy to navigate.
- Nice to look at.
- Responsive.

UI/UX designer goals – the bottom line
Ultimately, the final product is meant to be pleasant-looking, functional and give users a fantastic experience in terms of performance and navigation. You can only have that if both the UX and UI design are up-to-par, and because they complement each other so much, today we are increasingly talking about a profession involving both skillsets.
If you're wondering what the UI/UX design process is like, let us take a brief look at the crucial stages and the focus of each.

What is the user experience/user interface design process?
There are a few stages that every UI/UX design process must include in order to result in a quality product, and they can be grouped in three. Let's have a look at what they are and what they involve.

What is the difference between UX and UI?

UI design relates to the appearance and feel of a digital product. It focuses on visual factors like buttons, fonts, colour schemes, images, interactive elements, etc.
So, what is user experience design, then? Well, it refers to the experience a user has interacting with your product. Its main focus is enabling seamless goal achievement for users.

You can look at the difference through the example of a car. The UX would be the mechanics of your car, like engine power, transmission type or fuel consumption. The UI would be the aesthetics, such as the livery, paint, rims, dashboard or seats. Or, you can imagine furnishing a home; think of the UX designer as the construction manager, and the UI designer as the interior designer.

Understanding the importance of user-centered design.

Understanding the importance of user-centered design is crucial for creating products and experiences that meet the needs and expectations of your target audience. Here are several key reasons why it's so essential:

Meets User Needs and Goals:

User-centered design starts by understanding the goals, needs, and pain points of your target users. This ensures that the product addresses real-world problems and provides value.

Enhances User Satisfaction:

When users find a product easy to use and it helps them accomplish their tasks efficiently, they are more likely to be satisfied and have a positive perception of the brand.

Reduces User Frustration:

A design that doesn't consider the user's perspective can lead to confusion, frustration, and ultimately, user abandonment. User-centered design helps minimize these negative experiences.

Increases User Engagement and Retention:

When a product is designed with the user in mind, it's more likely to capture and maintain user attention. Engaged users are more likely to stick around and continue using the product.

Improves Usability and Efficiency:

By focusing on user needs, you can streamline the user interface and make it more intuitive. This reduces the time and effort required for users to accomplish tasks.

Reduces Development Costs and Time:

User-centered design helps identify and address potential issues early in the design process. This can prevent costly redesigns and rework during the development phase.

Enhances Brand Loyalty:

A positive user experience builds trust and loyalty. Users are more likely to return and recommend a product that has consistently provided them with value and a positive experience.

Facilitates Innovation:

Understanding the user's context and needs often leads to innovative solutions. By empathizing with the user, you can discover new ways to address their problems.

Measurable Results:

With user-centered design, you can establish key performance indicators (KPIs) and metrics to evaluate the success of your design based on real user data and feedback.

Adapts to Changing User Needs:

As user preferences, behaviors, and technologies evolve, a user-centered approach allows you to adapt and evolve your product to continue meeting those changing needs.

Competitive Advantage:

In a crowded market, products that offer a superior user experience have a significant advantage. Users are more likely to choose products that are easy to use and meet their needs effectively.

Principles of UI/UX design.

1. User-Centered Design:

Focus on the needs, preferences, and behaviors of the end-users.

Conduct user research to understand their goals and pain points.

2. Usability:

Ensure that the product is easy to use and navigate.

Intuitive design minimizes the need for instructions.

3. Accessibility:

Make the product usable for people of all abilities and disabilities.

Consider factors like screen readers, color contrast, and keyboard navigation.

4. Consistency:

Maintain uniformity in design elements like colors, fonts, and layout throughout the product.

5. Feedback and Response:

Provide clear feedback for user actions to prevent confusion.
Indicate loading times and status updates.
6. Simplicity and Minimalism:
Keep the design simple to avoid overwhelming users.
Eliminate unnecessary elements or information.
7. Aesthetics:
Design should be visually pleasing and aligned with the brand's identity.
8. Hierarchy and Prioritization:
Guide the user's attention towards important elements using visual cues like size, color, and placement.
9. Navigation:
Ensure users can move through the product easily and find what they're looking for.
Use clear labels and intuitive menus.
10. Testing and Iteration:
Continuously test the design with real users to gather feedback and make improvements.
11. Emotional Design:
Consider the emotional impact of design elements on users.
Design choices can evoke feelings of trust, confidence, or delight.
12. Performance:
Optimize the product for speed and efficiency to enhance user experience.
13. Adaptability and Responsiveness:
Ensure the design works well on different devices and screen sizes.
14. Storytelling:
Use design elements to guide users through a narrative or flow.
15. Learnability:
Make it easy for users to learn how to use the product without extensive training.
16. Feedback Loops:
Allow users to provide feedback or report issues easily.

The main reasons UI/UX matters for your business are:

- User satisfaction
- User retention
- Greater potential for lead generation
- Stronger brand reputation
- Adaptability (to changes in user needs and trends)
- Consistency in brand image
- Consistency in user experience

70% of online businesses that fail do so because of bad usability, and 68% of users give up on a brand if they think it doesn't care about them.

User Centered Design (UCD) offers a fresh approach for product managers and designers who are determined to build only the best experiences for their users. Top brands genuinely live and breathe the voice of their customer and use insights to build products their users love.

In this guide, you will learn about the art and science of listening to your users and solving their problems in a meaningful way.

What is User Centered Design?

User Centered Design (sometimes called user-centric design) is a highly customer-centric approach to building and developing products. Sometimes referred to as human-centered design, it is more than a design process - it is both a philosophy and a framework.

It means putting the needs and desires of your customers at the center of everything you do to create ideal user experiences.

User-centric design requires a solid understanding of who your customers are and what they need. It might sound obvious, but teams often lack a deep, nuanced understanding of who their customers are and where product-market fit is strongest.

User centric design asks questions like, what problems are worth solving? What kind of solution could help? With user profiles and psychographics, it taps into users' psychology and emotional background to understand their needs and desires on a deeper level. At the same time, you need to drive business value and balance your company goals, priorities, and resources alongside users needs.

UX Designers and product managers who use the UCD methodologies know they need the right data to understand user needs. This typically involves extensive research, surveys, user testing, and tools such as UXCam to gain insight into user behavior.

Ultimately, user-centered design is an asset to any company. It enables you to build and ship products that provide significant and long-term value to the user - the kind of products people fall in love with and tell their friends about.

Why is User-Centered Design important?

User Centered Design is important because it leads to superior usability - and these days, user experience (ux) is a business advantage.

With user profiles and psychographics, it taps into users' psychology and emotional background to understand their needs and desires on a deeper level.

Your customers might not have heard of user-focused design, but they almost certainly prefer to buy products that are easy to use and navigate and solve a problem.

Users should not have to change their behaviors or expectations to fit your product. Understanding their needs and building a product around it can make the difference between a great user experience and a frustrating failure.

User-Centered Design offers many benefits, including:

- Avoid user frustration
- Streamline the development process
- Increase ROI
- Enhance competitiveness

In today's competitive business landscape, brands that don't put their customers first are at risk of falling behind. User-oriented design plays a key role in helping you achieve your business goals, such as increased sales, improved customer satisfaction, and higher customer retention.

User Centered Design Principles

Before jumping into the specific steps of a UCD process, a solid understanding of the User-Centered Design principles is vital. We have identified eight UCD principles that are the backbone of a successful design strategy. Without these, you will struggle to implement user-centered design successfully, and it may not even truly be user-centric design. So, let's dive in.

Product vision and impact

Above all else, you need an inspiring vision for your brand. This vision is the "north star" for your team and your product. What does your brand stand for? What problem do you solve for your users? Why should they come to you? These fundamental questions are a key part of your brand identity. This piece of the puzzle needs to be solved before you start building or even talking about features and benefits.

Understand your customers

To be successful, a business needs to align with the needs of its customers. People come to you because they need something. This is usually because you solve a problem they are facing or fulfill a desire they have.

Businesses should use qualitative and quantitative data to paint a full picture of user needs and behavior. These days, you have an abundance of options available at your fingertips. Popular research methods include surveys, user testing tools, and user analytics platforms such as UXCam.

If you're curious about how users are behaving on your app, a UXCam free plan includes 10,000 sessions / month, and a free trial with 100,000 sessions / month.

Choose the most significant opportunity

User needs are complex and near-infinite. Most design or development teams will tell you they have a long development backlog. Making their way through everything on the list of desired features or suggested design changes would be near impossible. That means you need to prioritize. But how can you decide what is a must-have and what is merely nice-to-have?

In User-Centered Design, prioritization is based on the expected impact. Pick a specific opportunity space or problem to solve where the need and business value are most significant, then synthesize the research into UX design challenges.

Measure success through outcomes

Ultimately, it's about impact, not output. Even today, many businesses are still stuck in the old way of measuring how many features they shipped and how many hours they spent on development time.

We aim to drive our business goals by serving our users in user-oriented design. The goal is not to build more features, but to positively influence user behavior by better meeting our customers' needs. We also move the needle on our business and financial KPIs when we do this.

Collaborate to innovate

A user-centered approach to product design is highly collaborative. It starts with ideation as part of a team. Together, you select design challenges to pursue through collaborative design thinking and framing.

Designers, product managers, and UCD experts can share ideas, analyze a feature or issue from various angles and suggest different solutions. A team benefits from the strengths of their peers and can learn from each other.

User Centered Design is not something you can effectively do on your own. It works best when teams come together to share their best ideas and deliver on them.

Create hypotheses

Acting on assumptions is risky. Let's say you think your users need X or don't like Y. You could be right - on the other hand, you might have misinterpreted their behavior or misunderstood their needs.

Before building anything or taking any tangible action, frame your assumption as a hypothesis. This hypothesis should connect to desired outcomes and solutions, which will help you to check relevant KPIs and measure the potential impact later.

Test and validate

Testing is not just a single step of the UDC process. It is an ongoing activity that happens before, during, and after you are working on a solution.

It is highly recommended to evaluate design decisions through usability testing and analytics platforms, such as UXCam, that provide data and insights around user behavior.

Prototyping and testing your ideas to evaluate your hypothesis will help you scrap bad ideas before you ship them, double down on key focus areas and give you and your team confidence that you are headed in the right direction. Testing also increases your credibility internally, as other teams and managers see you have a business justification for your design projects.

Iterate based on data

UCD is a highly data-driven and iterative design process. It is not enough to bring out a new release and hope for the best. When you release a new product or feature, track and measure the impact on user behavior and keep an eye on the associated business metrics.

World-class UX design teams and product managers are diligent about measuring the impact of what they have delivered, and they take action based on the data when needed.

For instance, you might release a slick new design of your User Interface (UI) and then realize users now take longer to accomplish a task, the completion rate for certain user flows has decreased, or users are using the search function more because they cannot find what they are looking for. Even if the data shows that your new design has enhanced the user experience, there is always room for further improvement.

In both cases, it is important to monitor user behavior, analyze the data and take action when

What is the User Centered Design (UCD) Process Like?

User Centered Design process is a cycle that flows smoothly to create first-class products with every release. Let's dive into the UCD process and how the individual steps come together to enable you to build and ship customer-centric products.

Generative Research

Gain a deep understanding of your users by empathizing with them. Interview them to learn about the context of their daily lives, goals, motivations, and problems, including the areas where your product might be involved. Analyze and synthesize into personas who represent a general portrait of your users.

Ideation

Ideation is a key part of the UCD process. In your initial brainstorming sessions, you will identify the personas who will use the product or feature, the purpose they will use it for and the circumstances under which they will use it. Afterward, you can use data to refine your user scenarios and flesh out use cases.

Outline business requirements

The next step is to specify the business requirements and user goals that must be met for the product to be successful.

Conceptualization and prototyping

Now, it is time to start creating real design solutions. This involves starting with a general design concept and increasing the level of detail in your prototype.

User testing

You have a design - now, it's time to put it to the test. Usability tests with real users are the optimal way to collect constructive feedback.

Iteration

Do you need to make changes based on what your users said? At this stage, you act on user feedback where necessary to build the best product possible.

Launch

Congratulations, you've built and tested your product or feature. You're ready to share it with the world and feel confident that your customers will love it.

How to measure the outcome of User-Centered Design

So, you and your team are living and breathing human centered design. You know your customers' needs inside and out, and you're fairly confident you're doing a great job. How do you know for sure if your new, customer-centric approach is working?

Any new feature is expected to create value - for the user and your business. Identify correct qualitative and quantitative signals before launching the feature so you know what to track and measure the outcome.

If you solve a genuine problem for your customers, your data will reflect this. So, what should you be looking for? You need to pay attention to two sets of metrics: **product metrics and business metrics.**

Product metrics refer to KPIs around usability, such as task success rate and time on task. In other words, is the feature helping users accomplish their goal, in a timely manner? Are they able to find what they are looking for using the navigation instead of relying on search?

Overall business metrics include the adoption rate of the feature and its financial impact, customer retention, and customer satisfaction.

Google Heart Framework

If you don't have specific business and product metrics in place yet, the HEART framework designed by Google's research team is a great place to start.

The HEART framework is a carefully-selected set of usability and business metrics that enables you to measure user experience on a larger scale. Its purpose is to guide your decision-making in the product development process.

There are five metrics used in the HEART framework:

- **Happiness:** user attitudes and level of satisfaction
- **Engagement:** how much a user interacts with a product
- **Adoption:** how many new users you have acquired over a specific time-frame
- **Retention:** how good you are at keeping your existing users
- **Task Success:** % of completions of a specific goal such as checkout or registration

What else do you need to know about User-Centered Design?

User Centered Design means taking into account the user's objectives, requirements, and feedback on the product. However, these are constantly evolving as society, and user habits change. This is why User-Centered Design is an ongoing cycle with multiple iterations.

You need to keep a close eye on user behavior and continuously adapt and optimize your product. It is not enough to launch a new feature and review the impact one time.

Of course, this is more effort than the "ship it and forget it" approach many businesses today have. We get it - there are always new projects, new goals, new management changes and too little time to get it all done. However, the business results are worth it.

How do you ensure accessibility and usability of mobile apps and websites?

Mobile apps and websites are essential tools for enterprise mobility, allowing you to work remotely and access data and services on the go. However, to ensure a positive user experience and avoid frustration, accessibility and usability are key factors to consider. In this article, you will learn how to design and test mobile apps and websites that are accessible and usable for a diverse range of users and devices.

Understand your users

The first step to ensure accessibility and usability is to understand who your users are, what their needs and preferences are, and how they interact with your mobile app or website. You can use various methods, such as user research, personas, scenarios, and user feedback, to gather and analyze user data and insights. This will help you to define your target audience, their goals and pain points, and the features and functions that are most important and relevant for them.

This is where invited experts will be adding contributions.

Experts are selected based on on their experience and skills.

Follow accessibility guidelines

The second step is to follow the accessibility guidelines and standards that apply to your mobile app or website, such as the Web Content Accessibility Guidelines (WCAG) and the Mobile Accessibility Best Practices. These guidelines provide recommendations and techniques to make your mobile app or website accessible to people with different abilities and disabilities, such as vision, hearing, motor, and cognitive impairments. For example, you should ensure that your mobile app or website has sufficient color contrast, clear and readable text, alternative text for images, captions for videos, keyboard and touch navigation, and compatibility with assistive technologies.

This is where invited experts will be adding contributions.

Experts are selected based on on their experience and skills.

Apply usability principles

The third step is to apply the usability principles and best practices that enhance the user experience and satisfaction of your mobile app or website. Usability refers to how easy and intuitive it is for users to accomplish their tasks and goals with your mobile app or website. Some of the usability principles and best practices are: simplicity, consistency, clarity, feedback, error prevention, and responsiveness. For example, you should ensure that your mobile app or website has a simple and logical layout, consistent and familiar design elements, clear and concise labels and instructions, timely and relevant feedback, error prevention and recovery mechanisms, and fast and reliable performance.

Optimize for mobile devices

The fourth step is to optimize your mobile app or website for different mobile devices, such as smartphones, tablets, and wearables. Mobile devices have various characteristics and constraints, such as screen size, resolution, orientation, input methods, battery life, and connectivity, that affect the user experience and functionality of your mobile app or website. Therefore, you should ensure that your mobile app or website is responsive, adaptive, and scalable to fit different screen sizes and orientations, supports different input methods, such as touch, gesture, voice, and keyboard, consumes minimal battery power and bandwidth, and handles network interruptions gracefully.

Test and evaluate

The fifth step is to test and evaluate your mobile app or website with real users and devices, using various methods and tools, such as usability testing, accessibility testing, user feedback, analytics, and emulation. Testing and evaluation will help you to identify and fix any issues or problems that affect the accessibility and usability of your mobile app or website, as well as to measure and improve the user experience and satisfaction. You should test and evaluate your mobile app or website throughout the design and development process, as well as after launching it, to ensure that it meets the user needs and expectations.

Keep learning and improving

The sixth and final step is to keep learning and improving your mobile app or website based on the user feedback, analytics, and trends. Accessibility and usability are not static or one-time goals, but rather ongoing and dynamic processes that require constant monitoring and updating. You should always listen to your users, track their behavior and preferences, and adapt to their changing needs and expectations. You should also keep up with the latest technologies, standards, and best practices that can enhance the accessibility and usability of your mobile app or website.

5 essential considerations for mobile design

The world is shrinking and it's only getting smaller. Computers used to fill a room but now fit into backpacks. We used to be tied to our homes by our phones now they are our travel companions, always there keeping us entertained.

Our search for information has transformed and now we live in a fast-paced world where access must be fast and immediate. If you're going to design for mobile, there are many things to consider and while many are understanding the target audience or standard UI/UX considerations; there are mobile-specific design considerations too to keep in mind, but don't forget that these don't replace the need for user research.

To get started, will you use responsive design or adaptive design and if you do, the most basic aspect is context. Think of the way in which the mobile device will be used and the specifics of the devices themselves.

If your users access the mobile web from their desks, great! But we all know, that more and more people are using their mobile devices to access the internet, and we can likely expect this trend to continue. That's why we rounded up 5 essential considerations for mobile design!

Essential considerations for mobile design

Screen size

Responsiveness is the name of the game here. We all know mobile devices have much smaller screens than desktops, and therefore designers should be aware of which elements are necessary to include in the design and which aren't. That being said, mobile devices have a very wide range of sizes. Now think about how many different phones and tablets there are on the market today. The designer must consider these variations in screen sizes in order to ensure the design is consistent regardless of the device being used.

Simple navigations

Simplicity goes along with the first consideration of screen size. This does not mean that now you have to stick with minimalistic designs with no creative juices, you just need to choose which features are worth sprucing up and which aren't. Since the screens on mobile devices are relatively small, it is important to keep the design simple.

So try to:

- Remember to make targets big enough so that they're easy for users to tap. (making navigation comfortable)
- Minimize the levels of navigation involved by offering short-key access to different features
- Ensure labelling is clear and concise for navigation
- Avoid excessive scrolling (create navigation that can lead the user to the specific points he needs to get in the quickest way possible)
- Make it easy to swap between the mobile and full site (if you choose to implement separate versions)

Consistency

It is essential to maintain an overall consistent appearance throughout mobile applications which will provide familiarity and ease of use for the end-user. Consistency is a fundamental principle of design that will eliminate confusion. Additionally, elements that further consistency can include but are not limited to style and colour, fonts and font sizes, and even buttons.

Consistency means the following:

- Visual consistency: Ensure all typefaces, buttons and labels are consistent across the entire app. (the arrangement of the icons and the text must form a whole without including empty spaces inside)
- Functional consistency: Improves usability and learnability by allowing users to leverage existing knowledge about how the design functions.
- External consistency: Design should be consistent across multiple products. This way, the user can apply prior knowledge when using another product.

Finger-friendly mobile design

It is important to consider how users interact with mobile. With mobile devices, we use our fingers to navigate through different hand gestures that have become standard use for tasks.

Gestures like;

- Tap: Touch the surface briefly
- Double tap: Touch the surface with two quick motions (often to zoom)
- Drag: Move along the surface without breaking contact
- Pinch/spread: Touch the surface with two fingers to move in (pinch) or out (spread)
- Press: Touch the surface and hold
- Flick: Scrolls quickly

When using mobile devices, users expect things to work a certain way, and it is important that designers consider these actions.

Button design is also an important consideration in this category. Because users will be navigating with their fingers, buttons need to be large enough to tap with ease. Similarly, important buttons like clear, save, delete or submit should be placed in a location on the screen that won't accidentally get tapped, whereas the commonly used buttons and call to action should be placed in areas easy to reach to encourage use.

Optimize the content for mobile

As we know, well-crafted content can do wonders for your application. Excellent typography is key for application success. It is the art of arranging text in such a way that it looks readable and accessible on all mobile screens. When the user lands on the screen, the content should be clear at first glance.

To ensure this, do the following:

- Use proper spacing between the lines (do not squeeze the lines just because you need everything to fit in)
- Try to keep font standard (don't use more than 2 typefaces)

- Use a text colour that looks aesthetically appealing to the user
- Standardize font size to fit any screen size (16px is a good place to start when choosing your default mobile font size.)
- To increase the readability of content, avoid all caps.
- Do not extend the length of the paragraphs, keep it short (30 to 40 characters per line)

How Mobile Screen Size, Resolution, and PPI Screen Affect Test Coverage

MOBILE APPLICATION TESTING

Mobile screen sizes are growing bigger. And it's more important than ever to understand key device characteristics from a development and testing standpoint. This includes:

- Mobile screen size.
- Screen resolution.
- Screen PPI (pixel per inch / pixel density).

These characteristics are important whether you are developing native apps, hybrid apps, or responsive web apps.

A Guide to Mobile Screen Size and Density

There are a variety of mobile screen sizes available. As of November 2015, tablet and mobile screen sizes range from 4.7" to 9.7".

Common mobile screen resolutions include: 480×800, 640×1136, 720×1280, 750×1334, 1080×1920, and 1440×2560. The most common resolution is 720x1280.

How Android Categorizes Mobile Screen Density

Android categorizes mobile screen density into one of six families:

- **LDPI** (low): ~120dpi
- **MDPI** (medium): ~160dpi
- **HDPI** (high): ~240dpi
- **XHDPI** (extra high): ~320 dpi
- **XXHDPI** (extra extra high): ~480 dpi
- **XXXHDPI** (extra extra extra high): ~640 dpi

A Comparison of Mobile Screen Sizes, Resolution, and PPI			
Device Name	**Mobile Screen Sizes**	**Screen Resolution**	**PPI**
iPhone 6	4.7”	750 x 1334	326 (XHDPI)
Samsung Galaxy Note 5	5.7"	1440 x 2560	515 (XXXHDPI)
iPhone 6 Plus	5.5"	1080 x 1920	401 (XHDPI)
iPad Air 2	9.7"	1536 x 2048	264 (HDPI)

Samsung Galaxy S6	5.1"	1440 x 2560	515 (XXXHDPI)
iPhone 6S	4.7"	750 x 1334	326 (XHDPI)
Blackberry Passport	4.5"	1440 x 1440	452 (XXHDPI)
iPhone 6S Plus	5.5"	1080 x 1920	401 (XHDPI)
Google Motorola Nexus 6	6.0"	1440 x 2560	515 (XXXHDPI)
HTC One M9	5.0"	1080 x 1920	440 (XXHDPI)

Source: (Perfecto's Device Coverage Index, Nov. 2015)

It's important that you don't mix devices and operating systems based on the location of the devices. And it's also important to understand that screen size and resolution can change visuals.

For example, two Samsung devices with the same screen size but different resolutions might show different visuals to the user. Or they will consume more CPU and battery to process the various app visuals on these devices.

As a starting point, it is easy to determine your device's PPI and classify it as one of the density families accordingly.

How to Calculate PPI Screen

1. Calculate the Dp Value

Here's the formula to calculate the Dp value:

(calculate the device screen resolution square root --> for iPhone 6 it would be 750 x 1334)

So, you'll get 1,530 as the Dp value.

2. Match the Screen Size to the Dp Value

Match the screen size to the above Dp value to get the PPI.

An iPhone 6 device, which has a 4.7'' screen size, will result in 326 PPI.

Why Do Mobile Screen Size & PPI Matter?

Mobile apps run on different devices equipped with very different hardware (systems on chip). Therefore, these devices have a varied amount of CPU as well as battery types.
This impacts performance.

So, loading a given app on an LG G4 (5.5'') with 515 PPI would be different from an iPhone 6S Plus (also 5.5") but with a 401 PPI. The CPU, battery usage, and performance (responsive time) would be different for each device.

Because there is such a large, fragmented market of devices, there's a lot for developers and testers to consider. You ought to be thinking of the end user experience. And that means app performance, visuals, usability, and robustness.

It will be critical to know your app's benchmark performance on a variety of devices. This is a good step toward having the right test coverage for your mobile app.

How to Test Mobile Screen Size With Perfecto

Testing screen size, resolution, and PPI among manufacturers should be your first step in assuring good user experience across platforms.

Another aspect of mobile screen sizes teams should incorporate in their testing is the advent of foldable phones. Teams will need to ensure their apps can correctly adjust and function with all sorts of devices with foldable screens. In 2022 alone, more than 14.2 million foldable smart phones were sold.

When you use a tool like Perfecto for mobile testing, it's easy to test for a variety of mobile screen sizes, resolutions, and PPI. You can leverage our mobile test lab to make sure that your app runs smoothly, regardless of device.

See for yourself how Perfecto can help you test screen size, resolution, and PPI. Get started with a free trial.

Common Screen Resolutions for Mobile in 2023

- 360×800 (11.01%)
- 390×844 (7.92%)
- 414×896 (5.55%)
- 393×873 (5.26)
- 412×915 (5%)

Common Screen Resolutions for Desktop in 2023

According to the Worldwide Screen Resolution Stats September 2023, the most common screen resolutions across mobile, desktop, and tablet are:

1920×1080 (22.18%)

1366×768 (14.04%)

1440×900 (6.41%)

1280×720 (5.45%)

1280×1024 (4.52%)

Navigation complements search for several reasons:

- sometimes users don't know what to search for, and need help figuring out the partitioning of the search space;
- coming up with a good query and typing it requires more mental effort and higher **interaction cost** than **recognizing** and tapping a navigation link (and, in fact, **users are notoriously bad at formulating good queries**);
- site search often works a lot more poorly than the search-engines users expect it to.

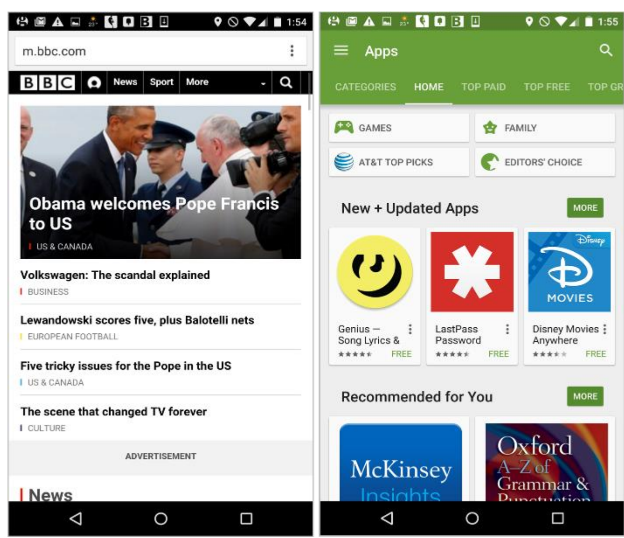
However, on mobile devices, both navigation and search come at a price: they occupy screen space and grab users' attention, both of which are at an even higher premium on mobile than on desktop. If the screen space is really scarce, a search box or navigation links at the top of the page can interfere with the users' ability to get to new information fast and may make the user work more. Pay attention to navigation and search, make them accessible and discoverable, but don't forget one

of the basic tenets of **mobile usability**: prioritize content over **chrome**. This is in fact one of the big challenges of implementing navigation on mobile: **how to prioritize content while making navigation accessible and discoverable.** Different approaches sacrifice either content prioritization or the accessibility of the navigation.

The Navigation Bar and the Tab Bar

Top Navigation Bar

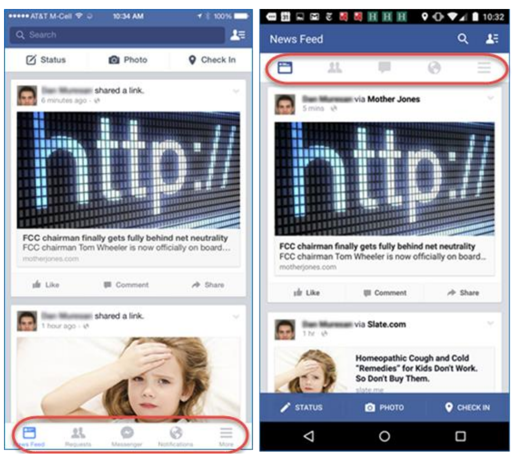
The top navigation bar is essentially inherited from desktop design. It is simply a bar that enumerates the main navigational options across the top of the screen. This is quite efficient, but has two disadvantages: (1) it works well only when there are relatively few navigation options; (2) it takes up valuable **real estate above the fold**.



The BBC's website (left) and Google Play for Android (right) both used a top navigation bar for the main navigation. Google Play was able to fit more items in the navigation bar by using a carousel.

The Tab Bar

The tab bar is a close relative of the top navigation bar specific to apps. It can appear at the top (Android mostly) or at the bottom of the page (iOS mostly). It is usually present on most pages within an app and has the same disadvantages as the navigation bar. One important difference between tab bars and navigation bars is that tab bars are **persistent,** that is, they are always visible on the screen, whether the user scrolls down the page or not. Navigation bars usually start out being present at the top of the page but disappear once the user has scrolled one or more screens down. (A **sticky version of the navigation** bar stays put at the top of the screen, or as the user starts scrolling up, reappears at the top of the page.)



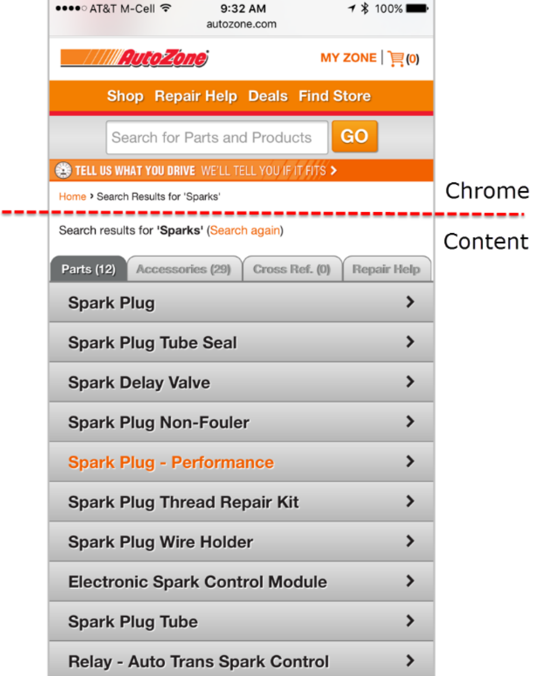
Facebook on iPhone (left) and Android (right) used a tab bar for the main navigation options. The tab was positioned in accordance with official operating-systems guidelines: at the bottom on iPhone and at the top of the page on Android. Note that the icons are labeled on the left screenshot: a recommended best practice in most cases.

Tab bars and navigation bars are well suited for sites with relatively few navigation options. If your site has more than 5 options, it's hard to fit them in a tab or navigation bar and still keep an optimum touch-target size. Solutions such as using a **carousel** navigation bar or tab bar, like in the Google Play example at the beginning of the article, may work but they are not always appropriate. Out of sight is out of mind, and if the categories are widely different (like the case of an older version of Weather Channel illustrated below), it's likely that users won't think to scroll to get to those options, simply because the weak information scent from the visible categories may prevent them from guessing what items are hidden.

An older version of Weather Channel for Android had a carousel tab bar at the bottom of the screen; the categories in the carousel were not similar or predictable, and thus the ones that were not visible had little discoverability.

If you decide to use a navigation bar or a tab bar, they should be the main chrome area of the screen and little extra space should be devoted to other utility-navigation options or to search. If the site has 4–5 main navigation options, it may make sense to have them all visible on the screen at all times, especially if these are options that will likely be needed. However, keep in mind that navigation needs to be judged in the context of the overall chrome on a page: even if a site may only have a few top-tier categories, if other utility-navigation links (e.g., shopping cart, account

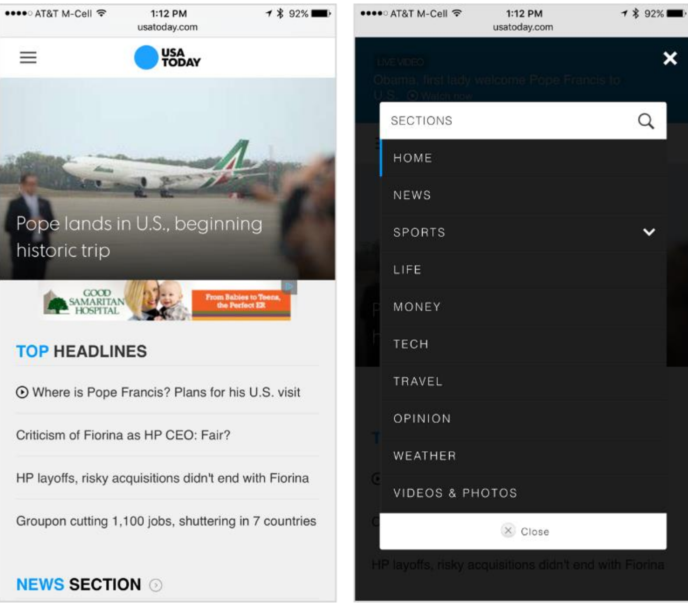
information) and search must also be included, the overall chrome may add up wasting too much space on the page.



*Autozone.com: Although the site only had 4 main-navigation categories (*Shop, Repair Help, Deals*, and* Find Store*), there were a lot more chrome elements present on the page (logo, shopping cart,* My Zone *link, search box) that all occupied too large an area.*

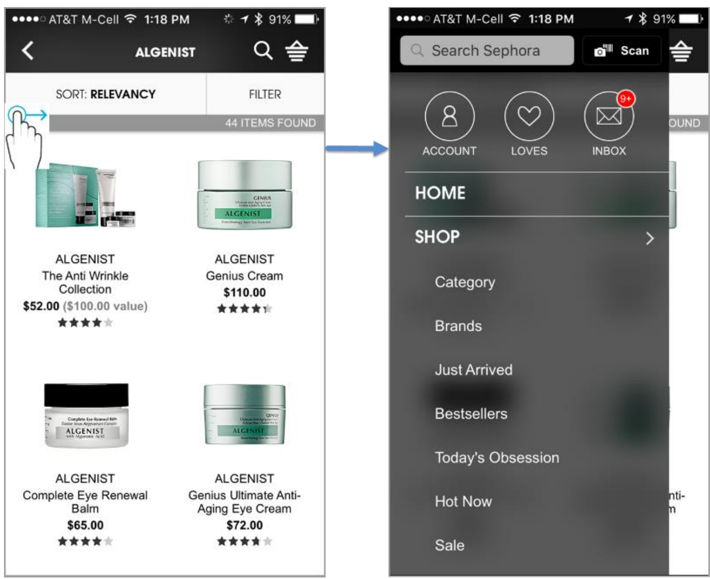
The Hamburger Menu (and Variants)

The navigation menu is a menu that contains the main navigation options in a manner that usually hides the detailed options but makes them visible upon request. While the hamburger icon is perhaps the most talked-about signifier for a navigation menu, other labels and/or **icons** can be used for navigation. (In fact, **third-party research** seems to suggest that using the word *Menu* instead of the hamburger icon is slightly more popular with users.) The main advantage of the navigation menu is that it can contain a fairly large number of navigation options in a tiny space and can also easily support submenus, if needed; the disadvantage is that it is less discoverable, since, as the old adage says, "out of sight is out of mind."



A hamburger menu was used for the global navigation options on USAToday.com.

A version of the navigation menu is when the menu has no signifier and is discovered through a gesture. In the Sephora app, on deep pages the menu can be accessed by swiping horizontally on the left edge (a gesture that is problematic in iOS, since, from iOS 7 on, Apple has started pushing the horizontal swipe as *Back*).



Sephora for iPhone: There was no visible menu button, but a horizontal swipe on the left edge exposed the menu. Most users would never discover this feature and would restrict themselves to using the visible options.

The navigation menu makes the navigation options least discoverable and is best suited for content-heavy, browse-mostly sites and apps.

If users rarely care about navigating to specific sections of the site and are mostly content to digest whatever information is presented to them (as is often the case on news sites), then a navigation menu is appropriate. The navigation menu also has the advantage of stealing a minimum amount of space from content, which is the star of browse-mostly sites.

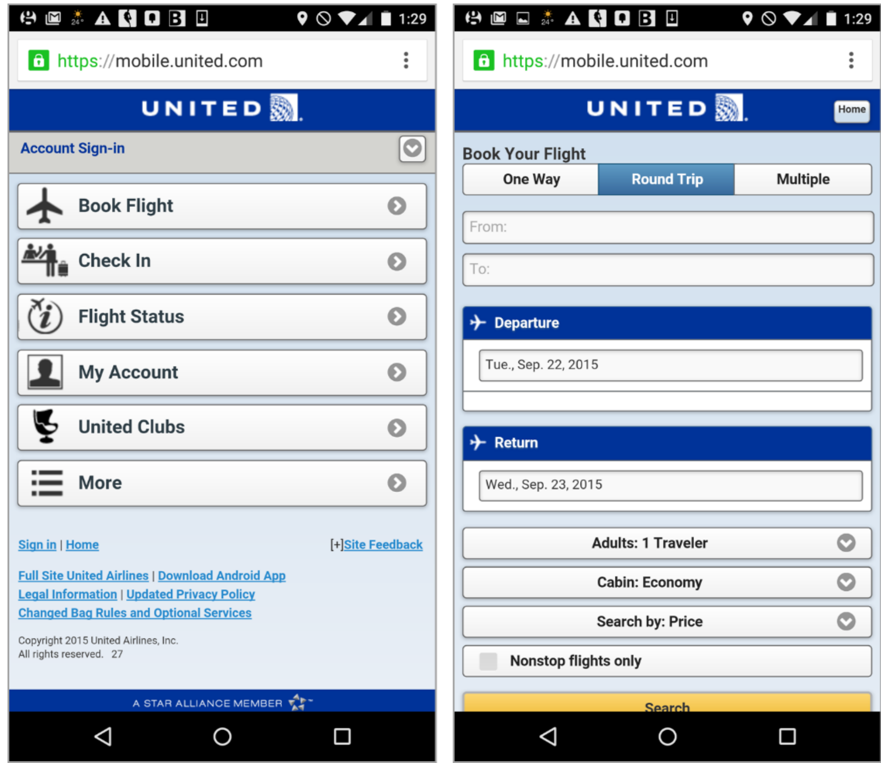
However, keep in mind that when navigation is hidden under a menu, even though that menu as a whole may be salient enough, users will have to make a decision to open it and check whether the individual navigation options are relevant. While the navigation menu is becoming standard and many mobile users are familiar with it, many people still simply don't think to open it. Even users who tried the navigation menu at some point during a session may not remember to do so later on.

If you decide to use a navigation menu, you need to think seriously about **supporting navigation in other ways**, such making the IA structure of your site more discoverable by increasing the cross-section links.

The Navigation Hub

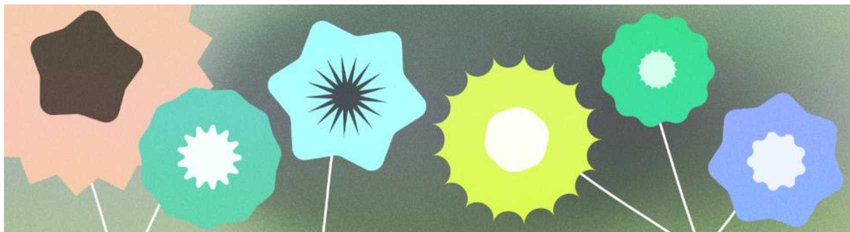
The navigation hub is a page (usually the homepage) that lists all the navigation options. To navigate to a new location, users have to first go back to the hub and then use one of the options listed there. This navigation approach usually devotes the homepage exclusively to navigation (at the expense of content), and incurs an extra step (back to the hub) for each use of the navigation. **It can work well in task-based websites and apps, especially when users tend to limit themselves to using only one branch of the navigation hierarchy during a single session.**

With the homepage as navigation hub, prime real estate will be wasted for chrome and all navigation will have to go through the homepage. While these two may seem as major disadvantages (and they are for most types of sites or apps), they can be less of a problem for those sites and apps used not for browsing and consuming content, but for accomplishing a very specific task (for example, checking in for a flight or changing the settings of the phone). Such sites and apps can take advantage of the homepage-as-navigation-hub pattern, especially if users rarely accomplish more than one task during a single session, and thus they don't need to traverse the navigation tree often (an action that is relatively difficult and annoying if all navigation must go through the homepage).



United used the homepage as a navigation hub. On deep pages, users had to use the on-page Home *button to go back to the homepage if they wanted to select a different navigation option.*

In the United example above, most likely you want to *either* buy a ticket *or* check in for a flight, but not do both in a single session. Thus, most users won't have to return to the hub in this example, meaning that it rather serves as an efficient distribution point.



Use color to express style and communicate meaning. Setting your app's colors can be crucial for personalization, defining semantic purpose, and of course defining brand identity.

- To ensure accessibility:
    - Check color contrast and avoid pairing colors with similar tones.
    - Consider that red and green are common patterns, but also that they're not accessible to users with certain kinds of color blindness.
- Practice using colors meaningfully: Apps can be vibrant and expressive, but stick to a palette of colors. Extending your scheme with too many semantic colors can be confusing, while having too many decorative colors can be overwhelming.

- Colors can have patterns, so repeat established color patterns. If using semantic colors in your app, use consistent colors.
- To allow your app to work well in different contexts, build a light and dark color scheme (and ideally contrast themes).
- Assign colors with tokens to indicate the element's color role, instead of using a hardcoded value.
- Colors can come from various dynamic and static sources, but avoid mixing too many within the same view.
- When using dynamic content color, try to avoid pulling colors from multiple content pieces.

Color space on Android

To properly understand how Android applies color to your UI, it's important to first recognize how it's displayed on a device.

How color is displayed on a device

Your app is displayed on a backlit screen, which uses digital color and adheres to certain models and rules that help our eyes perceive that color. Digital color is *additive color*, created by "adding" or mixing of different lights to create a full spectrum of color. The way humans perceive colors from one screen to another can vary greatly depending on a device's color calibration, screen type, settings, and the color space.

When designing an app, consider the colors used may not be identical due to these factors, not to mention the unique color perceptions of individual users.

About color spaces

A color space is an organization of colors that uses a color model. RGB is an additive color model that creates the spectrum of colors through red, green, and blue, while CMYK, which is used for printing, is *subtractive*. For this reason, interactive designers typically use RGB or similar models to choose colors.

Material 3 (M3) introduced HCT, a new color space that uses hue, chroma, and tone to define colors that are perceptually accurate compared to other models like HSL

Hue, chroma, and tone

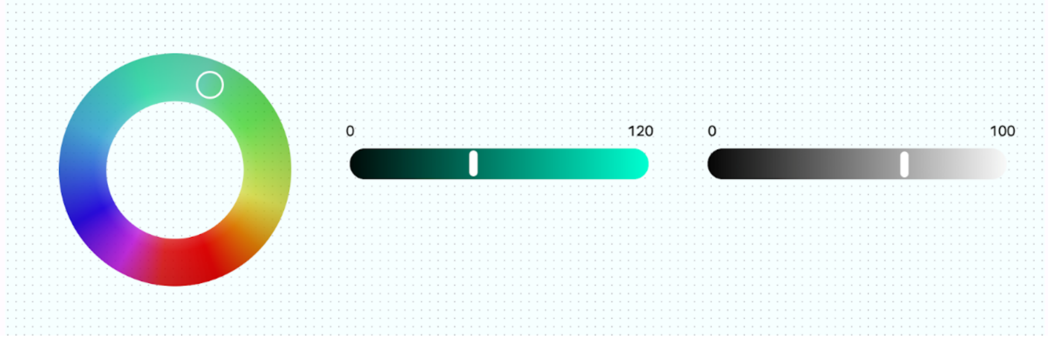


Figure 1: Hue, chroma, and tone visualized.

HCT allows for more personalized and flexible uses of color that stay within the system parameters. HCT models colors using hue, chroma, and tone:

- **Hue**: Hue is analogous to the adjective an individual user might use to describe the color, for example, "red" or "electric violet." The HCT value of hue ranges from 0–360.
- **Chroma**: Chroma represents the colorfulness of color, ranging from neutral gray to full vibrancy. In the HCT color space, chroma has a maximum value of around 120.
- **Tone**: Tone is the luminance, or brightness, of a color. HCT uses tone to create contrast. Colors set to the same tone value aren't accessible for certain accessibility contexts. Lower value tones are darker, and higher value tones are brighter.

Color system process

M3 color is built around the HCT model to derive harmonious accessible color schemes and helps dynamic color features. The M3 color system begins from a source color. This source color is translated into five key colors: primary, secondary, tertiary, neutral, and neutral variant. These five key colors create tonal palettes composed of tonal increments for each key color.

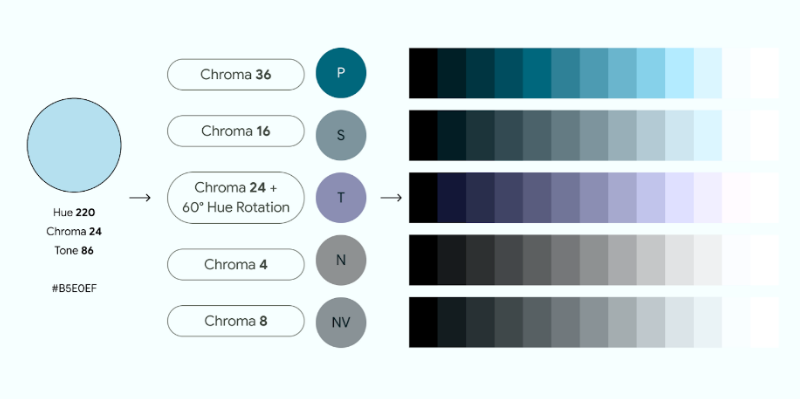


Figure 2. When generating a scheme from one source color, its HCT is modified to create the five key colors. Specific tonal values are then assigned to a color scheme.

If you manually assign a key color, note the input's chroma and tone, as the color's tone may not be the color role's tonal value.

Figure 3. Inputting a color, revealing the HCT values. While the hue persists, the input color has a tone of 86, so it will be close to the Primary Container input, but not Primary.

The M3 color system is powered by the Material Color Utilities (MCU), a set of color libraries containing algorithms and utilities that make it easier for you to develop color themes and schemes in your app.

The following video describes how color schemes are derived.

Color limitations

Color limitations are the physical limits of color—whether it's the actual physics, our own biological visual limitations, or the limitations of on-screen color rendering. For example, some hues cannot exist with certain chroma or tones. Color limitations are the reason colors such as light blue or bright light red are not quite possible. Tone color mapping must be consistent across all hue values.

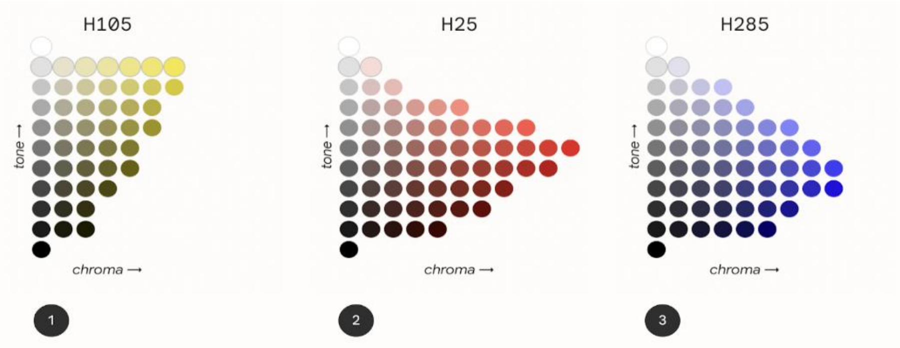


Figure 4: Tone mapping charts for H105, H25, and H285 values.

The preceding figure shows three different tone mapping charts for the H105, H25, and H285 hue values.

- **Chart 1–hue 105 (yellow)**. Indicates the availability of color. Chroma and Tone work like a graph. The yellow hue has limited chroma with certain tones along the graph, yellow does not have a wide range of vibrancy at lower tones.
- **Chart 2–hue 25 (red)**. Shows more chromatic options than hue 105 (yellow). In this tone map, the point of highest colorfulness is at a lower tone level.
- **Chart 3–hue 285 (blue)**. Shows that the highest colorfulness is found at an even darker tone. On the flipside, chroma capacity is lost at a lighter tone.

Color scheme

A color scheme is the set of accents and surfaces derived from specific tones and assigned to color roles, which are then mapped to UI elements and components. Color roles refer to the color's use, rather than the color's hue. For example, on-primary rather than on-blue.

Color schemes are designed to be harmonious, ensure accessible text, and distinguish UI elements and surfaces from one another. Color role pairs (composed of container and on-container roles) have tonal values that provide accessible contrast.

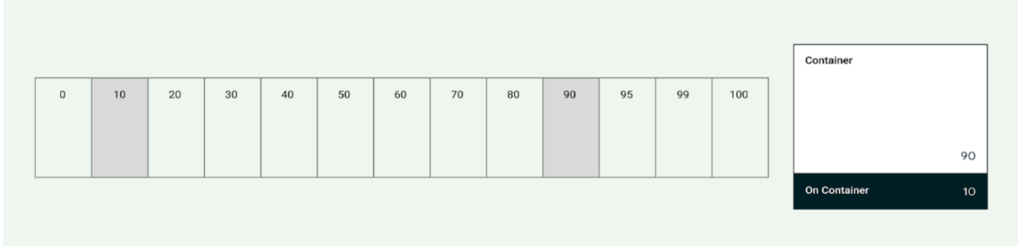


Figure 5: Color schemes are composed of multiple color groups and pairings derived from specific palette indices.

Light and dark schemes are created and have their own specific tone assignments.

The Material color system and custom schemes provide default values for color as a starting point for customization.

Learn more about the M3 color system.

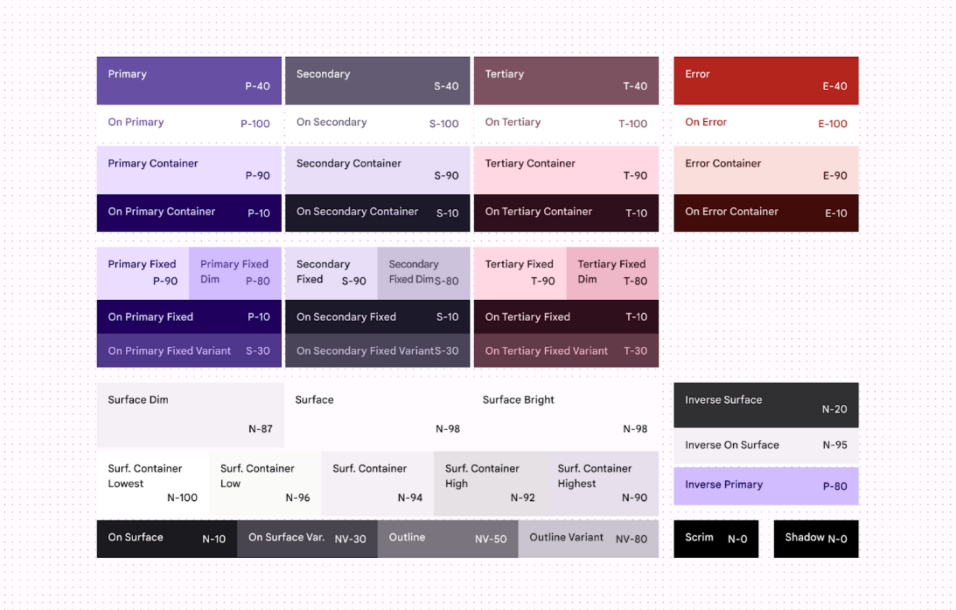


Figure 6: M3 light color scheme.

For a customizable color scheme, explore the Android UI Kit.

Apply color to your UI

UI color consists of accent, semantic, and surface colors.

- Accent colors refer to core colors that are typically part of the Android brand color palette.
- Semantic colors (or custom colors within Material 3), are colors with specific meaning.
- Surface colors refer to any neutral derived colors used for background colors.

Accent color

Accent colors usually exhibit the most expressiveness within a UI, whether it's for branding, highlighting actions, personal expression, or user expression.

Each accent color (primary, secondary, and tertiary) is provided in a group of four to eight compatible colors of different tones for pairing, defining emphasis, and visual expression.

Dynamic color

Accent colors can be defined from dynamic sources.

Starting in Android 12 (API level 31), dynamic color lets the system extract a source color from a user's wallpaper or in-app content, like a keyart asset. Dynamic color uses MCU algorithms and

processes to create schemes and implement them with little effort. To apply dynamic color to your app, read Enable users to personalize their color experience in your app.

Try out the code lab for Visualizing Dynamic color for a hands-on look at dynamic color.

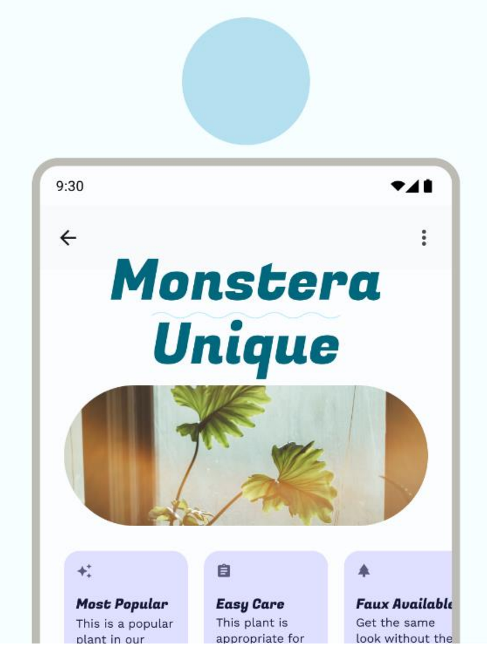


Figure 7: App color derived from a single source color.

Static

A static scheme is a scheme that has unchanging (or relatively) values. A common way of creating a static scheme is with brand colors, aligning primary, secondary, and tertiary colors to the brand's main color palette.

Even if you're using dynamic color, we strongly recommend creating a static scheme as a fallback if dynamic color isn't available to the user's device. Otherwise, the system uses the baseline purple color scheme.

Using the Material Theme Builder, you can apply the MCU color algorithm to generate a static, custom theme. This results in colors that you've chosen, but which are aligned to the M3 color system tokens and harmonious accessibility principles.

It's still possible to create a fully customized static scheme. To do this, assign different values within the color styles (color.kt or color.xml), or export the theme file from Material Theme Builder for Figma after updating the Figma style properties.

Figure 8. An app with colors derived from interpreted key color inputs (left), and a fully custom static scheme (right).

Usage

Material components have preassigned color roles, but you can utilize color tokens throughout your UI and custom elements. Use all accent colors mindfully, taking into account that the human eye is particularly drawn to vibrant colors.

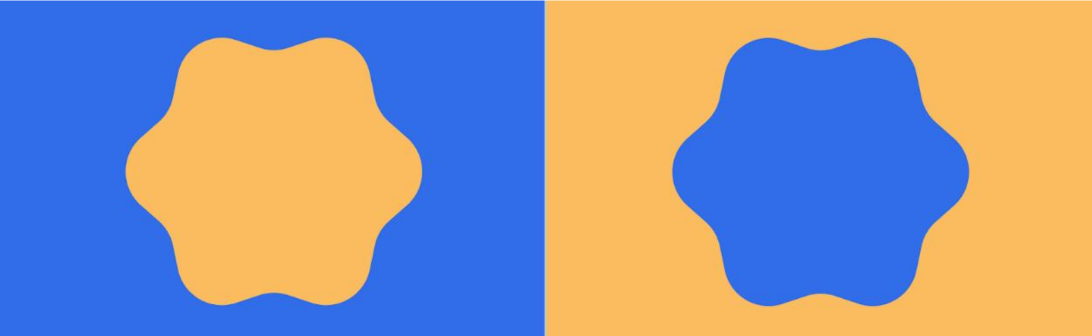


Figure 9. A human's eyes view cool vibrant color objects as foreground objects.

As with type, the system applies color in a hierarchy, with primary color and its respective roles assigned to crucial calls to action (CTAs). We recommend components such as floating action buttons (FABs) to take on primary roles.

When you're choosing a primary color, it's a good idea to assign your brand's primary color. Alternatively, you can select a color to represent interactive components, allowing your brand colors to be used more sparingly. Secondary and tertiary colors continue down the hierarchy of highlighting importance.

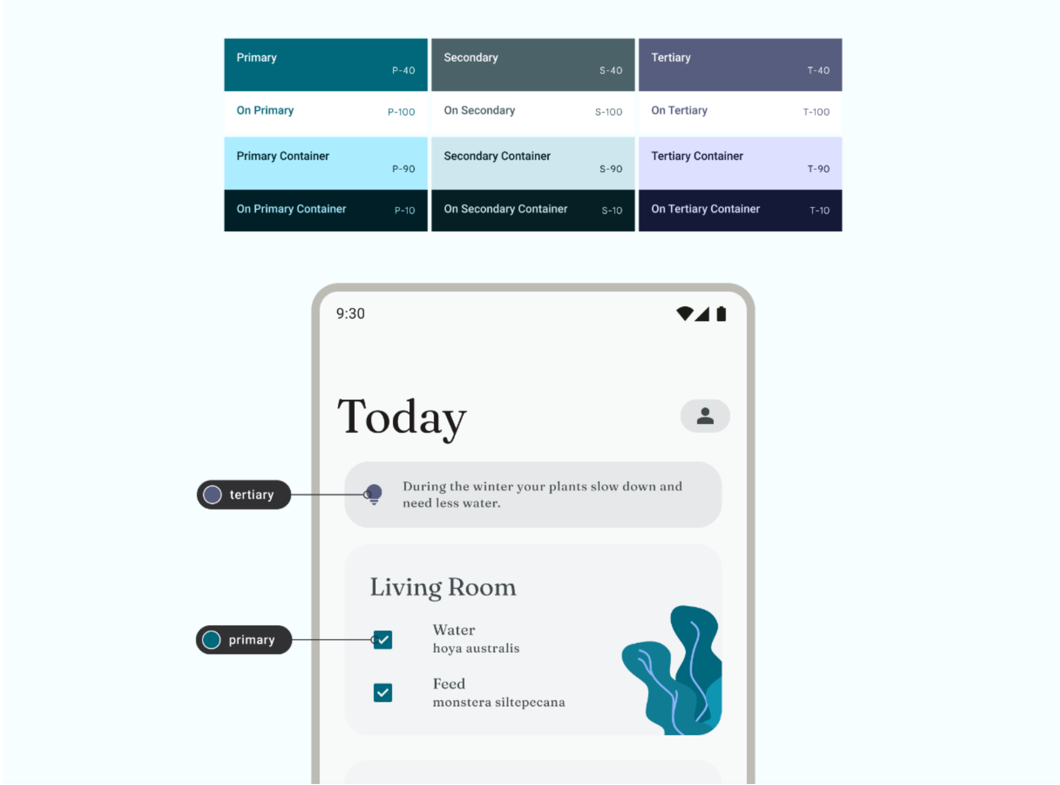


Figure 10. Application of an accent color in an app used in primary controls.

An oversaturated look can result in using only the base color roles of primary, secondary, or tertiary. To help with your color hierarchy, apply color schemes to include less vibrant container colors and outline roles.

To ensure a better user experience, use more vibrant primary colors to signify actions of greater prominence in your app's visual hierarchy. In the following figure, the FAB in the first image has a muted color with the same tone and chroma as navigation, making it blend in. The second image shows a FAB that calls more attention to itself with a vibrant primary color.

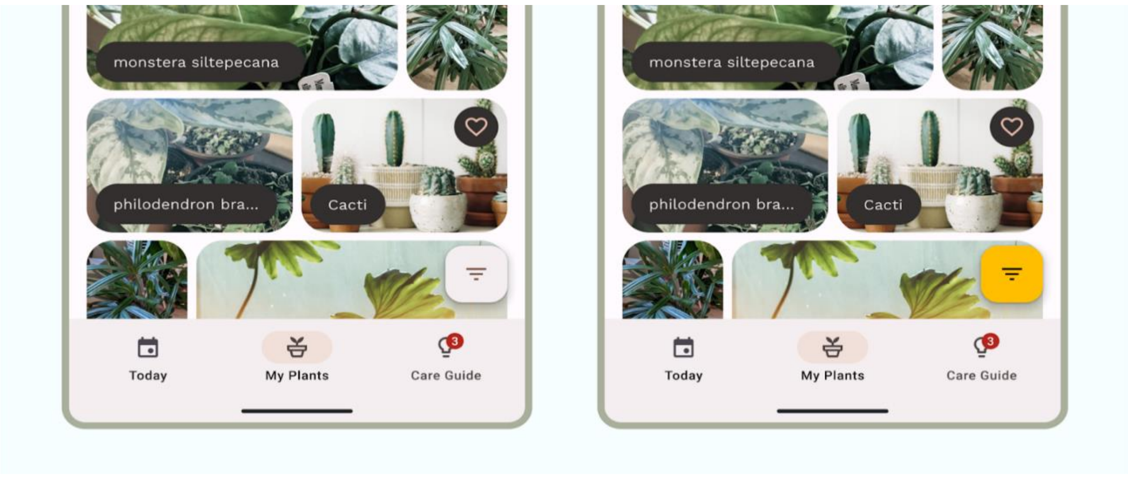


Figure 11. A FAB with muted color that blends in (left), and a FAB with a vibrant color that makes it stick out (right).

For a hands-on look at dynamic color, try out the code lab for Customizing Material color.

Semantic color

Semantic color are colors that have assigned specific meanings. For example, *Error* is a semantic color.

Be consistent with the meaning of color–if you establish a pattern, repeat it throughout the app. For example, if you have established purple to indicate a membership feature, use purple for all instances of this membership feature.

In the following example, an app uses red to indicate an error in one text field, but purple for the other–this would cause confusion when skimming a form.

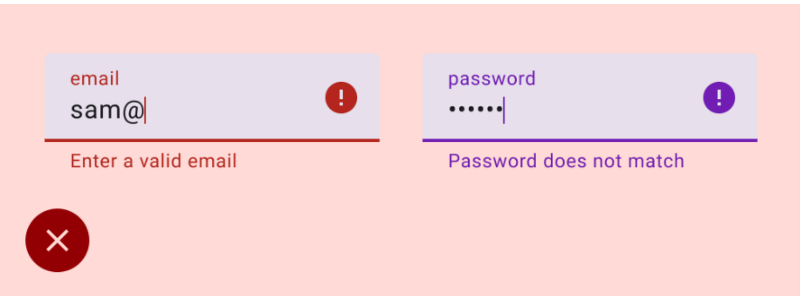


Figure 13: Example of poor consistency in text error colors.

Although the Material color scheme provides the semantic *error* color, additional semantic colors are created through custom colors to extend your color scheme. Read more about custom colors.

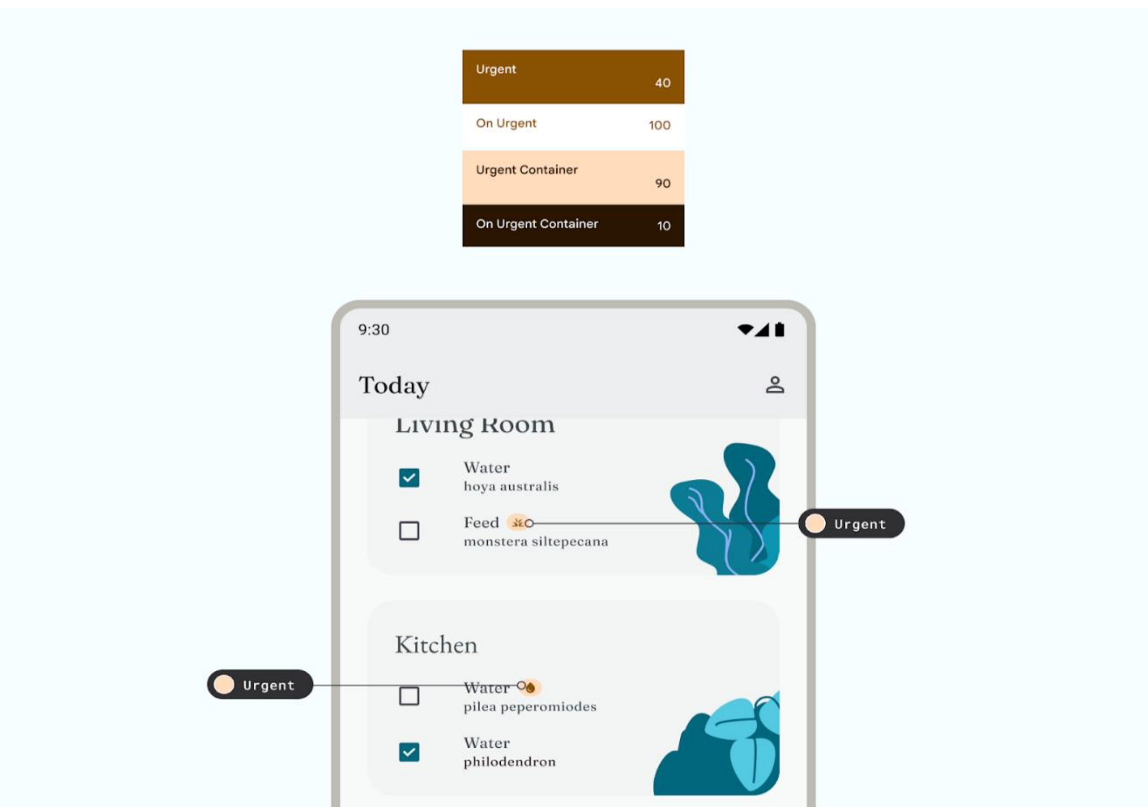


Figure 14. Application of a semantic color: an app alerting the user to an urgent task

Harmonization provides a way to align dynamic user-generated color with custom colors within your app to create more harmonious color palettes.

Surface colors

Surface colors are designed for background elements such as component containers, sheets, and panes, and represent the majority of your app's colors. Don't be shy to use lots of surface space; the human eye needs space to relax. Surfaces also help contain content and direct the reader.

M3 introduced the concept of tonal surfaces, meaning all colors are derived from the tonal palettes. Tones create both depth and more contrast to aid accessibility. For more on surface roles, see the surface roles M3 guidance.

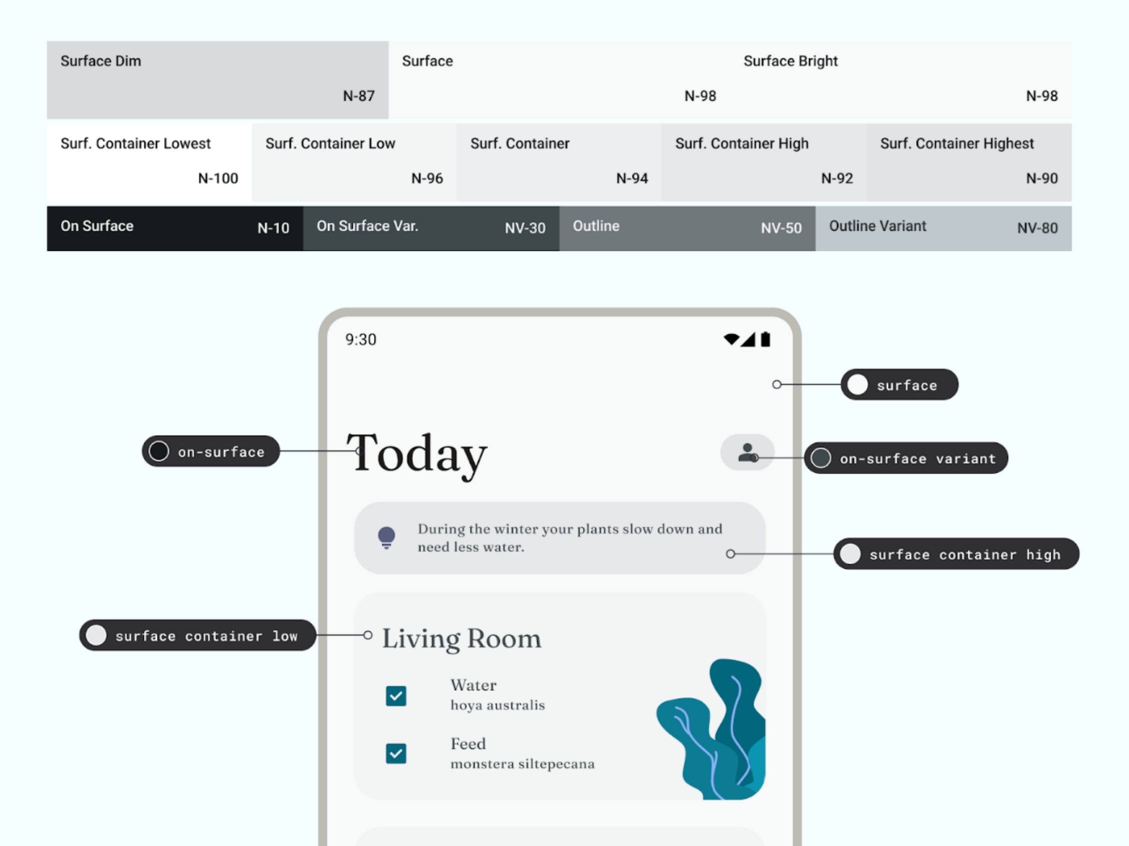


Figure 15. Application of surfaces in an app.

Accessibility and color

People view color in various ways depending on their visual acuity. Because some readers are color blind, you need to check color combinations to ensure UI elements don't blend together. Although opacity and weight might not be the literal hue of a color, they have a powerful visual effect on how users perceive color.

Color contrast is the difference between the luminance of the foreground and background elements, presented in a ratio format. This ratio criteria is given grades. For example, measuring the contrast between text on a button and its container helps determine the legibility of the text. Color contrast guidelines are divided into text and non-text, each with their own set of grades.

Never make color the only affordance or indicator for an available action. Utilize a component button, change of font weight, or even an icon to help inform your user that they can interact with the element.

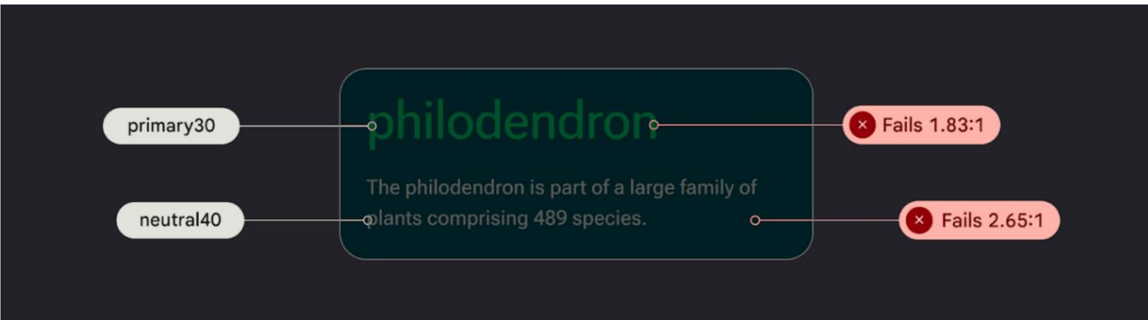


Figure 16. Color contrast

Implement color

Tokens are small variable semantic representations of design data. They are repeatable and replace static values, such as hex codes for color, with self-explanatory names. To assign the color role of an element, use tokens instead of hard-coded color values.

Check out the [Now in Android Figma sample](#) for examples of color role mapping.

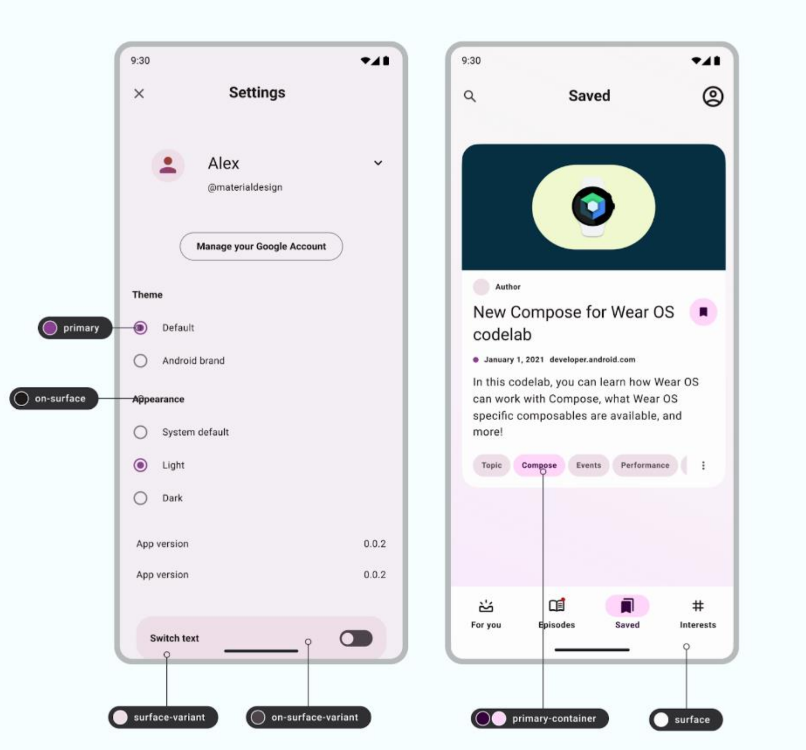


Figure 17: UI with assigned tokens

Color values are set within a color file color.kt using Compose (or color.xml using Views). These colors set as styles are part of a theme. See more about this in [Design for Android mobile themes](#).

To set color values on Android, use hex code, which represents RGB in a 6-digit format. To capture opacity, append the value to the front to make an 8-digit code.

Using Material Theme Builder:

You can create customized light and dark color schemes using the Material Theme Builder (MTB).

MTB lets you visualize dynamic color, generate Material design tokens, and customize your color schemes.

The color scheme can be fully customized by updating style properties within the Figma inspector panel. These modified values are exported.

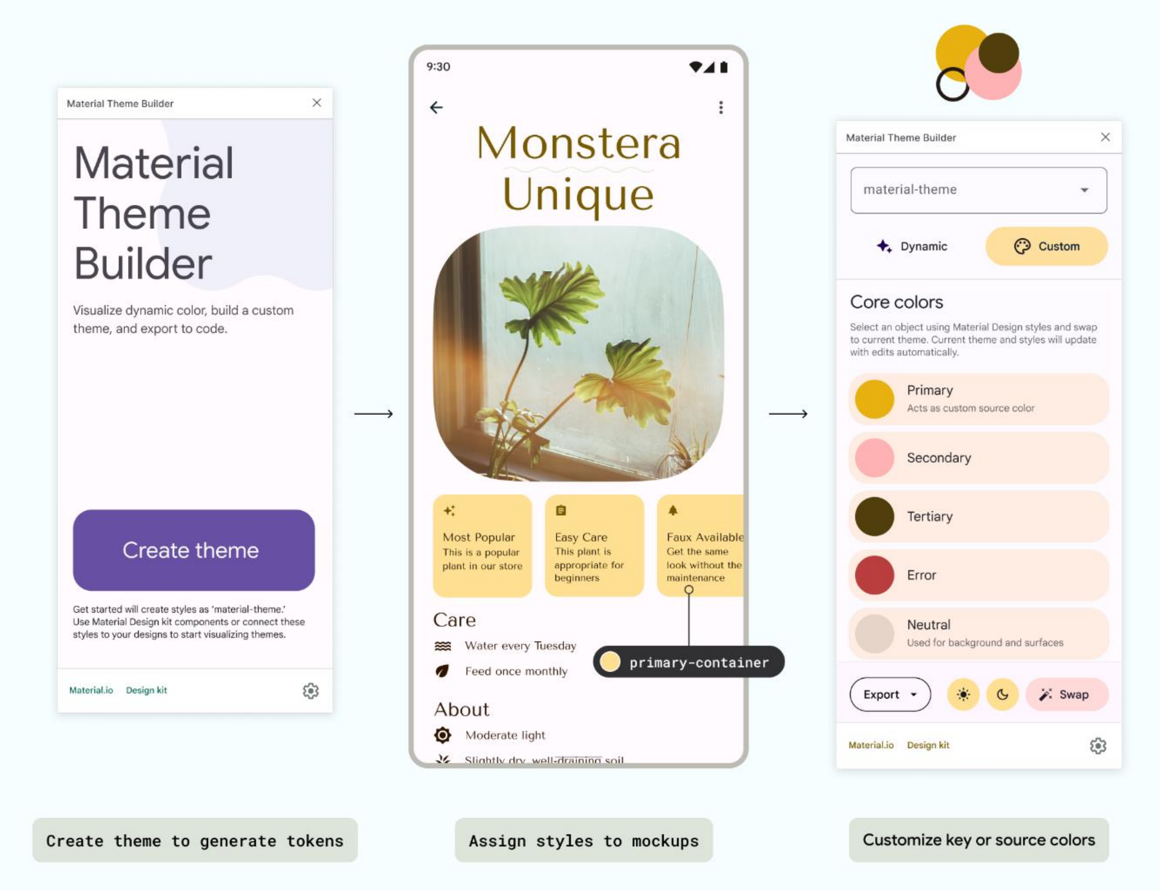


Figure 18: Customizing style properties and exporting color schemes. (turn on state layers in Settings for design kit). Or download a color file to assign color values using export.

Choosing a Color Palette for Your Project

Somewhere between wireframing and prototyping, you need the right color selection for your design project. In some respects, your color palette matters just as much as the structure of your design. Given its importance, you don't want to pick colors without careful consideration.

The following information should help you choose a color palette that makes your products more attractive and useful.

Did you pick the colors that you need for your UI? Then, let's jump to design. Create advanced, interactive prototypes using UXPin. Improve your design process by prototyping user interfaces that feel and look like a real product. Make design consistency a no-brainer. Sign up for a trial.

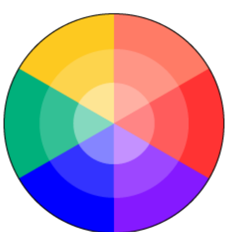
Build advanced prototypes

Design better products with States, Variables, Auto Layout and more.

[Try UXPin]

Understanding color theory

The color wheel has a deceptively simple look. At first glance, you might assume that it gives you little more than a selection of colors.



The more you study it, the more complex it becomes. You will discover that it gives you examples of:

- Cool and warm colors. Cool colors include blue, green, and purple. Warm colors include red, orange, and yellow.
- Value (the tint or shade of a color).
- Saturation (the color's intensity).

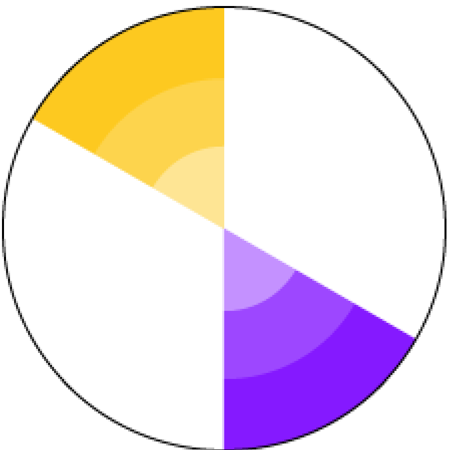
More insight reveals nearly countless ways that you can use the color wheel to create color palettes.

Different types of color palettes

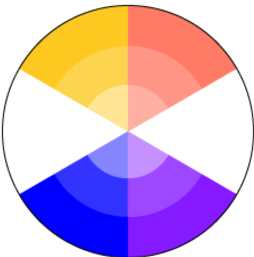
Some popular color palettes include:

- Analogous color schemes: Color palettes based on colors located next to each other on the color wheel

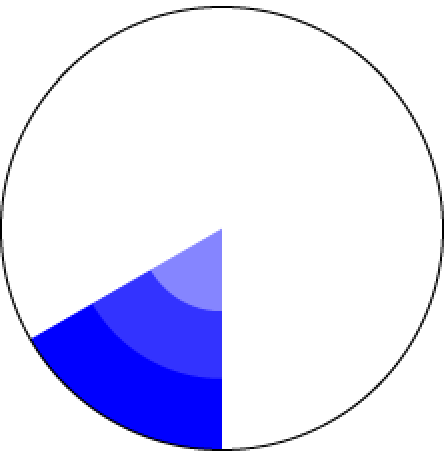
- Complementary color schemes: Color palettes based on colors at opposite sides of the color wheel.



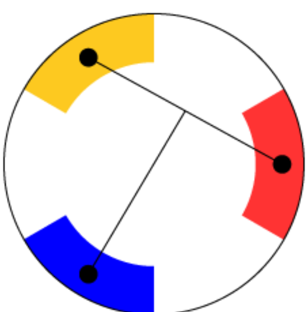
- Double complementary color schemes: Similar to a complementary color scheme, except you use four points on the color wheel instead of two.



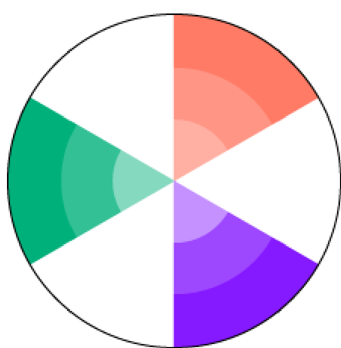
- Monochromatic color schemes: Color palettes that use variations of the same color.



- Split Complementary color schemes: Color palettes that start like a complementary color scheme, except you choose one color and the two colors next to its complementary color.



- Triadic Complementary color schemes: Color palettes that use triangles on the color wheel



60-30-10 rule for color use

The 60-30-10 rule gives you an easy way to choose a color palette and stick to it. When done well, it can also help establish a brand's identity.

With this rule, you use a primary color 60% of the time; a secondary color 30% of the time; and an accent color 10% of the time.

The rule works especially well in website design because you can keep your work clean and simple. For example, you could use your secondary color as a light background for your page.

Headers, subheads, and other critical aspects of the design can appear in the primary color. Finally, you can use the accent color to add a little panache to buttons or small pieces of text.

Trending color palettes

Trending color palettes change constantly. Don't fall too deeply in love with one, because it will change before the year ends. Some color palette options expected to trend during 2021 include:

- Human skin tones.
- Super-saturated colors.

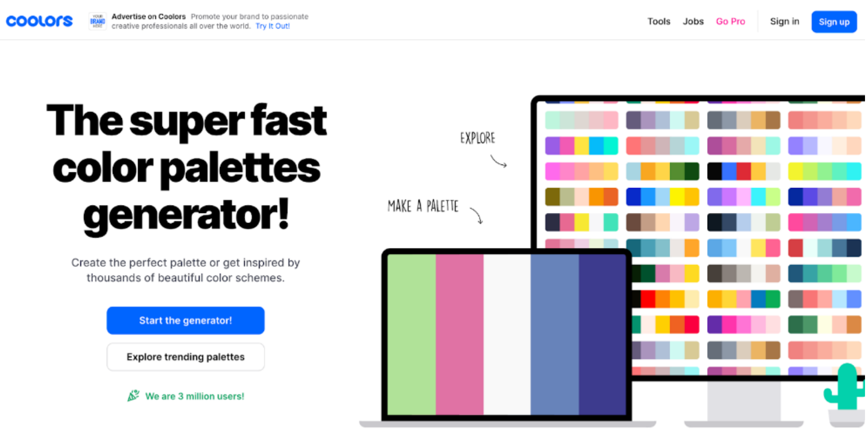
- Textured colors that look faded.
- Surreal, eye-popping color combinations.

Not every product wants a trendy color palette but rather goes for timeless brand colors. A client with an established brand, for example, might reject new color palettes that feel contradictory to the personality customers and clients already know.

Color tools and apps

Struggling to develop the perfect color palette for your design project? There's plenty of online tools and apps can help you explore creative options that will make your product stand out from the pack no matter if you like pastel, retro, or whatever your color choices are.

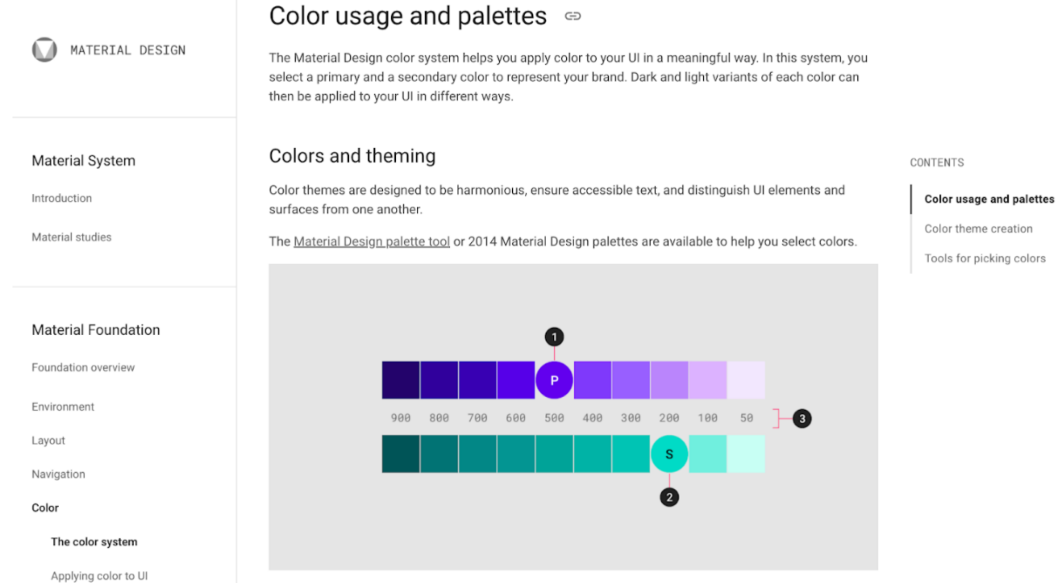
Coolors



One of the most popular color scheme generators in recent years, Coolors has tools designed to help you choose colors for websites, smartphone apps, and other products.

Many people like Coolors because they can create free accounts that let them generate color schemes by sampling online images, entering hex codes, or exploring trending color palettes.

Material Colors



If you use Google's Material Design, you might as well rely on the Material Design color system. The tool promises to help you use color in ways that add meaning to your UI.

Material Design has a wealth of tools that can use to create imaginative color combinations. It doesn't assume that you've mastered the color wheel, though. If you feel lost, use Google's tutorials and videos to learn more about how colors can influence the way people use your products.

Accessibility concerns for color blindness

Some people perceive visuals differently. Those with color blindness, for example, often struggle to identify green, red, blue, or yellow colors.

About 300 million people in the world live with some type of color blindness, so it makes sense to choose color palettes that they can experience vividly.

A tool like Venngage makes it easier to automatically use color palettes that are friendly for people with color blindness. You can also make designs easier to understand by using high-contrast colors, adding more textures to colors, and including more symbols that don't rely on color for meaning.

Improve brand consistency with color palettes in UXPin

No matter what color palette you choose for your next design, you need to establish consistency that helps users understand how the products work. If your color palette changes from page to page, visitors will feel very confused.

You also need to make sure all of your designers know what color palette they should use on each project. UXPin helps managers retain control of projects by creating design systems that limit the color palettes designers can access.

When you have a design system, you set up guardrails that let designers explore their creative impulses without going off-brand. A good design system can also streamline your process and limit the number of changes that you need to make before finishing the product.

The Impact of Color in Mobile App Design: Psychology, Theory, and Trends

Color plays a significant role in mobile app design, influencing user perceptions, emotions, and the overall user experience. Knowing about the psychology of colors, designing theories, and current trends can help you harness the power of color to create visually appealing and effective mobile apps. In this article, we will explore the impact of color in mobile app design and how it can enhance user engagement.

The psychology of colors:

Different colors evoke distinct emotions and associations. For example, blue is often associated with trust and security, while red can evoke passion or urgency. Understanding the psychological impact of colors can help you choose the right color scheme that aligns with your app's purpose and target audience. Consider the emotions you want to evoke and the message you want to convey through your app's design.

Color harmonies and theories:

Design theories such as color harmonies provide guidance on selecting colors that work well together. Common color harmonies include complementary (using colors opposite each other on the

color wheel), analogous (using colors adjacent to each other), and triadic (using colors evenly spaced on the color wheel). These harmonies can create visual balance and harmony in your app's design.

Branding and consistency:

Colors play a crucial role in branding and creating a recognizable app identity. Choose colors that align with your brand's personality and values. Consistency in color usage across different screens and elements within your app promotes a cohesive and unified user experience. Create a style guide that outlines color palettes, usage guidelines, and accessibility considerations to maintain a consistent visual identity.

Accessibility considerations:

Color choices should also consider accessibility for users with visual impairments. Ensure sufficient contrast between text and background colors to ensure legibility. Utilize color-blind friendly palettes and provide alternative ways to convey information, such as using icons or patterns in addition to color.

Current color trends:

Staying updated with color trends can give your app a contemporary and fresh look. Research current color trends in mobile app design and evaluate how they align with your app's purpose and target audience. However, remember that trends can change quickly, so it's important to strike a balance between following trends and maintaining a timeless design.

A/B testing and user feedback:

Colors can have subjective effects on individuals, so it's crucial to test and validate your color choices. Conduct A/B testing by presenting different color variations to different user groups and gather feedback on their preferences and perceptions. User feedback is invaluable in fine-tuning your color choices to create an app that resonates with your target audience.

The margins!

Margins are spaces that sit between content and the edges of the screen. They are defined with fixed width — usually 16 px, but some apps use 20 or 24 px margins. All the sizes of UI elements and spaces between them are set as a multiple of the number 8. Why 8? Because it's easy to divide most mobile screens into an 8pt grid.

TIP: iPhone 13 mini screen has 375 px width, so you'll have to steal 1 px from a component to keep an 8pt grid. Don't worry, devs use percentages anyway.

Things between the margins

The content is placed inside the columns (within margins). These help us make decisions on where to place an element and make a well-structured interface. Columns help developers to create consistent layout across multiple screen sizes. Most common number is 4, but you can try 6 or 8. The width is defined using percentages, rather than fixed values.

TIP: Not every element in the component has to align with the vertical column. Align the bounds of the components with the columns.

What about gutters?

Gutters separate the content in the columns. The recommended value for the gutters is also 16 px. Less than 16 px is usually not enough to keep elements visually separated, but hey, maybe 8 px might work sometimes. Try it out for yourself.

Should I divide the screen horizontally too?

A horizontal UI grid is used to horizontally align UI elements — such as buttons, cards, and switches. The horizontal grid is set on an 8 px rhythm (here's that number 8 again). This will ensure that the vertical and horizontal spacings are in sync.

Baseline

The text sits on a baseline grid. The baseline grid helps you to keep the rhythm of various texts on the screen consistent. If you want to know more, you can search for "4 px grids".

Ok, but how do I set that up?

It's not that hard, assuming you use Figma. Here are the steps:

1 -Create a new frame (shortcut **F**), in this case iPhone 13 mini

2-Select the frame → Add Layout grid → Layout grid settings

3-In the Layout settings window type in the values

A grid is like invisible glue that holds a design together. Even when elements are physically separated from each other, something invisible connects them together.

While grids and layout systems are a part of the heritage of design, they're still relevant in this multiscreen world we live in. Technology devices have fundamentally changed the way we search for information and how we function in our daily lives. Today, 90% of all media interactions are screen-based, where content is viewed across mobile phones, tablets, laptops, TVs and smart watches. Multiscreen behavior is quickly becoming the norm, and designing for multiple screens has become integral to businesses. As designers, we want to provide delightful and enjoyable experiences to people who use our products — and grids can help us do that.

Grids help designers to build better products by tying different design elements together to achieve effective hierarchy, alignment and consistency, with little effort. If executed properly, your designs will appear thoughtful and organized.

In this article, I've put together a lot of information about grids, such as:

- what grids are,
- a brief history of the grid,
- a basic theory of grids,
- four types of layout grids,
- layout grids in interactive design.

How does this apply to e-Learning?

E-Learning comprises all forms of electronically supported learning including, computer-based learning, the internet, cellular phones and tablets. Implementing the Hierarchy of Information in e-

Learning would entail considering how each electronic device functions, what its limitations are, how it displays information and how users interact with the device.

The challenge is then to make sure your design creates an order of importance and facilitates comprehension while at the same time adhering to any device's functions, limitations and display features. In other words, the design must be compatible with both the content and technology.

Why is Visual Hierarchy Important in Design?

Visual hierarchy is important in design because it defines the importance and sequence of elements within a composition. It influences the order in which your audience views your content. Order can significantly impact comprehension, impact, and value.

Naturally, there are elements with a composition that are more important than others or that contextualize other elements. You can use various visual hierarchy principles to draw eyeballs to those important elements first.

Good design uses visual hierarchy strategically to attract the viewer to the "whole" composition and leads them through its "parts" by creating different levels of priority and intuitive flow.

Graphic designers understand how the human brain perceives visuals and the influence those visuals have on cognition.

Understanding Information Architecture and Navigation

Before diving into the specifics of creating intuitive navigation and information architecture in mobile apps, it is important to understand the concepts of information architecture and navigation. Information architecture involves organizing and labelling content in a way that enhances findability and usability. It focuses on classifying content, establishing user-centric relationships, and defining naming conventions for easy navigation. On the other hand, navigation refers to the components and patterns that guide users through the app, helping them find information and functionality quickly and intuitively.

Importance of Intuitive Navigation and Information Architecture

Intuitive navigation and information architecture are essential for mobile apps to provide a seamless user experience. When users can easily navigate through an app and find the information they need, they are more likely to engage with the app and achieve their goals efficiently. Here are some key reasons why intuitive navigation and information architecture are important:

1. **Enhanced User Experience**: Intuitive navigation and information architecture make it easier for users to interact with an app, resulting in a positive user experience. When users can find what they need quickly, they feel more satisfied and are more likely to continue using the app.
2. **Improved Findability**: Effective information architecture ensures that content is organized and labelled in a logical and meaningful way. This improves the findability of information, allowing users to locate specific content without wasting time searching.
3. **Increased Engagement**: When users can easily navigate through an app and find relevant content, they are more likely to engage with the app and spend more time exploring its features. This increased engagement can lead to higher retention rates and improved user satisfaction.

4. **Reduced Cognitive Load**: Intuitive navigation and information architecture reduce the cognitive load on users. When an app is well-organized and easy to navigate, users don't have to spend excessive mental effort trying to figure out how to find what they need.

Principles for Creating Intuitive Navigation and Information Architecture

To create intuitive navigation and information architecture within mobile apps, designers should consider the following principles:

1. Understand User Needs and Goals

The first step in creating intuitive navigation and information architecture is understanding the needs and goals of the target users. Conducting user research and gathering insights about user preferences and behaviours can help inform design decisions. By understanding what users are looking for and how they expect to find it, designers can create a navigation structure that aligns with their mental models.

2. Keep it Simple and Consistent

Simplicity and consistency are key principles in designing intuitive navigation and information architecture. Users should be able to navigate through the app without encountering any confusion or surprises. Keep the navigation structure simple and straightforward, avoiding unnecessary complexity. Consistency in the placement and appearance of navigation elements helps users build familiarity and navigate the app with ease.

3. Use Clear Labels and Categorization

Labels play a crucial role in guiding users through an app. Use clear and concise labels that accurately describe the content or functionality behind them. Categorize content logically, grouping related items together. This helps users understand the structure of the app and find what they need more easily.

4. Prioritize Important Content

Mobile screens have limited space, so it's important to prioritize important content. Identify the key features or information that users are most likely to seek and make them easily accessible. Use visual cues such as size, colour, or position to highlight important elements and guide users' attention.

5. Provide Multiple Navigation Options

Different users may have different preferences when it comes to navigation. Provide multiple navigation options to cater to various user preferences. For example, include both a menu-based navigation and a search functionality to accommodate users who prefer either method.

6. Utilize Gestures and Touch Interactions

Mobile devices offer various touch interactions and gestures that can enhance navigation. Utilize swipe gestures, pinch-to-zoom, or drag-and-drop interactions to make the navigation experience more intuitive and engaging. However, ensure that these gestures are discoverable and don't require users to learn complex or hidden interactions.

7. Test and Iterate

Testing and iteration are crucial in creating intuitive navigation and information architecture. Conduct user testing to gather feedback and identify areas for improvement. Use analytics and user behaviour data to understand how users navigate through the app and make data-driven decisions for optimization.

Best Practices for Intuitive Navigation and Information Architecture

In addition to the principles mentioned above, here are some best practices to consider when designing intuitive navigation and information architecture in mobile apps:

1. Clear and Visible Navigation Elements

Make sure navigation elements, such as menus, buttons, or tabs, are easily visible and accessible. Use appropriate visual cues, such as icons or text labels, to indicate the purpose of each element. Avoid cluttering the screen with too many navigation options, as it can overwhelm users.

2. Progressive Disclosure

Consider using progressive disclosure to reveal content gradually. Instead of overwhelming users with a large amount of information at once, provide them with options to explore more details when needed. This helps to maintain a clean and organized interface while allowing users to delve deeper into specific areas.

3. Contextual Navigation

Contextual navigation refers to providing navigation options that are relevant to the current context or task. For example, when users are browsing a specific category, show related navigation options or filters to help them narrow down their search. This contextual relevance improves the efficiency of navigation and enhances the user experience.

4. Clear Feedback and Visual Cues

Provide clear feedback and visual cues to indicate the user's current location within the app and their progress in completing a task. Use visual indicators, such as breadcrumbs or progress bars, to help users navigate and understand their position within the app's structure.

5. Responsive Design

Ensure that the navigation and information architecture are designed with responsiveness in mind. Mobile apps are accessed on various screen sizes and orientations, so the navigation elements should adapt and remain usable in different contexts. Test the app on different devices to ensure a consistent and user-friendly experience.

6. User-Friendly Search Functionality

Implement a user-friendly search functionality that allows users to find specific content quickly. Include autocomplete suggestions, filters, and sorting options to enhance the search experience. Ensure that the search results are relevant and displayed in a clear and organized manner.

7. Continuous Improvement

Navigation and information architecture should be continuously monitored and improved based on user feedback and data analysis. Stay updated with industry trends and evolving user expectations to ensure that the app's navigation remains intuitive and user-friendly.

By following these principles and best practices, designers can create mobile apps with intuitive navigation and information architecture, ensuring that users can easily find what they need. Remember to prioritize user needs, keep the design simple and consistent, use clear labels and categorization, and provide multiple navigation options. Regular testing and iteration will help refine the navigation and information architecture to meet the evolving needs of users. With these strategies in place, mobile apps can deliver a seamless and engaging user experience, leading to increased user satisfaction and app success.

Day 2: Designing for Mobile Apps

Mobile-Specific Considerations
1.Screen Sizes and Resolutions:
Mobile devices come in various screen sizes and resolutions. Designing with responsiveness in mind ensures your app looks and functions well across different devices.
2. Touchscreen Interaction:
Mobile devices rely on touchscreens for user input. Design buttons, links, and interactive elements to be easily tappable with fingers. Ensure there's enough space between elements to prevent accidental taps.
3. Navigation Patterns:
Due to limited screen space, consider using intuitive navigation patterns like:
Hamburger menus for accessing secondary options.
Bottom navigation bars for core features.
Gestures (e.g., swipe, pinch) for navigation.
4. Thumb-Friendly Design:
Prioritize placing frequently used elements and navigation within the "thumb zone" for one-handed usability.
5. Orientation and Landscape Mode:
Design for both portrait and landscape orientations. Ensure essential functionality is accessible in both modes.
6. Optimizing Images and Media:
Compress images and media files to reduce loading times and conserve bandwidth.
7. Typography and Readability:
Use legible fonts and appropriate font sizes. Ensure there's enough contrast between text and background for readability on smaller screens.
8. Adaptive Layouts:
Design layouts that adapt to different screen sizes and orientations. Consider using flexible grids and elements that can adjust dynamically.
9. App Icon and Splash Screen:
Create an easily recognizable app icon and a visually appealing splash screen for a strong first impression.
10. Offline Functionality:
Provide offline capabilities or caching for essential features to ensure usability without an internet connection.
11. Reduced User Input:
Minimize the need for excessive typing or complex interactions. Leverage features like autocomplete and predictive text.
12. Feedback and Animation:
Provide visual feedback for user interactions to confirm actions and maintain a smooth flow. Use animations judiciously to enhance user experience.
Screen sizes, resolutions, and aspect ratios.
1. Screen Size:
Screen size refers to the physical dimensions of the display measured diagonally from one corner to the opposite corner.
For example, common screen sizes include 4.7 inches (iPhone SE), 5.8 inches (iPhone 11 Pro), and 6.5 inches (iPhone 11 Pro Max), android systems have their own sizes.

2. Resolution:
Resolution refers to the total number of pixels on the screen, usually represented as width x height (e.g., 1920 x 1080 pixels).
For example, a Full HD resolution is 1920 x 1080 pixels.
3. Aspect Ratio:
Aspect ratio is the proportional relationship between the width and height of the screen.
Common aspect ratios include 16:9 (widescreen), 4:3 (standard), and 18:9 (common for modern smartphones).

Day 3: Visual Design Basics

1. Hierarchy:
Hierarchy establishes the order of importance and guides the user's eye through the design. It's achieved through variations in size, color, contrast, and placement of elements.
2. Balance:
Achieving visual equilibrium in a design. It can be symmetrical (equal on both sides) or asymmetrical (unequal but balanced).
3. Contrast:
Contrast involves using opposing elements (like light and dark colors) to create visual interest and highlight important information.
4. Emphasis/Focal Points:
Design elements that stand out to draw the user's attention. They're used to guide the user towards important content or actions.
5. Unity and Consistency:
Maintaining a cohesive look and feel throughout the design. Consistency in elements like colors, fonts, and spacing creates a sense of unity.
6. Whitespace (Negative Space):
The empty space around design elements. It helps reduce clutter, improve readability, and highlight important content.
7. Typography:
Choosing and using fonts effectively. Consider factors like legibility, readability, and appropriateness for the brand or content.
8. Alignment:
Ensuring that elements are placed purposefully and follow a logical structure. This creates a sense of order and professionalism.
9. Repetition:
Repeating certain design elements (like colors, fonts, or shapes) helps create a consistent and cohesive visual experience.
10. Scale and Proportion:
Using different sizes and proportions of elements to create visual interest and hierarchy.
11. Texture and Patterns:
Adding depth and visual interest through the use of textures or patterns, which can be both physical (like in print design) or implied in digital design.
12. Images and Icons:
Choosing and using visuals (images, icons, illustrations) that complement the overall design and convey the intended message.
Color Theory and Usage
1. Color Wheel:
Understanding the color wheel is fundamental. It's a visual representation of colors arranged by their chromatic relationship.
2. Primary, Secondary, and Tertiary Colors:
Primary: Red, blue, and yellow - cannot be made by mixing other colors.
Secondary: Green, orange, and purple - created by mixing two primary colors.
Tertiary: Colors like red-orange, yellow green, etc. - created by mixing a primary and a secondary color.
3. Complementary Colors:
Colors opposite each other on the color wheel. They create strong contrast and reinforce each other's intensity.
4. Analogous Colors:
Colors that are adjacent to each other on the color wheel. They create a harmonious and cohesive look.
5. Triadic and Tetradic Colors:
Triadic: Three colors evenly spaced around the color wheel. They provide a balanced, vibrant palette.
Tetradic: Four colors together in the form of two complementary color pairs.
6. Monochromatic Colors:

Variations in lightness and saturation of a single color. This creates a clean, minimalist look.
7. Warm and Cool Colors:
Warm colors (reds, oranges, yellows) evoke energy and vibrancy. Cool colors (blues, greens, purples) convey calm and tranquility.
8. Psychology of Color:
Different colors can evoke different emotions or associations. Consider the context and target audience when choosing colors.
9. Accessibility and Color Contrast:
Ensure sufficient contrast between text and background colors for readability, especially for users with visual impairments.
Choosing an appropriate color scheme.
Choosing an appropriate color scheme is a critical aspect of visual design. It sets the tone, conveys emotions, and influences how users perceive and interact with your design. Here are steps to help you select a fitting color scheme:
1. Define Your Brand or Design Goals:
Consider the purpose and message of your design. Are you aiming for a professional, playful, calming, or vibrant look? Understanding your brand's personality or your design's intent will guide your color choices.
2. Consider Your Target Audience:
Think about the preferences, tastes, and cultural backgrounds of your intended audience. Different colors can evoke different emotions and reactions in different cultures and demographics.
3. Start with a Base Color:
Choose a primary color that will dominate your design. This color will often represent your brand or the main theme of your project. It could be a color associated with your company's logo or a hue that reflects the mood you want to convey.
4. Create Color Harmony:
Decide on a color scheme that complements your base color. There are various types of color harmonies to consider:
Analogous: Colors adjacent on the color wheel for a harmonious look.
Complementary: Colors opposite on the wheel for strong contrast.
Triadic: Three evenly spaced colors for vibrancy.
Tetradic: Four colors in two complementary pairs for versatility.
5. Consider Color Psychology:
Understand the emotional and psychological associations of colors. For example, blue often represents trust and calmness, while red can evoke energy and passion. Use this knowledge to reinforce your design's message.
6. Test for Accessibility:
Ensure that the chosen color scheme meets accessibility standards. High contrast between text and background colors is crucial for readability, especially for users with visual impairments.
7. Use Tools for Inspiration:
Online tools like Adobe Color Wheel, Colors, or even Pinterest can help you explore and create color palettes. They often provide pre-made palettes or generate harmonious combinations based on your chosen colors.
8. Consider Trends (with Caution):
While staying current is important, don't blindly follow design trends. Ensure that trendy colors align with your brand or project's goals and resonate with your target audience.
9. Test on Different Devices:
Colors can appear differently on various screens due to variations in display technology. Test your color scheme on different devices to ensure consistency.
10. Iterate and Seek Feedback:
Don't be afraid to experiment with different color combinations. Get feedback from peers or potential users to see how they perceive your chosen colors.
11. Maintain Consistency:
Once you've chosen a color scheme, use it consistently across your design. This helps build a cohesive and recognizable brand identity.

Day 4: Layout and Composition

What Is A Grid?

In the most basic terms, a grid is a structure comprising a series of lines (vertical or intersecting) that divide a page into columns or modules. This structure helps designers to arrange content on the page. While the lines of a grid themselves are not necessarily visible (although in some designs, they are), the structure helps you to manage the proportions between the elements to be aligned on the page. This grid would serve as the framework for the page's layout. Think of it as a skeleton on which a designer can organize graphic elements (for example, text sections, images and other functional or decorative elements) in an easy-to-absorb way.

The grid system originates in print design but has been applied to many disciplines. In fact, if we look around, we'll see that a lot of things we use daily were designed using a grid:

A bookshelf is a kind of grid

)

Barcelona's Eixample district shows how architects used a grid to lay out the neighborhood.

	A	B	C	D	E	F
1	Name	Team	January	February	March	April
2	Adam	SG	£4,100	£6,700	£3,500	£2,500
3	Amy	JJ	£5,400	£2,300	£7,200	£3,700
4	Jacob	SG	£6,100	£7,300	£3,500	£2,700
5	John	JJ	£7,200	£2,500	£2,800	£7,300
6	Nick	JJ	£3,700	£2,700	£3,500	£2,500
7	Sarah	JJ	£7,500	£8,300	£8,200	£2,800

Tables in Microsoft Excel are an example of a grid system applied to content.

Grids are usually applied to screen design. This page contains a grid of elements. (Image credit: Material Design)

Brief History Of The Grid

Before we dive into details on layout grids and how they can be applied to digital products, it's essential to step back and look to the past to understand the basics. This knowledge will help us better design for digital experiences. To learn more about the historical context of grids, be sure to check out Lucienne Roberts' article "A Brief History of Grids."

GRID AND EARLY BOOK DESIGN #

Grids are closely tied to typography. As a system, grids were first used to arrange handwriting on paper, and then were applied to manuscript layout. Since the early days of book design, the grid has helped designers arrange page layouts to aid the user in the act of reading.

RENAISSANCE ERA AND HARMONIOUS DESIGN

Paintings in the Renaissance era had a significant impact on the development of grid systems. Artists strived to create a perfect geometry, which resulted in centered and symmetrical canvas layouts, and it characterizes the work of artists in that period.

Baseline Grid

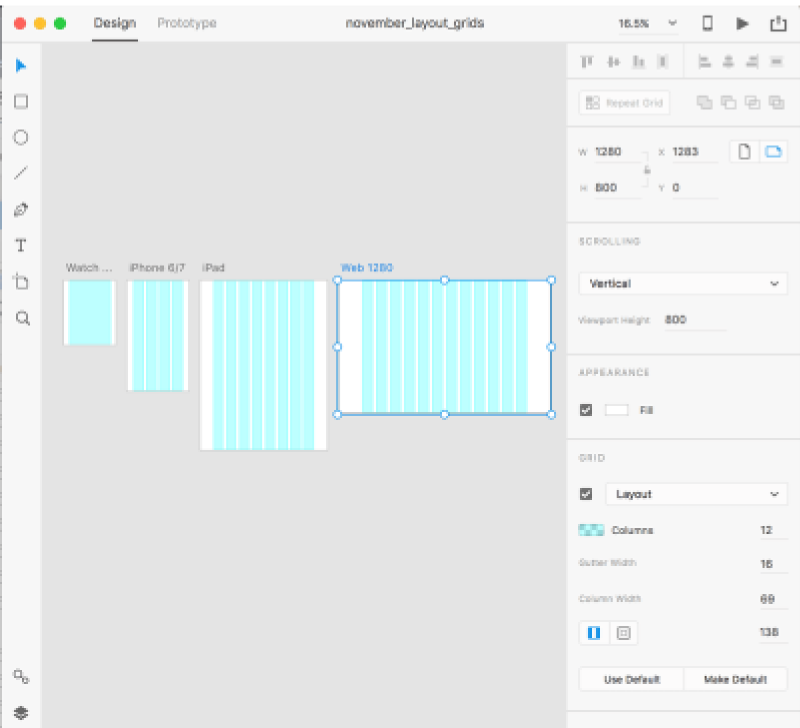
A baseline grid is an underlying structure that guides the vertical spacing in a design. It's used primarily for horizontal alignment and for hierarchy. Similar to how you would use columns and

modules as guides in your design, you can use a baseline grid to build consistency in your layout. Using this type of grid is akin to writing on a ruled piece of paper — the grid ensures that the bottom of each line of text (its baseline) aligns with the vertical spacing. This makes a baseline grid not only an excellent typographic tool, but also extremely helpful when you're laying out elements on the page because you can quickly check whether something on the page is missing a row of space.

A baseline grid shapes the vertical spacing of a design. Here, a modular grid is created by positioning horizontal guides relative to a baseline grid.

Layout Grids In Interaction Design

Interaction design changed the way we think about grids. Interaction design is fluid and doesn't have a fixed size because people are using different types of devices to interact with the product, from the tiny screens of smartwatches to ultra-wide TV screens. When using a product, people often move between multiple devices to accomplish a single task with that product. Despite screen size, designers must organize content in the most intuitive and easy-to-follow way. One approach to achieving this is to use a layout grid system. A layout grid is preferable for interactive design because it defines the underlying structure of a design and how each component responds to different breakpoints. This type of grid is faster and easier to design for multiple screens and resolutions.



L

ayout grids in Adobe XD shown across multiple screen sizes.

Grid systems in digital product design organize elements on the page and connect spaces. A grid system improves the quality of a design (functionally and aesthetically) and the efficiency of the design process in several ways:

- **Creates clarity and consistency**
  A grid is the foundation for order in a design. Proportion, rhythm, white space and hierarchy are all design characteristics that directly affect cognitive speed. Grids create and enforce consistency of these elements throughout an interface. An effective grid guides the eye, making it easier and more pleasant to scan objects on the screen. This is especially important

in digital products because they are functional, meaning that people use the products to complete specific tasks, such as sending a message, booking a hotel room or hailing a car ride. Consistency helps the viewer understand where to find the next piece of information or what step to take next.

- **Improves design comprehension**
  The human brain makes judgments in a fraction of a second. A design that is poorly put together will make the product seem less usable and trustworthy. Grids connect and reinforce the visual hierarchy of the design by providing a set of rules, such as where elements should go in the layout.
- **Makes responsive**
  Responsive design is no longer a luxury, but rather a necessity because people experience apps and websites on devices with a broad range of screens. This means that designers can no longer build for a single device's screen. The multidevice landscape forces designers to think in terms of dynamic grid systems, instead of fixed widths. Using a grid creates a consistent experience across multiple devices with different screen sizes.
- **Quickens the design process**
  Grids enable designers to manage the proportions between UI elements, such as spacing and margins. This helps to create pixel-perfect designs from the start and avoid timely reworking caused by incorrect adjustments.
- **Makes the design easier to modify and reuse**
  Unlike print production, digital products are never finished — they're constantly changing and evolving. Grids provide a solid foundation because when everything conforms to a grid, previous solutions can be easily reused to create a new version of the design. A grid is a skeleton that can be used to produce completely different looks.
- **Facilitates collaboration**
  Grids make it easier for designers to collaborate on designs by providing a plan for where to place elements. Grid systems help to decouple work on interface design because multiple designers can work on different parts of the layout, knowing that their work will be seamlessly integrated and consistent.

GRIDS ARE A FUNDAMENTAL PART OF STYLE GUIDES

Implementation of most design projects involves collaboration between designers and developers. Nothing is worse for a UI designer than submitting a pixel-perfect design mockup and finding that it looks completely different in production.

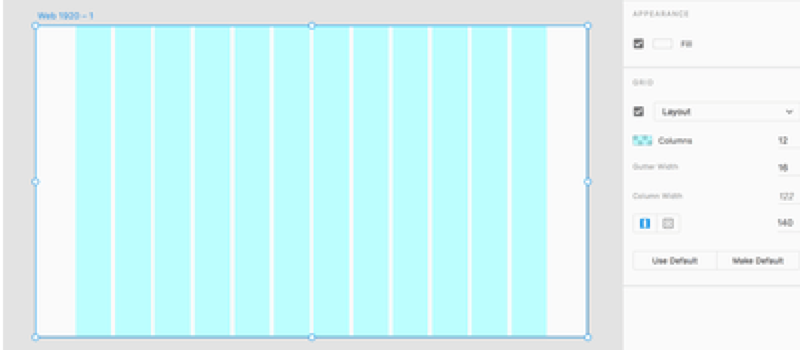
Grids are a framework that speeds up the designer-to-developer workflow by allowing developers to pre-set classes in their code that correspond to column sizes. This prevents inconsistent implementation and reduces the number of hours required to build a website. For more tips on how designers and developers can better work together, check out "Design Specifications: Speeding Up the Design to Development Workflow and Improving Productivity."

Best Practices For Layout Grids

While layout grids help designers to achieve a consistent, organized look in their designs and to manage the relationships and proportions between elements, there are a number of things to keep in mind when designing with a grid.

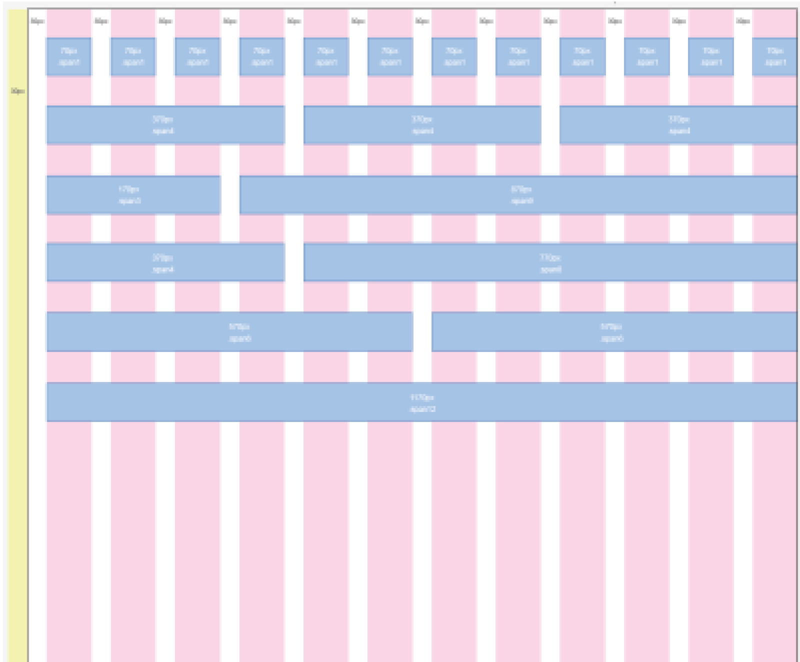
SELECT THE GRID YOU REALLY NEED

“How many columns?” is the first question designers ask when starting to work with a grid.



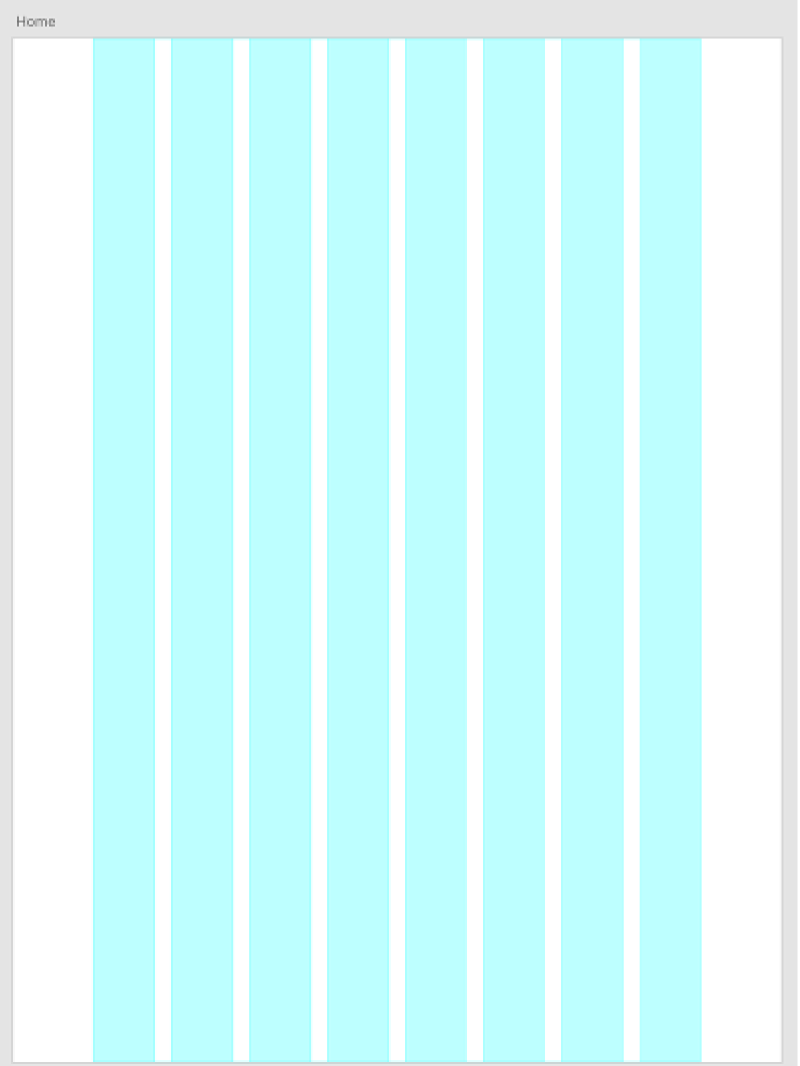
Adobe XD allows you to specify the number of columns, the gutter width and the column width.

Many popular frameworks use a grid system of 12 equal-widths column. The number 12 is the most easily divisible among reasonably small numbers; it’s possible to have 12, 6, 4, 3, 2 or 1 evenly spaced columns. This gives designers tremendous flexibility over a layout.



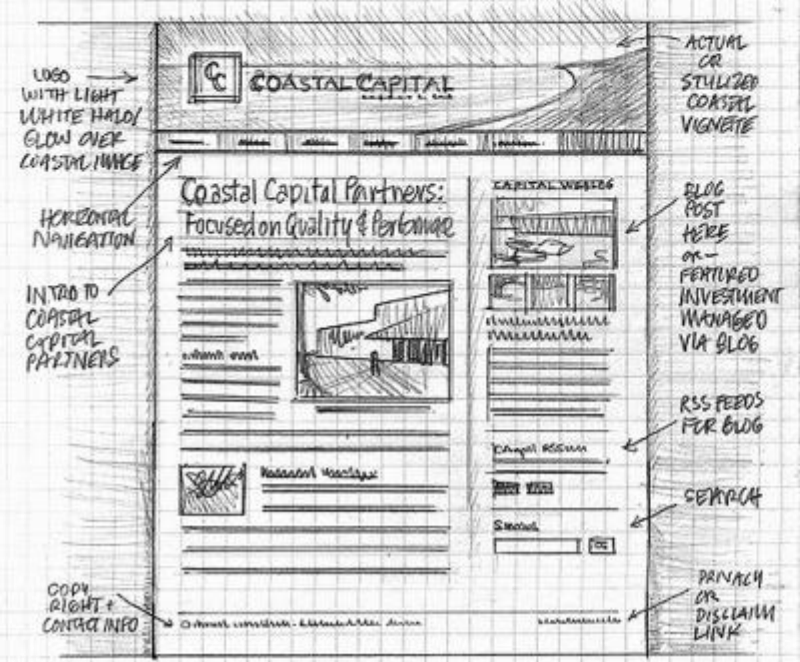
A grid system with 12 equal-width columns is robust and flexible and provides different ways of organizing the structure.

While the 12-column grid is a popular choice among many designers, it’s not a one-size-fits-all solution. When you choose a grid, select one with the number of columns you really need for your design. There’s no point in using a 12-column grid if your layout needs only 8 columns.



An 8-column layout grid in Adobe XD.

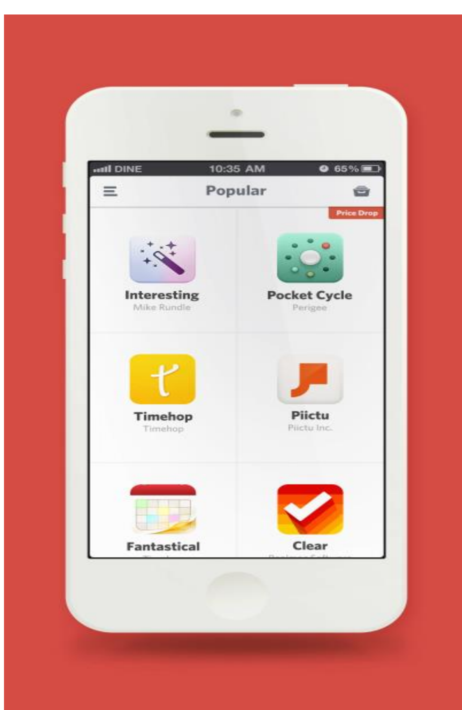
How do you know how many columns to use? Before deciding on the number of columns, sketch out your possible layouts (a paper sketch is fine). This means you'll need to know what content will be on the screen. The content will define the grid, not the other way around. With the sketches in hand, you'll be better informed on the number of columns you need.



A paper sketch for a web page layout. The number of columns should be defined by the content.

OPTIMIZE GRIDS FOR MOBILE

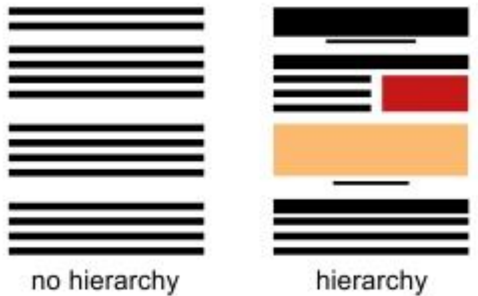
Mobile grids have limited space, making a multicolumn layout not really possible. Mobile content is typically limited to one or two columns. When designing for mobile, consider using a tile layout grid, in which the column and row heights are the same. This will give a look of square tiles across the design.



A tile grid on a mobile screen

On mobile, users have limited screen space and can only view a small amount of content at a time before having to scroll. Thus, when designing a grid layout, make images large enough to be recognizable yet small enough to allow more content to be seen at a time.

Design Principles: Hierarchy of Information



Hierarchy of information is the first of our new series of blog articles based on Design Principles in relation to e-learning design and development. This series will provide insight into various principles explaining their importance and why they are required in order to produce quality, well-crafted design.

The hierarchy of information is a universal design principle that should be used in all forms of design, including e-Learning design. By definition, it is the arrangement of elements or content on a page/screen in such a way that it reveals an order of importance (either ascending or descending).

What are the principles of the Hierarchy of Information?

Design elements can consist of anything including typography, graphics, colours, contrast, weight, position, size and space (including negative space). The trick is how you use these elements to accomplish the order of importance that you want.

Take this article for example, currently it's fairly plain, but if you click here, you can see what it would look like when the principles of the Hierarchy of Information have been put into practice.

I've had a go at listing some guidelines for applying the heirarchy of information to your own work:

1. Make a list of the different points of information that you're working with and order them numerically.
2. Now make sure that number one is standing out a little bit more than number two, you can do this by adjusting the elements of this particular point of information (i.e. its size, colour, weight, etc.).
3. Carry on with this throughout the list and then you would have created a descending hierarchy of information (do the reverse for an ascending hierarchy).
4. And remember it doesn't matter in which direction the document flows, just as long as it flows in a specific direction.

The principles of the hierarchy are actually pretty simple and easy to implement, but can have a massive impact on how your message is perceived and received.

Why is the Hierarchy of Information Important?

The hierarchy of information design principle allows the designer/developer to point out to the viewer what he wants the viewer to see first. This is very important in today's society as most people are in a hurry, even on the web. With the hierarchy principle, a designer can 'shout out' what he thinks is most important on the page/screen before the viewer gets bored and moves on.

If the hierarchy is done well, then the content should naturally become an easy read. It creates a path for the viewer's eye to follow through the page/screen. The viewer should be able to scan through the document and still get the picture. When it comes to designing layout this principle is functional and very effective.

Day 5: Interaction Design

Understanding User Flows

Creating intuitive navigation paths.

What is Micro Animation?

Micro animations are intricate reactions and events triggered by a specific user interaction on a website or mobile app. These design elements typically indicate a task is in progress or completed while creating an enjoyable mobile app user experience.

How Do Micro Interactions Help a Mobile App's User Experience?

These micro interactions and events are meant to engage the user in the task they are completing. Small animated details help guide users through their tasks, telling them when they've completed an action and gently suggesting next steps. They also add excitement and flair to an ordinary task, making users more connected to your brand and encouraging them to keep using your app.

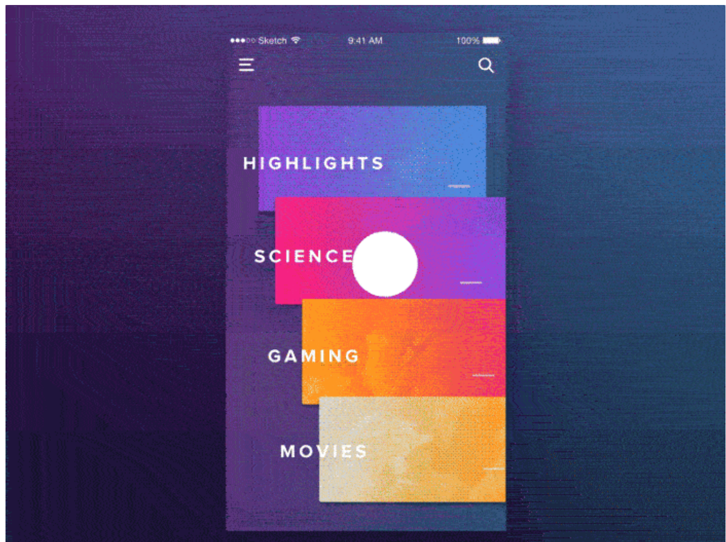
Think of these micro animation examples as a cause-and-effect occurrence. When a user clicks on or hovers over a specific element, the app is programmed to produce an exciting response that gives the user feedback. These can let the user know something is happening, encourage engagement throughout the app, signal a successful action or just make it easier and more enjoyable to use.

When UX research analysts are looking to set up micro interactions, they need to consider the standard psychology and mobile UX best practices. Thinking about user expectations will help you prepare users for a change in settings, guide them through a process, reward them for favorable actions and more. Utilizing mobile app micro animations in the correct way will help your brand stand out among the rest of the competition.

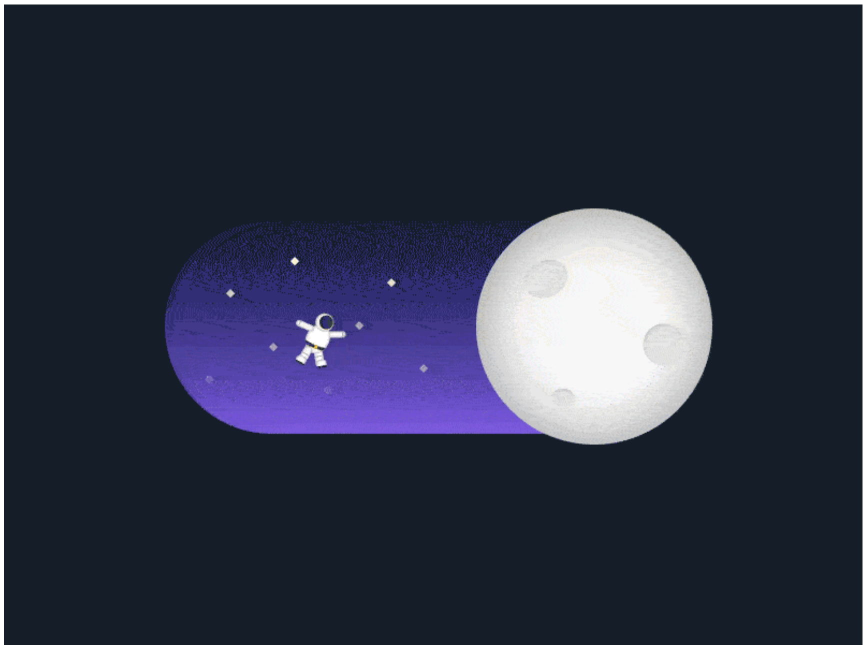
Now that we've covered some of the basics, we'll showcase a few micro animation examples and explain how these small design elements should be well thought out to provide the best mobile app user experience.

6 Types of Mobile App Micro Interactions

1. Micro Animation Examples that Signal a New Setting or Location



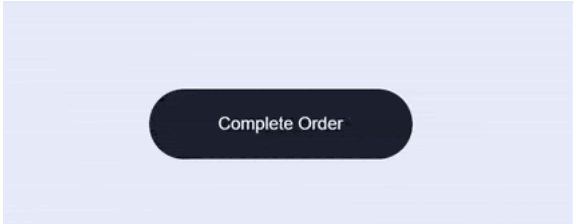
This sample uses many app UX best practices, like swiping to scroll and control your experience, the subtle guide at the top to help you retrace your path, as well as color coordination to tie everything together. We like the unique layout and how it gets creative to show the most prominent information.



This micro animation is an effective visual indicator of what this toggle button does, as it indicates a switch from light mode to dark mode. People understand visual cues more easily than written ones, so by making the toggle a visual cue, people can quickly understand its purpose while controlling their mobile app user experience. We've noticed dark mode is a growing web design trend, so we expect to see more of these clever "day and night" type micro animations. We recommend trying to find a way to tie it back to your brand or industry if possible.

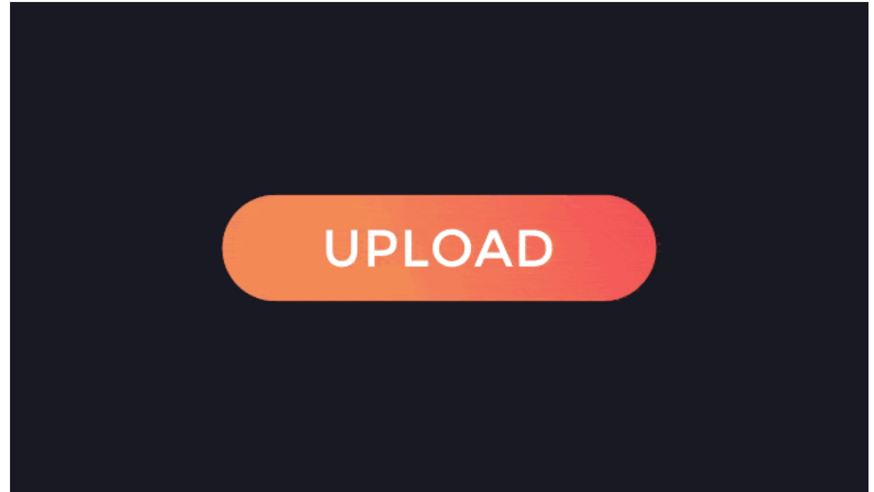
2. Micro Interactions in Apps Make Normal Tasks More Exciting

Apps are great for making quick and easy purchases, but that doesn't mean you should ignore the mobile app user experience. Get users talking about your app by adding some flair and real-life animation, like this example did for their ordering and delivery process.



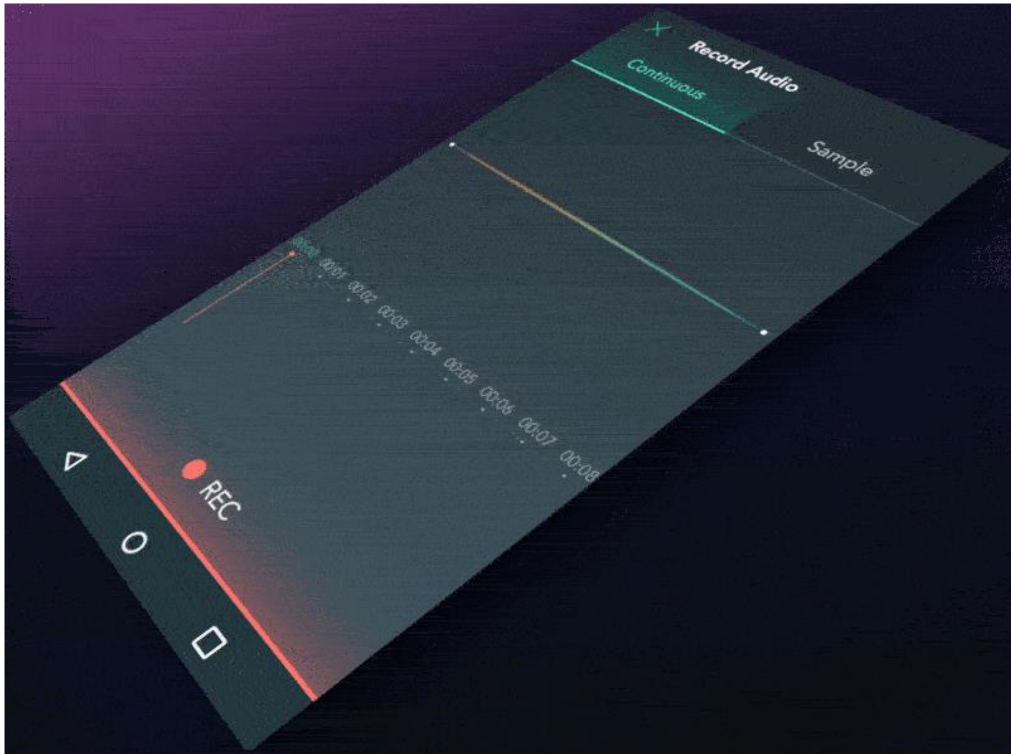
This micro animation example also used the checkout experience to get more creative. Once you've completed your shopping, a small delivery truck appears to load up your package and drives away, letting customers know the order they placed is on it's way!

3. Micro Animations Show the App is Processing Information



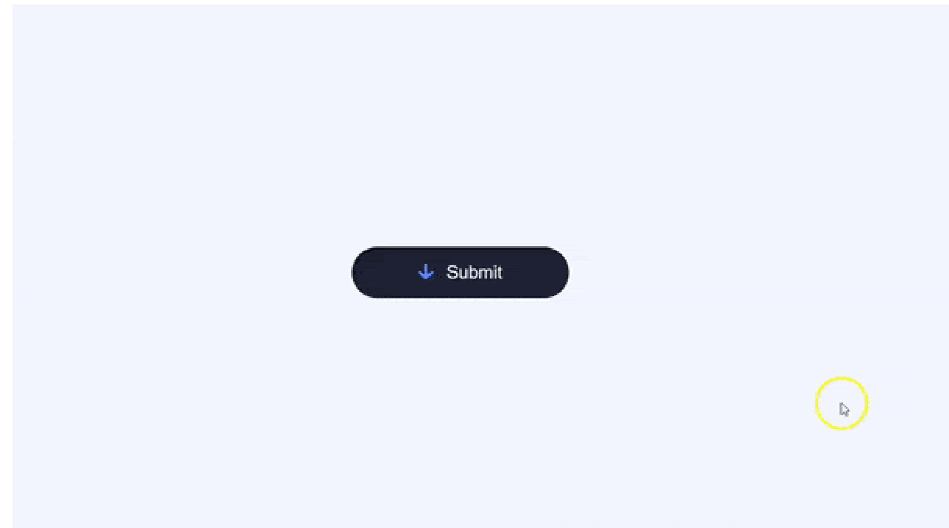
The act of a loading animation may have been one of the first types of micro interaction on apps or

websites. Loading animations are important to let users know that something is happening in the background. Without the animation to keep their interest, users can get bored or distracted, or worse, think there is a problem because nothing visibly happened when they clicked the button. Now, there are so many ways to make this process more exciting, so get creative with this mobile app micro interaction and think about how you can incorporate something exciting that fits your brand.



Another micro animation example that follows app UX best practices is voice search. Most devices that allow you to record your voice or music will show animated waves to show it is picking up the sound and processing it.

4. Celebratory Micro Animation Examples

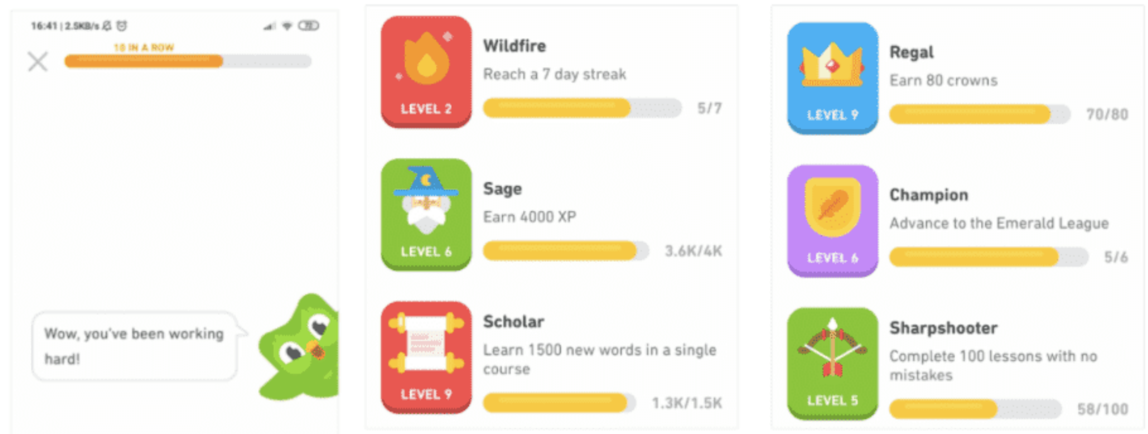


The goal of a mobile app is to help make your business easily accessible to users, and it should be simple for them to complete a desired task on your app. Once they do, why not make them feel a little better about a purchase or submission by celebrating them?

Mailchimp provides a great example of using a micro animation to celebrate a user's accomplishment. Each time you schedule out your latest and greatest newsletter, they confirm that your email is scheduled and say, "Rock On!"

5. Gamification Adds to the Mobile App User Experience

Some apps use various forms of gamification, or the use of elements you'd typically see when playing games into other online experiences, to engage users. Gamification can be achieved with polls, scores, questions, levels, badges and more to encourage the user to interact with others or feel accomplished. Adding micro animations to these fun experiences only adds to the exciting effect of completing a given task.



Duolingo has a great micro animation example that uses gamification best practices. Their mascot cheers users on while reminding them of their status at the top of the app. The badges and different levels users can earn and complete also adds to the game-like mobile app experience.

6. Humanizing Your Copy is a Mobile App Micro Interaction

A new example of micro interactions is also known as micro copy. People like to be talked to like they're, well, people instead of having to read formal jargon or scripts. This example kept things pretty professional, but the use of emojis really adds a little humanization to this micro interaction.

Taking the extra time to customize the verbiage on your app to reflect the voice and attitude of your brand helps you stand out! The key is to know your audience and be consistent - usually, a little sass or personality goes a long way to boost the mobile app user experience.

Keep App UX Best Practices in Mind When Planning Micro Animations

- **Don't let a micro animation distract the user.** Remember, the main goal of an app is to help your business earn more conversions or take designated actions. You don't want to plan an exciting interaction that will side track the visitor from what they're supposed to be doing on the app.
- **Keep it clean and simple.** Oftentimes, less is more when creating a mobile app or responsive design. Giving your users straightforward next steps or clear indicators of an action to complete will get the message across more than a cluttered design or excessive animation.
- **Mimic real movements for a natural flow.** There are certain expectations users have when they interact with apps. Swiping, scrolling, expanding and other common actions should follow their anticipated reactions. We recommend that your micro animations follow app UX best practices so you're not shocking the end user.
- **Ask yourself, will this add to the mobile app user experience?** Planning micro interactions in an app can be thrilling, but make sure you're not going overboard and putting them in for the sake of having more animation. Each mobile app micro interaction should serve a purpose. If the answer to this question is no, it is probably best not to include the triggered animation.

The Role of Animation in Enhancing User Experience

Animation has evolved from being a mere decorative element to a powerful tool in enhancing user experience (UX) across digital platforms. When used thoughtfully, animation can captivate users, provide context, guide interactions, and create delightful moments that leave a lasting impression. In this article, we will explore the significant role of animation in enhancing user experience and how it contributes to creating more engaging and user-centric digital products.

1. Visual Engagement and Attention

Animation has an innate ability to attract and hold users' attention. Dynamic and well-executed animations draw the eye and create a focal point on the screen. This engagement helps direct users' gaze toward important elements, such as calls-to-action (CTAs), key messages, or interactive features, ensuring that crucial information doesn't go unnoticed.

2. Context and Feedback

Animation can provide valuable context and feedback to users during interactions. For example, subtle hover animations can indicate clickable elements, while micro-interactions offer instant visual

feedback when an action is performed. These animations help users understand the interface's behavior and functionality, reducing uncertainty and enhancing usability.

3. Guiding User Flow

Animation is a powerful tool for guiding users through a sequence of actions or steps. Transitions, motion cues, and directional animations can help users understand the flow of a process, making complex interactions feel more intuitive. By visually guiding users, animation simplifies navigation and empowers users to complete tasks with confidence.

4. Storytelling and Narrative

Animation has the ability to convey a narrative or story in a visually engaging manner. Through animated sequences, users can be taken on a journey that explains a product's features, showcases a brand's values, or communicates complex concepts. Storytelling animations evoke emotions, making the user experience more memorable and impactful.

5. Visual Hierarchy and Emphasis

Motion can be used to establish visual hierarchy and emphasize content. For instance, a subtle animation can highlight a new feature or piece of content as users scroll down a page. By animating elements, designers can direct users' attention to what's most important and create a more dynamic and engaging layout.

6. Micro-Interactions and Engagement

Micro-interactions are subtle animations that respond to user actions, such as button clicks, form submissions, or toggles. These interactions add an element of delight to the user experience. Whether it's a playful loading animation or a satisfying "like" animation, these small details contribute to user engagement and make interactions feel more intuitive and enjoyable.

7. Reducing Cognitive Load

Complex processes or transitions can be made more comprehensible through animation. For example, an animated loading spinner or progress bar can give users a sense of how much time is left before an action is completed. Animation helps users understand what's happening behind the scenes, reducing cognitive load and preventing frustration.

8. Branding and Identity

Animation can reflect a brand's identity and personality. A unique animation style or signature motion can become synonymous with a brand, creating a memorable and consistent user experience. Animated brand elements contribute to a cohesive and recognizable digital presence.

9. Mobile and Responsive Design

Animation plays a crucial role in mobile and responsive design. Mobile devices have limited screen real estate, and animation can be used to reveal hidden menus, navigate between sections, and provide a seamless transition between different screen sizes. Animation ensures a smooth and enjoyable experience for users across devices.

10. Aesthetic Appeal and Delight

Beyond functionality, animation adds an element of aesthetic appeal and delight to the user experience. Well-crafted animations create a sense of elegance and sophistication, making the

interface more enjoyable to interact with. Users appreciate the effort put into creating visually pleasing and engaging designs.

Week 5- Building User Interface

Day 1: Implementing App Layouts and Views

Layout Structure

See Layout topic in IntelliJ Platform UI Guidelines for recommendations on arranging UI controls in dialogs.

Use `panel` to create UI:

```
panel {
  row {
    // child components
  }
}
```

Rows are created vertically from top to bottom, in the same order as lines of code that call `row`. Inside one row, you add components from left to right in the same order calls to factory method or `()` appear in each row. Every component is effectively placed in its own grid cell.

The label for the row can be specified as a parameter for the `row` method:

```
row("Parameters") { ... }
```

Rows can be nested. Components in a nested row block are considered to be subordinate to the containing row and are indented accordingly.

```
row {
  checkBox(...)
  row {
    textField(...) // indented relatively to the checkbox above
  }
}
```

To put multiple components in the same grid cell, wrap them in a `cell` method:

```
row {
  // These two components will occupy two columns in the grid
  label(...)
  textField(...)

  // These two components will be placed in the same grid cell
  cell {
    label(...)
    textField(...)
  }
```

```
}
```

To put a component on the right side of a grid row, use the `right` method:

```
row {
  rememberCheckBox()
  right {
    link("Forgot password")
  }
}
```

To visually debug layout, enable UI DSL Debug Mode from Internal Actions - UI Submenu.

Adding Components

There are two ways to add child components. The recommended way is to use factory methods `label`, `button`, `radioButton`, `link`, etc. It allows you to create consistent UI and reuse common patterns.

These methods also support property bindings, allowing you to automatically load the value displayed in the component from a property and to store it back. The easiest way to do that is to pass a reference to a Kotlin bound property:

```
checkBox("Show tabs in single row", uiSettings::scrollTabLayoutInEditor)
```

Note that the bound property reference syntax also can be used to reference Java fields, but not getter/setter pairs.

Alternatively, many factory methods support specifying a getter/setter pair for cases when a property mapping is more complicated:

```
checkBox(
  "Show file extensions in editor tabs",
  { !uiSettings.hideKnownExtensionInTabs },
  { uiSettings.hideKnownExtensionInTabs = !it })
```

If you want to add a component for which there are no factory methods, you can simply invoke an instance of your component, using the `()` overloaded operator:

```
val customComponent = MyCustomComponent()
panel {
  row { customComponent() }
}
```

Supported Components

Labels

Use the `label` method:

```
label("Sample text")
```

Checkboxes

See examples above.

Radio Buttons

Radio button groups are created using the `buttonGroup` block. There are two ways to use it. If the selected radio button corresponds to a specific value of a single property, pass the property binding to the `buttonGroup` method and the specific values to `radioButton` functions:

```
buttonGroup(mySettings::providerType) {
  row { radioButton("In native Keychain", ProviderType.KEYCHAIN) }
  row { radioButton("In KeePass", ProviderType.KEEPASS) }
}
```

If the selected radio button is controlled by multiple boolean properties, use `buttonGroup` with no binding and specify property bindings for all but one of the radio buttons:

```
buttonGroup {
  row { radioButton("The tab on the left") }
  row { radioButton("The tab on the right", uiSettings::activeRightEditorOnClose)
}
  row { radioButton("Most recently opened tab",
uiSettings::activeMruEditorOnClose) }
}
```

Text Fields

Use the `textField` method for a simple text field:

```
row("Username:") {
    textField(settings::userName)
}
```

For entering numbers, use `intTextField`:

```
intTextField(uiSettings::editorTabLimit, columns = 4, range = EDITOR_TABS_RANGE)
```

For password text fields, there is no factory function available, so you need to use `()`:

```
val passwordField = JPasswordField()
val panel = panel {
  // ...
  row { passwordField() }
}
```

To specify the size of a text field, either pass the `columns` parameter as shown in the `intTextField` example above, or use `growPolicy()`:

```
val userField = JTextField(credentials?.userName)
val panel = panel {
  row("Username:") { userField().growPolicy(GrowPolicy.SHORT_TEXT) }
}
```

Combo Boxes

Use the `comboBox` method with either a bound property, or a getter/setter pair:

```
comboBox(DefaultComboBoxModel<Int>(tabPlacements), uiSettings::editorTabPlacement)


comboBox<PgpKey>(
  pgpListModel,
  { getSelectedPgpKey() ?: pgpListModel.items.firstOrNull() },
  { mySettings.state.pgpKeyId = if (usePgpKey.isSelected) it?.keyId else null })
```

Spinners

Use the `spinner` method:

```
spinner(retypeOptions::retypeDelay, minValue = 0, maxValue = 5000, step = 50)
```

Link Label

Use the `link` method:

```
link("Forgot password?") {
  // handle click, e.g. showing dialog
}
```

To open URL in the browser, use `browserLink`:

```
browserLink("Open Homepage", "https://www.jetbrains.com")
```

Separators

Use the `titledRow` method and put the controls under the separator into the nested block:

```
titledRow("Appearance") {
  row { checkBox(...) }
}
```

Explanatory Text

Use the `comment` parameter:

```
checkBox(message("checkbox.smart.tab.reuse"),
       uiSettings::reuseNotModifiedTabs,
       comment = message("checkbox.smart.tab.reuse.inline.help"))
```

Integrating Panels with Property Bindings

A panel returned by the `panel` method is an instance of `DialogPanel`. This base class supports the standard `apply()`, `reset()`, and `isModified()` methods.

Dialogs

Reference: DialogWrapper

If you're using a `DialogPanel` as the main panel of a `DialogWrapper`, the `apply()` method will be automatically called when the dialog is closed using OK action. The other methods are unused in this case.

Use the `focused()` method to specify which control should be focused when the dialog is initialized:

```
return panel {
  row("Target class name:") {
    textField(::className).focused()
  }
}
```

Configurables

Reference: Settings Guide

If you're using the UI DSL to implement a `Configurable`, use `BoundConfigurable` as the base class. In this case, the `Configurable` methods will be automatically delegated to the panel.

Enabling and Disabling Controls

Use the `enableIf` method to bind the enabled state of a control to the values entered in other controls. The parameter of the method is a predicate.

```
checkBox("Show tabs in single row", uiSettings::scrollTabLayoutInEditor)
  .enableIf(myEditorTabPlacement.selectedValueIs(SwingConstants.TOP))
```

The available predicates are:

- `selected` to check the selected state of a checkbox or radio button
- `selectedValueIs` and `selectedValueMatches` to check the selected item in a combo box.

Predicates can be combined with `and` and `or` infix functions:

```
checkBox("Hide tabs if there is no space", uiSettings::hideTabsIfNeed)
  .enableIf(myEditorTabPlacement.selectedValueMatches { it != UISettings.TABS_NONE
} and
              myScrollTabLayoutInEditorCheckBox.selected)
```

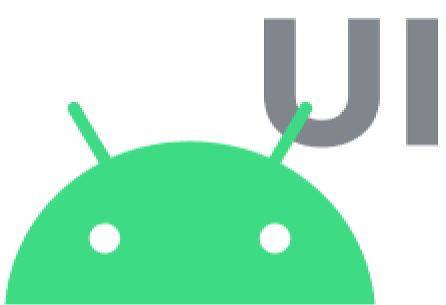
Layouts in Views

bookmark_border

Try the Compose way

Jetpack Compose is the recommended UI toolkit for Android. Learn how to work with layouts in Compose.

[Learn Compose layout basics →]



A layout defines the structure for a user interface in your app, such as in an activity. All elements in the layout are built using a hierarchy of View and ViewGroup objects. A View usually draws something the user can see and interact with. A ViewGroup is an invisible container that defines the layout structure for View and other ViewGroup objects, as shown in figure 1.

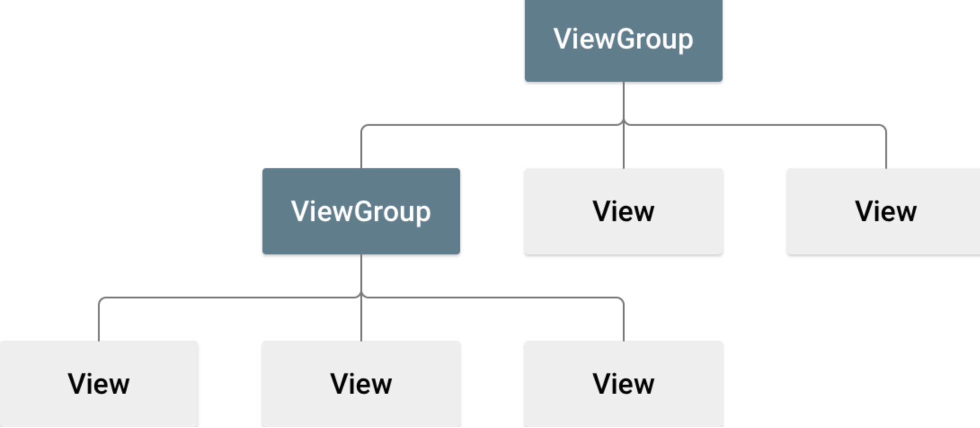


Figure 1. Illustration of a view hierarchy, which defines a UI layout.

View objects are often called *widgets* and can be one of many subclasses, such as Button or TextView. The ViewGroup objects are usually called *layouts* and can be one of many types that provide a different layout structure, such as LinearLayout or ConstraintLayout.

You can declare a layout in two ways:

- **Declare UI elements in XML.** Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses, such as those for widgets and layouts. You can also use Android Studio's Layout Editor to build your XML layout using a drag-and-drop interface.

- **Instantiate layout elements at runtime.** Your app can create View and ViewGroup objects and manipulate their properties programmatically.

Declaring your UI in XML lets you separate the presentation of your app from the code that controls its behavior. Using XML files also makes it easier to provide different layouts for different screen sizes and orientations. This is discussed further in Support different screen sizes.

The Android framework gives you the flexibility to use either or both of these methods to build your app's UI. For example, you can declare your app's default layouts in XML, and then modify the layout at runtime.

Tip: To debug your layout at runtime, use the Layout Inspector tool.

Write the XML

Using Android's XML vocabulary, you can quickly design UI layouts and the screen elements they contain, in the same way that you create web pages in HTML with a series of nested elements.

Each layout file must contain exactly one root element, which must be a View or ViewGroup object. After you define the root element, you can add additional layout objects or widgets as child elements to gradually build a View hierarchy that defines your layout. For example, here's an XML layout that uses a vertical LinearLayout to hold a TextView and a Button:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical" >
  <TextView android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a TextView" />
  <Button android:id="@+id/button"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="Hello, I am a Button" />
</LinearLayout>
```

After you declare your layout in XML, save the file with the .xml extension in your Android project's res/layout/ directory so it properly compiles.

For more information about the syntax for a layout XML file, see Layout resource.

Load the XML resource

When you compile your app, each XML layout file is compiled into a View resource. Load the layout resource in your app's Activity.onCreate() callback implementation. Do so by calling setContentView(), passing it the reference to your layout resource in the form: R.layout.*layout_file_name*. For example, if your XML layout is saved as main_layout.xml, load it for your Activity as follows:

KotlinJava

```
fun onCreate(savedInstanceState: Bundle) {
   super.onCreate(savedInstanceState)
   setContentView(R.layout.main_layout)
}
```

The Android framework calls the onCreate() callback method in your Activity when the Activity launches. For more information about activity lifecycles, see Introduction to activities.

Attributes

Every View and ViewGroup object supports its own variety of XML attributes. Some attributes are specific to a View object. For example, TextView supports the textSize attribute. However, these attributes are also inherited by any View objects that extend this class. Some are common to all View objects, because they are inherited from the root View class, like the id attribute. Other attributes are considered *layout parameters*, which are attributes that describe certain layout orientations of the View object, as defined by that object's parent ViewGroup object.

ID

Any View object can have an integer ID associated with it to uniquely identify the View within the tree. When the app is compiled, this ID is referenced as an integer, but the ID is typically assigned in the layout XML file as a string in the id attribute. This is an XML attribute common to all View objects, and it is defined by the View class. You use it very often. The syntax for an ID inside an XML tag is the following:

```
android:id="@+id/my_button"
```

The *at* symbol (@) at the beginning of the string indicates that the XML parser parses and expands the rest of the ID string and identifies it as an ID resource. The *plus* symbol (+) means this is a new resource name that must be created and added to your resources in the R.java file.

The Android framework offers many other ID resources. When referencing an Android resource ID, you don't need the *plus* symbol, but you must add the android package namespace as follows:

```
android:id="@android:id/empty"
```

The android package namespace indicates that you're referencing an ID from the android.R resources class, rather than the local resources class.

To create views and reference them from your app, you can use a common pattern as follows:

1. Define a view in the layout file and assign it a unique ID, as in the following example:

```
<Button android:id="@+id/my_button"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="@string/my_button_text"/>
```

2. Create an instance of the view object and capture it from the layout, typically in the onCreate() method, as shown in the following example:

KotlinJava

val myButton: Button = findViewById(R.id.my_button)

Defining IDs for view objects is important when creating a RelativeLayout. In a relative layout, sibling views can define their layout relative to another sibling view, which is referenced by the unique ID.

An ID doesn't need to be unique throughout the entire tree, but it must be unique within the part of the tree you search. It might often be the entire tree, so it's best to make it unique when possible.

Layout parameters

XML layout attributes named layout_*something* define layout parameters for the View that are appropriate for the ViewGroup it resides in.

Every ViewGroup class implements a nested class that extends ViewGroup.LayoutParams. This subclass contains property types that define the size and position of each child view, as appropriate for the view group. As shown in figure 2, the parent view group defines layout parameters for each child view, including the child view group.

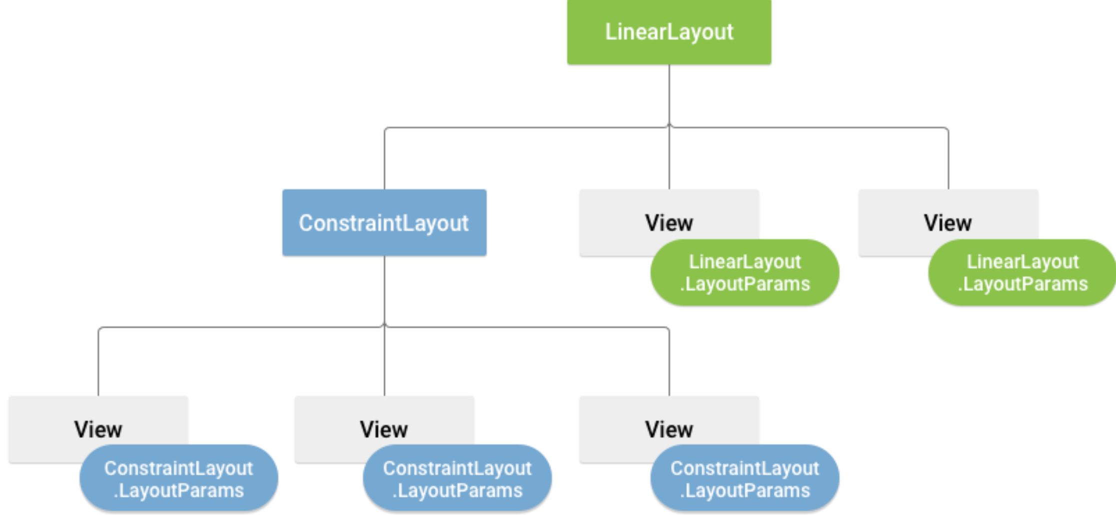


Figure 2. Visualization of a view hierarchy with layout parameters associated with each view.

Every LayoutParams subclass has its own syntax for setting values. Each child element must define a LayoutParams that is appropriate for its parent, though it might also define a different LayoutParams for its own children.

All view groups include a width and height, using layout_width and layout_height, and each view is required to define them. Many LayoutParams include optional margins and borders.

You can specify width and height with exact measurements, but you might not want to do this often. More often, you use one of these constants to set the width or height:

- wrap_content: tells your view to size itself to the dimensions required by its content.
- match_parent: tells your view to become as big as its parent view group allows.

In general, we don't recommend specifying a layout width and height using absolute units such as pixels. A better approach is using relative measurements, such as density-independent pixel units (dp), wrap_content, or match_parent, because it helps your app display properly across a variety of device screen sizes. The accepted measurement types are defined in Layout resource.

Layout position

A view has rectangular geometry. It has a location, expressed as a pair of *left* and *top* coordinates, and two dimensions, expressed as a width and height. The unit for location and dimensions is the pixel.

You can retrieve the location of a view by invoking the methods getLeft() and getTop(). The former returns the left (*x*) coordinate of the rectangle representing the view. The latter returns the top (*y*) coordinate of the rectangle representing the view. These methods return the location of the view relative to its parent. For example, when getLeft() returns 20, this means the view is located 20 pixels to the right of the left edge of its direct parent.

In addition, there are convenience methods to avoid unnecessary computations: namely getRight() and getBottom(). These methods return the coordinates of the right and bottom edges of the rectangle representing the view. For example, calling getRight() is similar to the following computation: getLeft() + getWidth().

Size, padding, and margins

The size of a view is expressed with a width and height. A view has two pairs of width and height values.

The first pair is known as *measured width* and *measured height*. These dimensions define how big a view wants to be within its parent. You can obtain the measured dimensions by calling getMeasuredWidth() and getMeasuredHeight().

The second pair is known as *width* and *height*, or sometimes *drawing width* and *drawing height*. These dimensions define the actual size of the view on screen, at drawing time and after layout. These values might, but don't have to, differ from the measured width and height. You can obtain the width and height by calling getWidth() and getHeight().

To measure its dimensions, a view takes into account its padding. The padding is expressed in pixels for the left, top, right and bottom parts of the view. You can use padding to offset the content of the view by a specific number of pixels. For instance, a left padding of two pushes the view's content two pixels to the right of the left edge. You can set padding using the setPadding(int, int, int, int) method and query it by calling getPaddingLeft(), getPaddingTop(), getPaddingRight(), and getPaddingBottom().

Although a view can define a padding, it doesn't support margins. However, view groups do support margins. See ViewGroup and ViewGroup.MarginLayoutParams for more information.

For more information about dimensions, see Dimension.

Besides setting margins and padding programmatically, you can also set them in your XML layouts, as shown in the following example:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical" >
   <TextView android:id="@+id/text"
        android:layout_width="wrap_content"
```

```
            android:layout_height="wrap_content"
            android:layout_margin="16dp"
            android:padding="8dp"
            android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="16dp"
            android:paddingBottom="4dp"
            android:paddingEnd="8dp"
            android:paddingStart="8dp"
            android:paddingTop="4dp"
            android:text="Hello, I am a Button" />
</LinearLayout>
```

The preceding example shows margin and padding being applied. The TextView has uniform margins and padding applied all around, and the Button shows how you can apply them independently to different edges.

Note: It is good practice to use **paddingStart**, **paddingEnd**, **layout_marginStart**, and **layout_marginEnd** instead of **paddingLeft**, **paddingRight**, **layout_marginLeft**, and **layout_marginRight**, as these behave better with both left-to-right and right-to-left language scripts.

Common layouts

Each subclass of the ViewGroup class provides a unique way to display the views you nest within it. The most flexible layout type, and the one that provides the best tools for keeping your layout hierarchy shallow, is ConstraintLayout.

The following are some of the common layout types built into the Android platform.

Note: Although you can nest one or more layouts within another layout to achieve your UI design, keep your layout hierarchy as shallow as possible. Your layout draws faster if it has fewer nested layouts. A wide view hierarchy is better than a deep view hierarchy.

Create a linear layout

Organizes its children into a single horizontal or vertical row and creates a scrollbar if the length of the window exceeds the length of the screen.

Build web apps in WebView

Displays web pages.

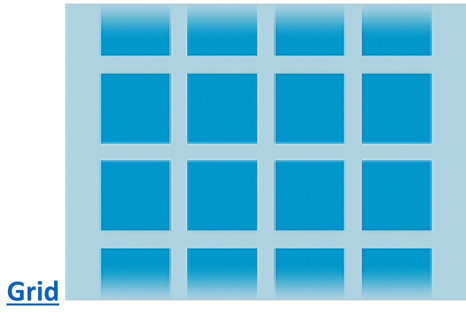
Build dynamic lists

When the content for your layout is dynamic or not pre-determined, you can use RecyclerView or a subclass of AdapterView. RecyclerView is generally the better option, because it uses memory more efficiently than AdapterView.

Common layouts possible with RecyclerView and AdapterView include the following:

List

Displays a scrolling single column list.

Grid 

Displays a scrolling grid of columns and rows.

RecyclerView offers more possibilities and the option to create a custom layout manager.

Fill an adapter view with data

You can populate an AdapterView such as ListView or GridView by binding the AdapterView instance to an Adapter, which retrieves data from an external source and creates a View that represents each data entry.

Android provides several subclasses of Adapter that are useful for retrieving different kinds of data and building views for an AdapterView. The two most common adapters are:

ArrayAdapter

Use this adapter when your data source is an array. By default, ArrayAdapter creates a view for each array item by calling toString() on each item and placing the contents in a TextView.

For example, if you have an array of strings you want to display in a ListView, initialize a new ArrayAdapter using a constructor to specify the layout for each string and the string array:

KotlinJava

```
val adapter = ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, myStringArray)
```

The arguments for this constructor are the following:

- Your app Context
- The layout that contains a TextView for each string in the array
- The string array

Then call setAdapter() on your ListView:

KotlinJava

```
val listView: ListView = findViewById(R.id.listview)
listView.adapter = adapter
```

To customize the appearance of each item you can override the toString() method for the objects in your array. Or, to create a view for each item that's something other than a TextView—for example, if you want an ImageView for each array item—extend the ArrayAdapter class and override getView() to return the type of view you want for each item.

SimpleCursorAdapter

Use this adapter when your data comes from a Cursor. When using SimpleCursorAdapter, specify a layout to use for each row in the Cursor and which columns in the Cursor you want inserted into the views of the layout you want. For example, if you want to create a list of people's names and phone numbers, you can perform a query that returns a Cursor containing a row for each person and columns for the names and numbers. You then create a string array specifying which columns from the Cursor you want in the layout for each result and an integer array specifying the corresponding views that each column need to be placed:

KotlinJava

```
val fromColumns = arrayOf(ContactsContract.Data.DISPLAY_NAME,
                ContactsContract.CommonDataKinds.Phone.NUMBER)
val toViews = intArrayOf(R.id.display_name, R.id.phone_number)
```

When you instantiate the SimpleCursorAdapter, pass the layout to use for each result, the Cursor containing the results, and these two arrays:

KotlinJava

```
val adapter = SimpleCursorAdapter(this,
        R.layout.person_name_and_number, cursor, fromColumns, toViews, 0)
val listView = getListView()
listView.adapter = adapter
```

The SimpleCursorAdapter then creates a view for each row in the Cursor using the provided layout by inserting each fromColumns item into the corresponding toViews view.

If during the course of your app's life you change the underlying data that is read by your adapter, call notifyDataSetChanged(). This notifies the attached view that the data has been changed and it refreshes itself.

Handle click events

You can respond to click events on each item in an AdapterView by implementing the AdapterView.OnItemClickListener interface. For example:

KotlinJava

```
listView.onItemClickListener = AdapterView.OnItemClickListener { parent, view, position, id ->
    // Do something in response to the click.
}
```

Android UI Layouts

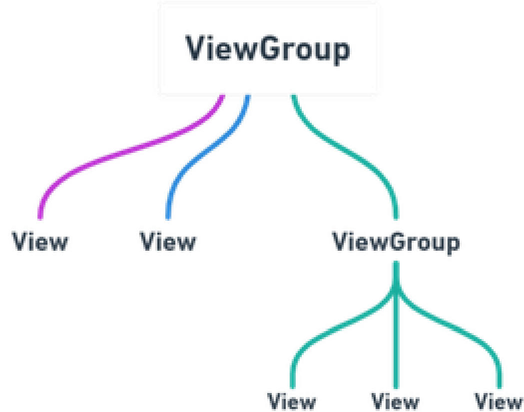
Android **Layout** is used to define the user interface that holds the UI controls or widgets that will appear on the screen of an android application or activity screen. Generally, every application is a combination of View and ViewGroup. As we know, an android application contains a large number of activities and we can say each activity is one page of the application. So, each activity contains multiple user interface components and those components are the instances of the View and ViewGroup. All the elements in a layout are built using a hierarchy of **View** and **ViewGroup** objects.

View

A **View** is defined as the user interface which is used to create interactive UI components such as TextView, ImageView, EditText, RadioButton, etc., and is responsible for event handling and drawing. They are Generally Called Widgets.

ViewGroup

A **ViewGroup** act as a base class for layouts and layouts parameters that hold other Views or ViewGroups and to define the layout properties. They are Generally Called layouts.



ViewGroup

The Android framework will allow us to use UI elements or widgets in two ways:

- Use UI elements in the XML file
- Create elements in the Kotlin file dynamically

Types of Android Layout

- **Android Linear Layout:** LinearLayout is a ViewGroup subclass, used to provide child View elements one by one either in a particular direction either horizontally or vertically based on the orientation property.
- **Android Relative Layout:** RelativeLayout is a ViewGroup subclass, used to specify the position of child View elements relative to each other like (A to the right of B) or relative to the parent (fix to the top of the parent).

- **Android Constraint Layout:** ConstraintLayout is a ViewGroup subclass, used to specify the position of layout constraints for every child View relative to other views present. A ConstraintLayout is similar to a RelativeLayout, but having more power.
- **Android Frame Layout:** FrameLayout is a ViewGroup subclass, used to specify the position of View elements it contains on the top of each other to display only a single View inside the FrameLayout.
- **Android Table Layout:** TableLayout is a ViewGroup subclass, used to display the child View elements in rows and columns.
- **Android Web View:** WebView is a browser that is used to display the web pages in our activity layout.
- **Android ListView:** ListView is a ViewGroup, used to display scrollable lists of items in a single column.
- **Android Grid View:** GridView is a ViewGroup that is used to display a scrollable list of items in a grid view of rows and columns.

Use UI Elements in the XML file

Here, we can create a layout similar to web pages. The XML layout file contains at least one root element in which additional layout elements or widgets can be added to build a View hierarchy. Following is the example:

- XML

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout

  xmlns:android="http:// schemas.android.com/apk/res/android"

  xmlns:tools="http:// schemas.android.com/tools"

  android:orientation="vertical"

  android:layout_width="match_parent"

  android:layout_height="match_parent"

  tools:context=".MainActivity">


  <!--EditText with id editText-->

  <EditText

    android:id="@+id/editText"

    android:layout_width="match_parent"

    android:layout_height="wrap_content"
```

```
        android:layout_margin="16dp"
        android:hint="Input"
        android:inputType="text"/>

    <!--Button with id showInput-->
    <Button
        android:id="@+id/showInput"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:text="show"
        android:backgroundTint="@color/colorPrimary"
        android:textColor="@android:color/white"/>

</LinearLayout>
```

Load XML Layout File and its elements from an Activity

When we have created the layout, we need to load the XML layout resource from our activity **onCreate()** callback method and access the UI element from the XML using **findViewById**.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    // finding the button
    val showButton = findViewById<Button>(R.id.showInput)

    // finding the edit text
    val editText = findViewById<EditText>(R.id.editText)
```

Here, we can observe the above code and find out that we are calling our layout using the **setContentView** method in the form of **R.layout.activity_main**. Generally, during the launch of our activity, the *onCreate()* callback method will be called by the android framework to get the required layout for an activity.

Create elements in the Kotlin file Dynamically

We can create or instantiate UI elements or widgets during runtime by using the custom View and ViewGroup objects programmatically in the Kotlin file. Below is an example of creating a layout using LinearLayout to hold an EditText and a Button in an activity programmatically.

- Kotlin

```
import android.os.Bundle
import android.widget.Button
import android.widget.EditText
import android.widget.LinearLayout
import android.widget.Toast
import android.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // create the button
        val showButton = Button(this)
        showButton.setText("Submit")

        // create the editText
        val editText = EditText(this)

        val linearLayout = findViewById<LinearLayout>(R.id.l_layout)
        linearLayout.addView(editText)
        linearLayout.addView(showButton)

        // Setting On Click Listener
        showButton.setOnClickListener
```

```
        {
            // Getting the user input
            val text = editText.text

            // Showing the user input
            Toast.makeText(this, text, Toast.LENGTH_SHORT).show()
        }
    }
}
```

Different Attribute of the Layouts

XML attributes	Description
android:id	Used to specify the id of the view.
android:layout_width	Used to declare the width of View and ViewGroup elements in the layout.
android:layout_height	Used to declare the height of View and ViewGroup elements in the layout.
android:layout_marginLeft	Used to declare the extra space used on the left side of View and ViewGroup elements.
android:layout_marginRight	Used to declare the extra space used on the right side of View and ViewGroup elements.
android:layout_marginTop	Used to declare the extra space used in the top side of View and ViewGroup elements.
android:layout_marginBottom	Used to declare the extra space used in the bottom side of View and ViewGroup elements.

XML attributes	Description
android:layout_gravity	Used to define how child Views are positioned in the layout.

What are the main parts of a layout?

Layout usually has three main parts through which layout features are organised- the **introduction**, **main body**, and **conclusions.**

The **introduction** is the first point of contact the reader has with the content. It must be engaging and outline the main points of the article.

The **main body,** on the other hand, is where the most important parts of the text can be found. It must be clear and concise, corresponding to what the introduction said it would discuss.

Finally, the **conclusion** brings all the strands of your argument together and summarises it. Sometimes the article presents a new idea in relation to the rest of the essay that can expand on the possibilities of the main body of the text. It must be able to stand on its own; one should be able to understand what the article is trying to convey by just reading the conclusion.

Thanks to this ad, StudySmarter remains free:

What are the layout feature examples?

Apart from having engaging content, it is important to include other features and elements of a 'layout' to attract readers to your work. In this section, we will discuss the different layout features with layout features examples, and how you can use them in your writing.

Thanks to this ad, StudySmarter remains free:

1. Headings and subheadings

In most cases, a heading is the first point of contact the readers have with the written material and its layout features. It must be able to capture their attention. Headings and subheadings are short statements describing what the section is about. Generally, headings are in a larger font compared to subheadings, like they are in this article. While headings are the building blocks, the subheadings are the roadmap! They help keep both the reader and the writer on track.

Thanks to this ad, StudySmarter remains free:

2. Second paragraph

It is important to note that generally, a paragraph revolves around **one** main idea, which is introduced at the beginning of the paragraph. Using paragraphs effectively helps get an idea across to your readers. Much like the introduction, main body, and conclusion, paragraphs help structure writing effectively.

Paragraphs open with a **topic sentence which** introduces the main point. A topic sentence should be short and precise. The next section, or what is also known as the **supporting sentences**, is the main body of the paragraph. This part provides evidence and supports the topic sentence.

Finally, you end the paragraph with a **concluding sentence** that links back to the main idea, whilst providing a transition into the next paragraph (or the next idea, if it applies).

3. Spacing

Howdoyoufeelreadingthis?

Reading the sentence above wasn't the best experience, was it? That is the power of spacing! Using white space helps the reader process information and gives the eyes a break, preventing strain.

4. Font Colour

Font colour is the colour that is used to write text. It is important because it can convey meaning. Colours can convey emotion, helping readers recall information and maintain attention. When used effectively, colours can make text aesthetically pleasing and help make large blocks of text easier to digest.

Tip: Notice how colour is used in this article and how different colours and headings break up the large chunks of text

5. Font size

The text of font size is the size of the characters that are displayed or printed. It is essential to get this right as it will impact the readability of the text. The wrong text size will make it difficult to read and disengage the reader, creating a negative reading experience. Using the correct text size is important to make the reading experience as easy as possible for the reader so they remain interested in the text. The most common font sizes are 11 and 12.

6. Font Type

Font is the graphical representation of text. There are many fonts such as Arial, Times New Roman, Georgia, etc. It is important to use the right font as it makes it easier for the reader to read, and this will allow you to gain more readers. Certain fonts are suitable in different contexts. In an academic essay, one would expect you to use a basic font, such as Times New Roman, whereas a more interesting and eye-catching font would be required in an advertisement.

7. Textboxes

Text boxes are a tool that allows you to add text or pictures anywhere in the document. This allows you to draw attention to a specific area of the file, by being able to design a particular text box for instance (by way of colours, borders, etc.).

We are able to design textboxes and add text to a document - Microsoft Word screenshot

The above picture shows you an example of a text box in Microsoft Word and the various options on the left-hand side you can use to design it. This can be decided based on the theme and the other colours you are using in the articles. An advantage of using a text box is that you can easily move around text when required.

Tip: Have a look at the article on font colour if you are confused about how many colours to use in your work.

8. Graphs, Charts, Tables, Pictures

Graphs, charts, tables, and pictures are important tools that help organise and visualise the content. Depending on the kind of text, they each serve a different purpose. For instance, if it was a book aimed at kids aged 3-4, the pictures would be very different from that in a business presentation, as can be seen below.

A children's book usually has big, bright pictures that entertain and educate the child - Pixabay

A business presentation will have specific images, graphs, charts, and tables - Pixabay

9. Captions

A caption is a small statement contextualising the image, helping the reader to understand the relevance of the image in relation to the text. Let's look at an example of this layout feature:

Captions help the reader to interpret and understand the image - Pixabay

What do you think of the image above? One could come to multiple conclusions:

- He could be a new guy in the city, lost? Mourning? Waiting for somebody?

There are multiple ways to interpret an image, right? Image captions help the reader understand the image in relation to the text. It is important to always have captions for your images to help the reader focus on the text.

Why is layout important?

The layout and layout features are important to structure content and written information. How layout features are organised affects how the content is interpreted by the reader. A poor layout can lead to disengaged and uninterested readers. On the other hand, a good layout allows for easy navigation of the text. In addition, a well-organised layout can help maintain the reader's attention, which is what you want as a writer!

Layout - key takeaways

- Layout describes the way something is designed or arranged on the page.
- The three main parts of any written text are: introduction, main body, and conclusion.
- Some of the features to keep in mind while designing the layout of your written material are - headings, subheadings, paragraphs, spacing, colour, text size, font, text boxes, graphs, pictures, and captions.
- A poorly designed layout makes a text more difficult to read and causes readers to disengage from the text.
- A good layout is beneficial to both the reader and the writer as it allows for easy navigation and maintains the reader.

How layouts affect the appearance of your app

- Consider the hierarchy—what grabs attention first? Is it the bold title, the eye-catching image, or that quirky icon in the corner? It's like setting up the stage for a play; each element has a role to play.
- Spacing is crucial too. Don't overcrowd your space; let things breathe. White space is like a visual exhale—it makes everything more digestible.

- Consistency is your design BFF. If your buttons have commitment issues and keep changing shapes and colors, users might get confused. Keep it steady, like a reliable friend who always has your back.
- Responsive design is like having a wardrobe that fits you at any size. Your app should look good on a phone, a tablet, or a giant desktop screen. No one likes a pixelated surprise.
- Remember, the goal is not just to make your app look pretty but to create a smooth, intuitive experience. It's like arranging furniture in a way that feels natural—you don't want users tripping over the coffee table trying to find the checkout button.

User Interface Elements

When designing your interface, try to be consistent and predictable in your choice of interface elements. Whether they are aware of it or not, users have become familiar with elements acting in a certain way, so choosing to adopt those elements when appropriate will help with task completion, efficiency, and satisfaction.

Interface elements include but are not limited to:

- *Input Controls*: checkboxes, radio buttons, dropdown lists, list boxes, buttons, toggles, text fields, date field
- *Navigational Components:* breadcrumb, slider, search field, pagination, slider, tags, icons
- *Informational Components:* tooltips, icons, progress bar, notifications, message boxes, modal windows
- **Containers**: accordion

Input Controls

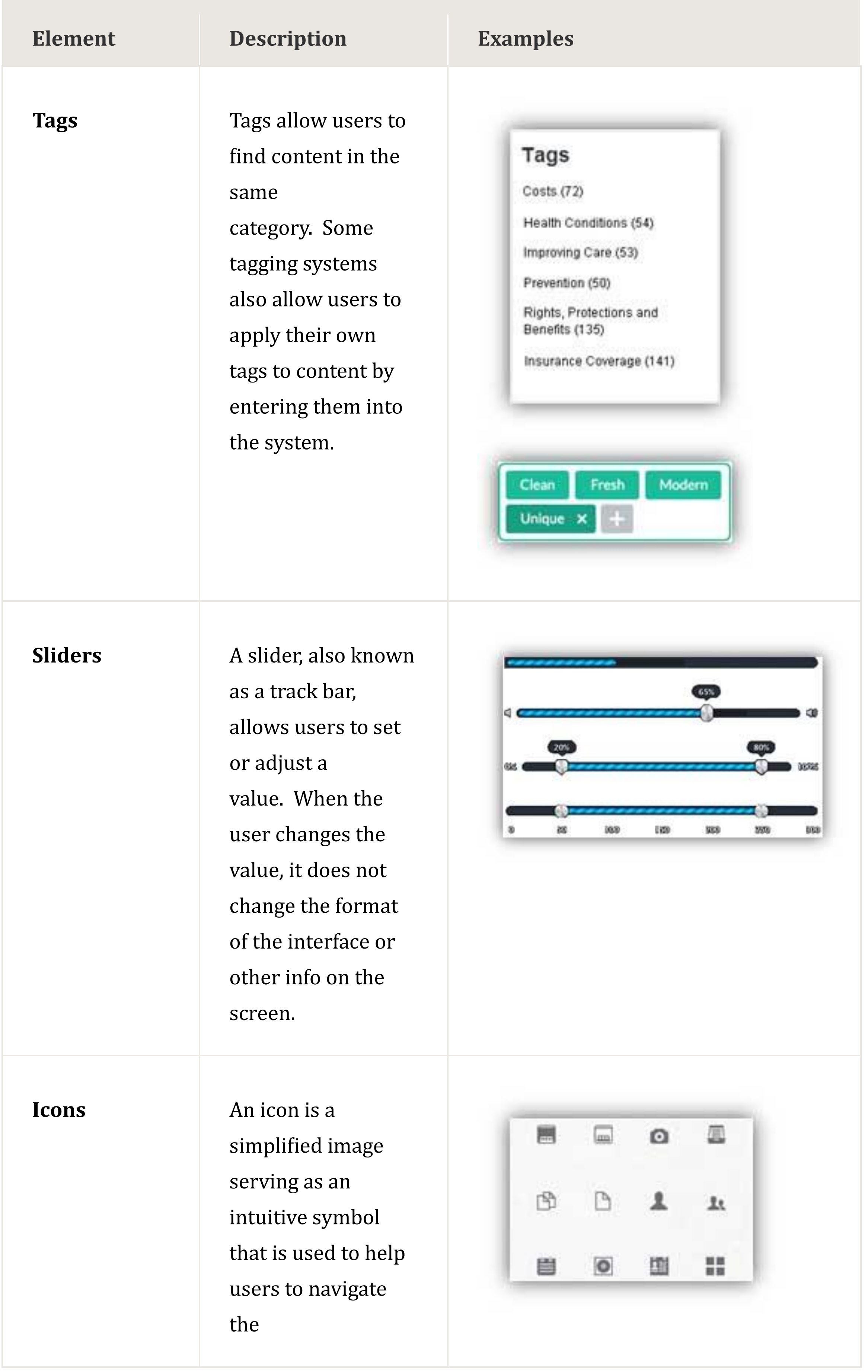
Element	Description	Examples
Checkboxes	Checkboxes allow the user to select one or more options from a set. It is usually best to present checkboxes in a vertical list. More than one column is acceptable as well if the list is long enough that it might require scrolling or if comparison of terms might be necessary.	☐ NonFederal (99) ☐ Federal (57)

Element	Description	Examples
Radio buttons	Radio buttons are used to allow users to select one item at a time.	Yes No
Dropdown lists	Dropdown lists allow users to select one item at a time, similarly to radio buttons, but are more compact allowing you to save space. Consider adding text to the field, such as 'Select one' to help the user recognize the necessary action.	Find your state or... Go
List boxes	List boxes, like checkboxes, allow users to select a multiple items at a time,but are more compact and can support a longer list of options if needed.	List boxes Are like dropdowns But they let You make Multiple selections Like checkboxes do
Buttons	A button indicates an action upon touch and is typically labeled using text, an icon, or both.	Send Post Tweet
Dropdown Button	The dropdown button consists of a button that when clicked displays a drop-down list of mutually exclusive items.	General Settings Your Profile Sign Out

Element	Description	Examples
Toggles	A toggle button allows the user to change a setting between two states. They are most effective when the on/off states are visually distinct.	ON OFF
Text fields	Text fields allow users to enter text. It can allow either a single line or multiple lines of text.	Text input fields let you input text
Date and time pickers	A date picker allows users to select a date and/or time. By using the picker, the information is consistently formatted and input into the system.	SEP 18TH 2010 · 22 JUN · Sep, 2010 · JUN 22 · 22 JUN

Navigational Components

Element	Description	Examples
Search Field	A search box allows users to enter a keyword or phrase (query) and submit it to search the index with the intention of getting back the most relevant results. Typically search fields are single-line text boxes and are often accompanied by a search button.	
Breadcrumb	Breadcrumbs allow users to identify their current location within the system by providing a clickable trail of proceeding pages to navigate by.	Home > Folder Index Page > Page You're On
Pagination	Pagination divides content up between pages, and allows users to skip between pages or go in order through the content.	

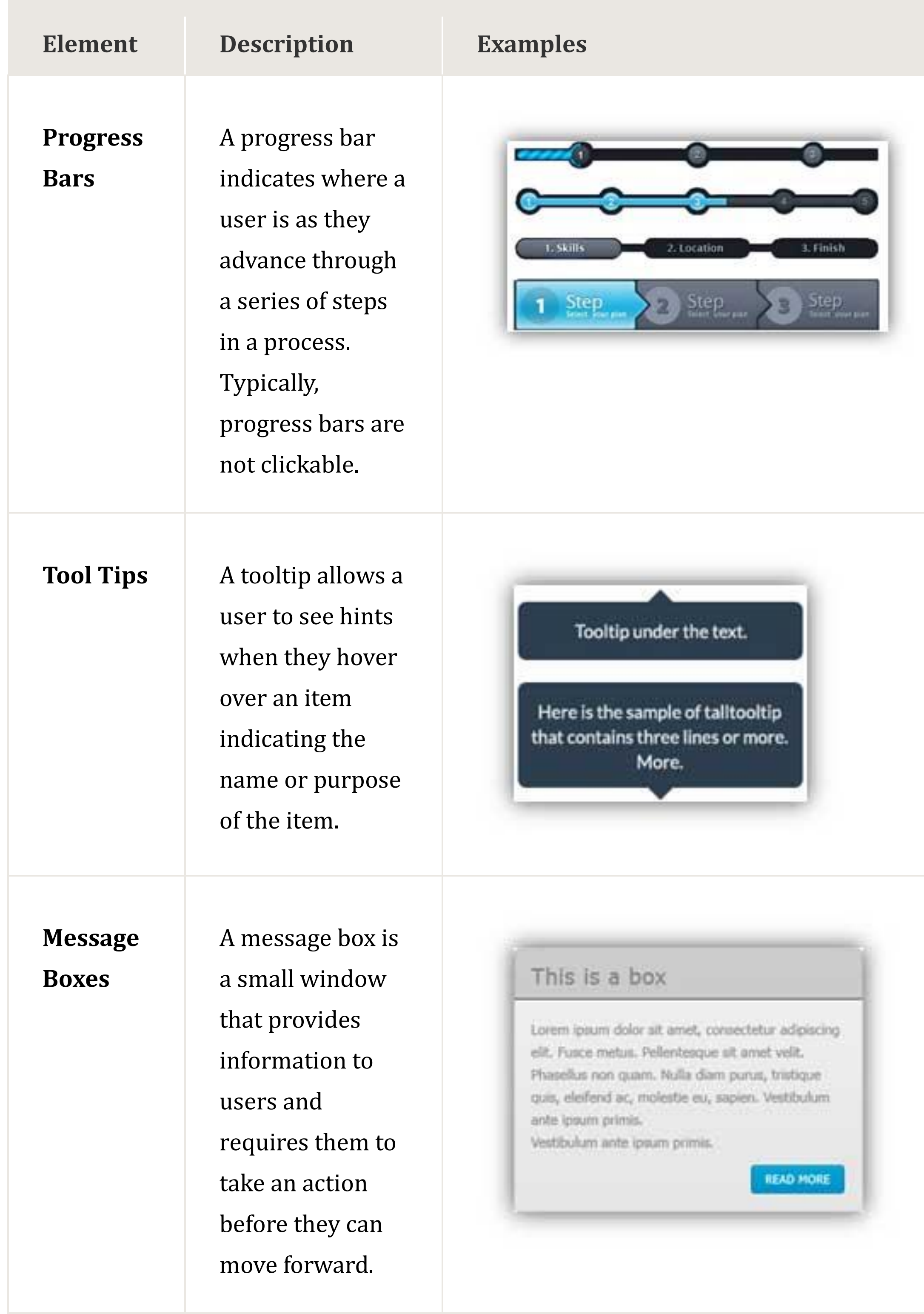


Element	Description	Examples
Tags	Tags allow users to find content in the same category. Some tagging systems also allow users to apply their own tags to content by entering them into the system.	
Sliders	A slider, also known as a track bar, allows users to set or adjust a value. When the user changes the value, it does not change the format of the interface or other info on the screen.	
Icons	An icon is a simplified image serving as an intuitive symbol that is used to help users to navigate the	

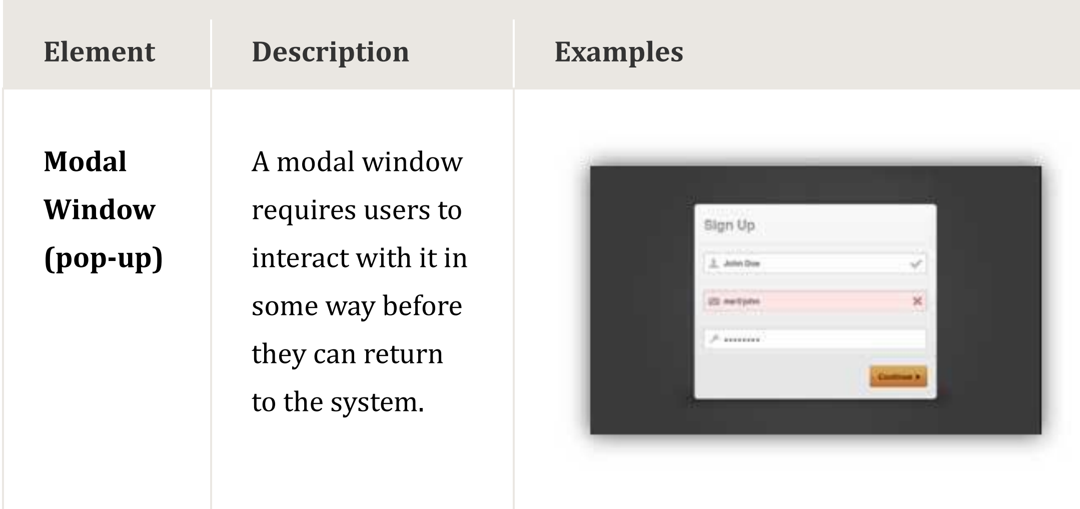
Element	Description	Examples
	system. Typically, icons are hyperlinked.	
Image Carousel	Image carousels allow users to browse through a set of items and make a selection of one if they so choose. Typically, the images are hyperlinked.	

Information Components

Element	Description	Examples
Notifications	A notification is an update message that announces something new for the user to see. Notifications are typically used to indicate items such as, the successful completion of a task, or an error or warning message.	



Element	Description	Examples
Progress Bars	A progress bar indicates where a user is as they advance through a series of steps in a process. Typically, progress bars are not clickable.	
Tool Tips	A tooltip allows a user to see hints when they hover over an item indicating the name or purpose of the item.	
Message Boxes	A message box is a small window that provides information to users and requires them to take an action before they can move forward.	

Element	Description	Examples
Modal Window (pop-up)	A modal window requires users to interact with it in some way before they can return to the system.	

Containers

Element	Description	Examples
Accordion	An accordion is a vertically stacked list of items that utilizes show/ hide functionality. When a label is clicked, it expands the section showing the content within. There can have one or more items showing at a time and may have default states that reveal one or more sections without the user clicking	

How to Position Views Properly in Layouts

Have you ever wondered what the difference between layout_gravity and gravity is? Or been frustrated because you included android:layout_gravity=”center” in your view but nothing was centering? In this tutorial I’ll try to address those issues and focus on the LinearLayout class.

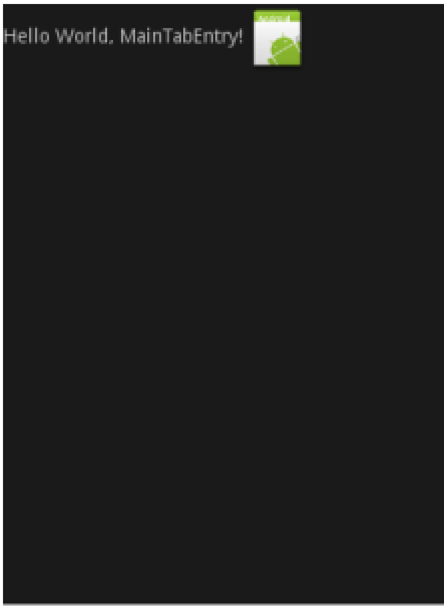
First off, the difference between android:gravity and android:layout_gravity is that **android:gravity** positions the contents of that view (i.e. what’s inside the view), whereas **android:layout_gravity** positions the view with respect to its parent (i.e. what the view is contained in). If this isn’t clear to you now, then maybe after these examples it will become clear.

Say you have an example Layout like this:

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3       android:orientation="vertical"
4       android:layout_width="fill_parent"
```

```
5          android:layout_height="fill_parent" >
6          <LinearLayout
7             android:orientation="horizontal"
8             android:layout_width="wrap_content"
9             android:layout_height="wrap_content"
10            android:gravity="center" >
11         <TextView
12            android:layout_width="wrap_content"
13         android:layout_height="wrap_content"
14            android:text="@string/hello" />
15         <ImageView
16            android:layout_width="wrap_content"
17            android:layout_height="wrap_content"
18            android:src="@drawable/icon" />
19         </LinearLayout>
20      </LinearLayout>
```

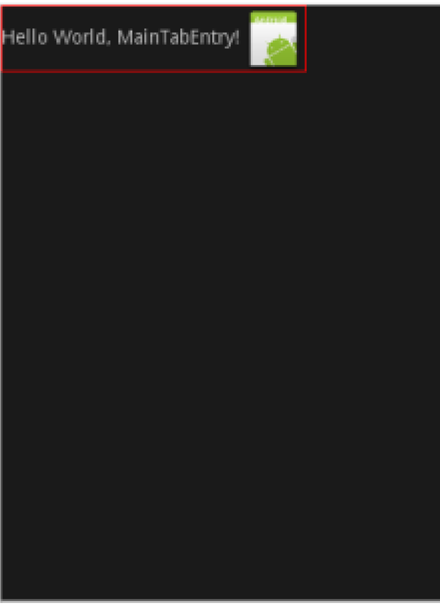
Which looks like:



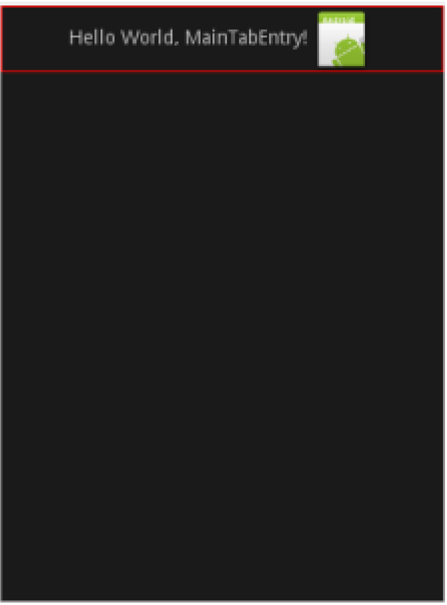
And you're wondering…

"Why is this not centering? I have two views, and I'm using gravity on the parent view so that its children (i.e. the TextView and ImageView) will be centered inside."

Well check out what happens when you click the LinearLayout in the XML preview:



See what happened? Because your parent LinearLayout has width “wrap_content”, it’s essentially “hugging” the two views inside and so they have no space to center themselves in. What’s the fix? Change the width to “fill_parent” and see what happens:

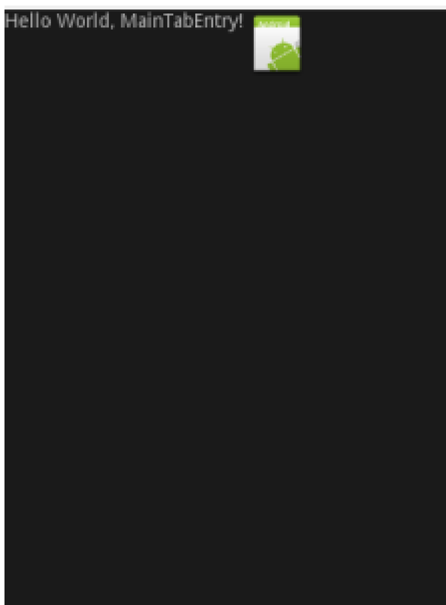


So now that we let the parent view expand to fill the entire width of the screen, its contents have room to actually center themselves and so we’re done.

Now, let’s say that we want the TextView to stay to the left and the ImageView to go to the right. The example XML is:

```
1       <?xml version="1.0" encoding="utf-8"?>

2       <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

3           android:orientation="vertical"

4           android:layout_width="fill_parent"
```

```
5          android:layout_height="fill_parent" >
6          <LinearLayout
7             android:orientation="horizontal"
8             android:layout_width="fill_parent"
9             android:layout_height="wrap_content" >
10         <TextView
11            android:layout_width="wrap_content"
12            android:layout_height="wrap_content"
13            android:layout_gravity="left"
14            android:text="@string/hello" />
15         <ImageView
16            android:layout_width="wrap_content"
17            android:layout_height="wrap_content"
18            android:layout_gravity="right"
19            android:src="@drawable/icon" />
20         </LinearLayout>
21      </LinearLayout>
```



And you're thinking…

"Okay so my my parent fills the entire width and I tell the TextView to position itself with respect to its parent on the left, and I tell the ImageView to position itself with respect to its parent on the right, but how come it doesn't work?"

Well my honest answer is: I don't know.

[UPDATE]
It's recently been brought to my attention (thanks to reader Sam Duke) that the "layout_gravity" property can only be used *orthogonally* with the orientation of the LinearLayout.

In other words, if you have a *horizontal* LinearLayout, then by construction, each inside child view can only have layout_gravity *top*, *bottom*, and *center*. The intuition behind this is that the LinearLayout is already told to place each child view horizontally adjacent to each other (left to right), and so it only allows vertical specification for the layout_gravity of each child. Vice versa for a vertical LinearLayout.

For a more in-depth explanation, I direct you to:

http://sandipchitale.blogspot.co.uk/2010/05/linearlayout-gravity-and-layoutgravity.html
[END UPDATE]

I'm not sure why this aspect of the layout is so unintuitive but there's a fix! Simply wrap the desired view that you want to shift to the right in another LinearLayout who has width "fill_parent" and tell it to position its children to the right! To be precise, the new XML example looks like:

```
1       <?xml version="1.0" encoding="utf-8"?>
2       <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3          android:orientation="vertical"
4          android:layout_width="fill_parent"
5          android:layout_height="fill_parent" >
6          <LinearLayout
7             android:orientation="horizontal"
8             android:layout_width="fill_parent"
9             android:layout_height="wrap_content" >
10         <TextView
11            android:layout_width="wrap_content"
12            android:layout_height="wrap_content"
13            android:layout_gravity="left"
14            android:text="@string/hello" />
15         <!-- NEW LAYOUT -->
16         <LinearLayout
17            android:orientation="horizontal"
18            android:layout_width="fill_parent"
```

```
19          android:layout_height="wrap_content"
20          android:gravity="right" >
21        <ImageView
22          android:layout_width="wrap_content"
23          android:layout_height="wrap_content"
24          android:src="@drawable/icon" />
25        </LinearLayout>
26        </LinearLayout>
27      </LinearLayout>
```

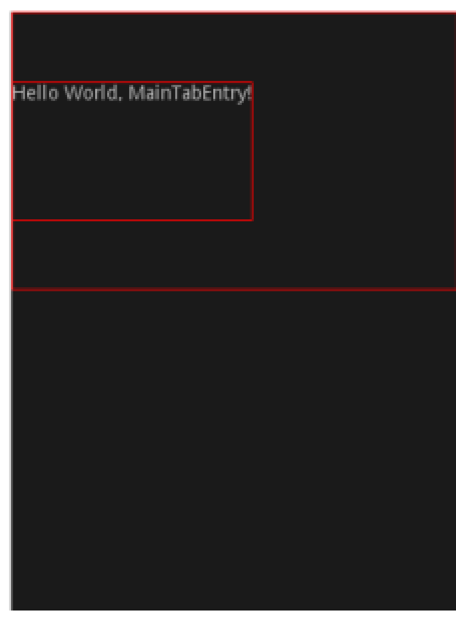
So notice how we removed the layout_gravity="right" from the ImageView and instead set android:gravity="right" in the new LinearLayout. There's no good reason for this besides that in my experience android:gravity has just worked better than layout_gravity (sometimes the behavior works as intended, and other times even I'm stumped for long periods of time). But any who, now we are left with:

And so we see the new layout in action as it fills the rest of the parent width and allows the ImageView to have the space it needs to position itself to the right!

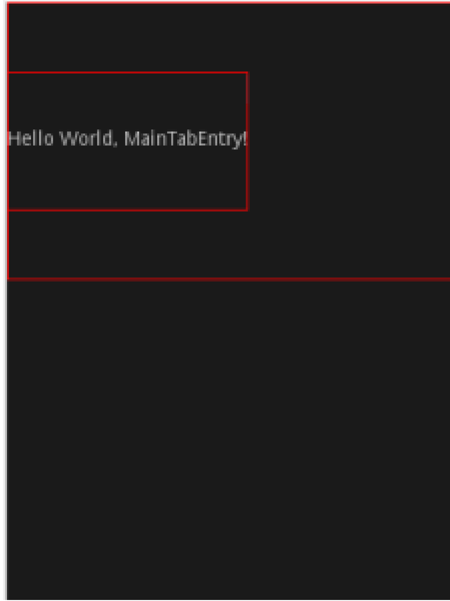
And finally, I'll conclude with two screen shots which illustrate the difference between layout_gravity and gravity:

```
1       <?xml version="1.0" encoding="utf-8"?>
2       <!-- LAYOUT_GRAVITY EXAMPLE -->
3       <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
4         android:orientation="vertical"
5         android:layout_width="fill_parent"
```

```
6           android:layout_height="fill_parent" >
7           <LinearLayout
8              android:orientation="horizontal"
9              android:layout_width="fill_parent"
10             android:layout_height="200px" >
11          <TextView
12             android:layout_width="wrap_content"
13             android:layout_height="100px"
14             android:gravity="center"
15             android:text="@string/hello" />
16          </LinearLayout>
17       </LinearLayout>
```



Again, why it does not center horizontally escapes me, but from the image you can see that at least the TextView is centering itself vertically within its parent LinearLayout. Notice however that the text in the TextView is clipped to the top, and now notice what happens when we include android:gravity=”center” in the TextView from our previous example:



Required Field validation

In the following example, we have an EditText with an id etName in the XML layout file. In the Kotlin code, we initialize the etName EditText using findViewById and set an OnEditorActionListener to trigger the validation when the Done action is performed. In the validateName() function, we retrieve the entered text, trim any leading or trailing spaces, and check if it is empty using TextUtils.isEmpty(). If the name is empty, we set an error message on the EditText using setError() to indicate that the field is required.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    etName = findViewById(R.id.etName)

    // Perform validation when a specific action is triggered
    etName.setOnEditorActionListener { _, actionId, _ ->
        if (actionId == EditorInfo.IME_ACTION_DONE) {
            validateName()
            return@setOnEditorActionListener false
        }
        false
    }
}
```

```
private fun validateName() {
    val name = etName.text.toString().trim()
    if (TextUtils.isEmpty(name)) {
        etName.error = "Name is required"
    } else {
        // Name is not empty, perform further actions
    }
}
```

Length validation

To demonstrate length validation, we have an EditText with an id etPassword in the XML layout file. In the Kotlin code, we set a maximum length of 8 characters for the password input using an InputFilter. We also add a TextWatcher to the EditText to perform the validation. In the afterTextChanged() method, we check if the length of the password is less than the specified maximum length. If it is, we set an error message using setError() on the EditText to indicate the required length.

```
class MainActivity : AppCompatActivity() {
    private lateinit var etPassword: EditText

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        etPassword = findViewById(R.id.etPassword)

        // Set maximum length for the password input
        val maxLength = 8
        val inputFilterArray = arrayOfNulls<InputFilter>(1)
        inputFilterArray[0] = InputFilter.LengthFilter(maxLength)
        etPassword.filters = inputFilterArray

        // Perform validation using a TextWatcher
        etPassword.addTextChangedListener(object : TextWatcher {
```

```
            override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, after: Int) {
                // Not needed for this example
            }

            override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int) {
                // Not needed for this example
            }

            override fun afterTextChanged(s: Editable?) {
                val password = s.toString().trim()
                if (password.length < maxLength) {
                    etPassword.error = "Password must be at least $maxLength characters"
                } else {
                    etPassword.error = null
                }
            }
        })
    }
}
```

Patterns matcher

In Android development, pattern matchers refer to classes or functions that allow you to search for specific patterns or structures within text or input data. They are used to identify and extract relevant information based on predefined patterns or rules.

Pattern matchers in Android often leverage regular expressions (regex) to define the patterns they are looking for. Regular expressions provide a flexible and powerful way to express complex search patterns.

One common use case for pattern matchers in Android is input validation. For example, if you have a form where users are required to enter an email address, you can use a pattern matcher to ensure that the input follows a valid email format. The pattern matcher will compare the entered text against a predefined regex pattern for email addresses and determine whether it matches or not.

Another example is parsing data from text or HTML. Suppose you have a string containing a list of URLs, and you want to extract the domain names from each URL. You can use a pattern matcher with a regex pattern to identify and extract the domain names from the string.

Pattern matchers in Android can be implemented using various APIs and classes provided by the Android framework, such as the Pattern class in the java.util.regex package. These classes offer methods for compiling regex patterns, performing matching operations, and extracting matched substrings.

By utilizing pattern matchers in Android, developers can effectively search for, validate, and extract data based on specific patterns or structures, enhancing the functionality and usability of their applications.

Some example patterns are the DOMAIN_NAME, EMAIL_ADDRESS, PHONE or WEB_URLmatches. We will show examples for the E-Mail and URL pattern matchers.

Email validation

To demonstrate an Email validation, we have put an EditText with an id etEmail in the XML layout file. The inputType attribute is set to "textEmailAddress" to enable the email address input type and provide email-specific keyboard features.

In the Kotlin code, we add a TextWatcher to the EditText to perform the validation. In the afterTextChanged() method, we retrieve the entered text, trim any leading or trailing spaces, and check if it is a valid email address using the isValidEmail() function.
The isValidEmail() function uses the Patterns.EMAIL_ADDRESS constant and its associated regular expression to validate the email format.

If the entered value is not a valid email address or is empty, we set an error message using setError() on the EditText. Otherwise, if the input is a valid email address, we clear the error.

```
<EditText

   android:id="@+id/etEmail"

   android:layout_width="match_parent"

   android:layout_height="wrap_content"

   android:hint="Enter your email"

   android:inputType="textEmailAddress"

   android:imeOptions="actionDone"

   />
```

```
import android.os.Bundle

import android.text.Editable

import android.text.TextUtils

import android.text.TextWatcher

import android.util.Patterns

import android.widget.EditText

import androidx.appcompat.app.AppCompatActivity
```

```
class MainActivity : AppCompatActivity() {
    private lateinit var etEmail: EditText

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        etEmail = findViewById(R.id.etEmail)

        // Perform validation using a TextWatcher
        etEmail.addTextChangedListener(object : TextWatcher {
            override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, after: Int) {
                // Not needed for this example
            }

            override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int) {
                // Not needed for this example
            }

            override fun afterTextChanged(s: Editable?) {
                val email = s.toString().trim()
                if (TextUtils.isEmpty(email) || !isValidEmail(email)) {
                    etEmail.error = "Please enter a valid email address"
                } else {
                    etEmail.error = null
                }
            }
        })
    }
```

```
    private fun isValidEmail(email: String): Boolean {
        return Patterns.EMAIL_ADDRESS.matcher(email).matches()
    }
}
```

URL validation

To test a URL for valid format we can use once again a certain patterns matcher.

In this example, we have an EditText with an id etUrl in the XML layout file.
The inputType attribute is set to “textUri” to allow the user to enter a URI or URL.

In the Kotlin code, we add a TextWatcher to the EditText to perform the validation. In the afterTextChanged() method, we retrieve the entered text, trim any leading or trailing spaces, and check if it is a valid URL using the isValidUrl() function. The isValidUrl() function uses the Patterns.WEB_URL constant and its associated regular expression to validate the URL format.

If the entered value is not a valid URL or is empty, we set an error message using setError() on the EditText. Otherwise, if the input is a valid URL, we clear the error.

```
<EditText
    android:id="@+id/etUrl"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Enter a URL"
    android:inputType="textUri"
    android:imeOptions="actionDone"
    />
```

```
import android.os.Bundle
import android.text.Editable
import android.text.TextUtils
import android.text.TextWatcher
import android.util.Patterns
import android.widget.EditText
import androidx.appcompat.app.AppCompatActivity


class MainActivity : AppCompatActivity() {
    private lateinit var etUrl: EditText
```

```
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        etUrl = findViewById(R.id.etUrl)

        // Perform validation using a TextWatcher
        etUrl.addTextChangedListener(object : TextWatcher {
            override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, after: Int) {
                // Not needed for this example
            }

            override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int) {
                // Not needed for this example
            }

            override fun afterTextChanged(s: Editable?) {
                val url = s.toString().trim()
                if (TextUtils.isEmpty(url) || !isValidUrl(url)) {
                    etUrl.error = "Please enter a valid URL"
                } else {
                    etUrl.error = null
                }
            }
        })
    }

    private fun isValidUrl(url: String): Boolean {
        return Patterns.WEB_URL.matcher(url).matches()
    }
}
```

Date validation

For date validations we will put the inputType attribute to “date” to enable a date picker dialog for selecting a date.

In the Kotlin code, we add a TextWatcher to the EditText to perform the validation. In the afterTextChanged() method, we retrieve the entered text, trim any leading or trailing spaces, and check if it is a valid date using the isValidDate() function. The isValidDate() function attempts to parse the date string using a SimpleDateFormat with the format “yyyy-MM-dd”. If the parsing is successful and the date is valid, it returns true; otherwise, it returns false.

If the entered value is not a valid date or is empty, we set an error message using setError() on the EditText. Otherwise, if the input is a valid date, we clear the error.

```
<EditText
    android:id="@+id/etDate"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Enter a date (YYYY-MM-DD)"
    android:inputType="date"
    android:imeOptions="actionDone"
    />
import android.os.Bundle
import android.text.Editable
import android.text.TextUtils
import android.text.TextWatcher
import android.widget.EditText
import androidx.appcompat.app.AppCompatActivity
import java.text.ParseException
import java.text.SimpleDateFormat
import java.util.*

class MainActivity : AppCompatActivity() {
    private lateinit var etDate: EditText
    private val dateFormat = SimpleDateFormat("yyyy-MM-dd", Locale.getDefault())
```

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    etDate = findViewById(R.id.etDate)

    // Perform validation using a TextWatcher
    etDate.addTextChangedListener(object : TextWatcher {
        override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, after: Int) {
            // Not needed for this example
        }

        override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int) {
            // Not needed for this example
        }

        override fun afterTextChanged(s: Editable?) {
            val date = s.toString().trim()
            if (TextUtils.isEmpty(date) || !isValidDate(date)) {
                etDate.error = "Please enter a valid date (YYYY-MM-DD)"
            } else {
                etDate.error = null
            }
        }
    })
}

private fun isValidDate(date: String): Boolean {
    return try {
        dateFormat.isLenient = false
        dateFormat.parse(date)
```

```
        true
    } catch (e: ParseException) {
        false
    }
  }
}
```

Custom regex validation

Regex, short for Regular Expression, is a powerful and versatile tool used for pattern matching and manipulating text. It is a sequence of characters that defines a search pattern, allowing you to find, match, and manipulate strings based on specific criteria.

Think of regex as a language for describing patterns within text. Just like you might use keywords or phrases to search for information in a document, regex uses a combination of characters, symbols, and special syntax to define patterns. These patterns can be as simple as finding a specific word or as complex as extracting data from structured text.

For example, let's say you have a long list of email addresses, and you want to extract all the addresses that belong to a specific domain, such as "@example.com". Using regex, you can construct a pattern that matches this specific format and retrieve the desired email addresses, ignoring the rest.

Another example could be validating a phone number format. If you want to check if a phone number is in the format "XXX-XXX-XXXX" (where X represents a digit), you can use regex to define this pattern and verify if a given phone number matches it.

Regex is widely used in various domains, including text processing, data extraction, data validation, and programming. It provides a concise and flexible way to search, replace, and manipulate textual data based on specific patterns, making it an invaluable tool for tasks involving text analysis and manipulation.

Numeric input validation

```
class MainActivity : AppCompatActivity() {
  private lateinit var etNumber: EditText


  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)


    etNumber = findViewById(R.id.etNumber)


    // Perform validation using a TextWatcher
```

```
        etNumber.addTextChangedListener(object : TextWatcher {
            override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, after: Int) {
                // Not needed for this example
            }

            override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int) {
                // Not needed for this example
            }

            override fun afterTextChanged(s: Editable?) {
                val number = s.toString().trim()
                if (number.isNotEmpty() && !isNumeric(number)) {
                    etNumber.error = "Please enter a valid number"
                } else {
                    etNumber.error = null
                }
            }
        })
    }

    private fun isNumeric(str: String): Boolean {
        return str.matches("-?\\d+(\\.\\d+)?".toRegex())
    }
}
```

Password validation

In this example, we have an EditText with an id etPassword in the XML layout file.
The inputType attribute is set to “textPassword” to hide the password characters as they are entered.

In the Kotlin code, we add a TextWatcher to the EditText to perform the validation. In the afterTextChanged() method, we retrieve the entered text, trim any leading or trailing spaces, and check if it is a valid password using the isValidPassword() function.
The isValidPassword() function uses a regular expression (passwordRegex) to define the password validation rules.

In the provided example, the password must be at least 8 characters long and contain at least one uppercase letter, one lowercase letter, and one digit. You can modify the regular expression in the isValidPassword() function to match your specific password validation rules.

If the entered value is not a valid password or is empty, we set an error message using setError() on the EditText. Otherwise, if the input is a valid password, we clear the error.

```
<EditText
    android:id="@+id/etPassword"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Enter your password"
    android:inputType="textPassword"
    android:imeOptions="actionDone"
    />
import android.os.Bundle
import android.text.Editable
import android.text.TextUtils
import android.text.TextWatcher
import android.widget.EditText
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {
    private lateinit var etPassword: EditText

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        etPassword = findViewById(R.id.etPassword)

        // Perform validation using a TextWatcher
        etPassword.addTextChangedListener(object : TextWatcher {
            override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, after: Int) {
```

```
            // Not needed for this example
        }

        override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int) {
            // Not needed for this example
        }

        override fun afterTextChanged(s: Editable?) {
            val password = s.toString().trim()
            if (TextUtils.isEmpty(password) || !isValidPassword(password)) {
                etPassword.error = "Please enter a valid password"
            } else {
                etPassword.error = null
            }
        }
    })
}

private fun isValidPassword(password: String): Boolean {
    // Add your password validation rules here
    // Example: At least 8 characters with at least one uppercase letter, one lowercase letter, and one digit
    val passwordRegex = "^(?=.*[a-z])(?=.*[A-Z])(?=.*\\d).{8,}\$"
    return password.matches(passwordRegex.toRegex())
}
}
```

Range validation

In this example, we have an EditText with an id etNumber in the XML layout file. The inputType attribute is set to “number” to allow the user to enter numeric values.

In the Kotlin code, we add a TextWatcher to the EditText to perform the validation. In the afterTextChanged() method, we retrieve the entered text, trim any leading or trailing spaces, and check if it is a valid number within the specified range using the isValidNumber() function.

The isValidNumber() function attempts to convert the entered text to an integer value. If the conversion is successful, it checks if the number falls within the specified range (between minValue and maxValue). If the number is within the range, it returns true; otherwise, it returns false. If the conversion fails (due to non-numeric input), it also returns false.

If the entered value is not a valid number or is empty, we set an error message using setError() on the EditText to indicate that a number within the specified range is required. If the input is a valid number within the range, we clear the error.

```
<EditText

    android:id="@+id/etNumber"

    android:layout_width="match_parent"

    android:layout_height="wrap_content"

    android:hint="Enter a number (between 1 and 100)"

    android:inputType="number"

    android:imeOptions="actionDone"

    />

import android.os.Bundle

import android.text.Editable

import android.text.TextUtils

import android.text.TextWatcher

import android.widget.EditText

import androidx.appcompat.app.AppCompatActivity


class MainActivity : AppCompatActivity() {

    private lateinit var etNumber: EditText

    private val minValue = 1

    private val maxValue = 100


    override fun onCreate(savedInstanceState: Bundle?) {

        super.onCreate(savedInstanceState)

        setContentView(R.layout.activity_main)


        etNumber = findViewById(R.id.etNumber)
```

```
        // Perform validation using a TextWatcher
        etNumber.addTextChangedListener(object : TextWatcher {
            override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, after: Int) {
                // Not needed for this example
            }

            override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int) {
                // Not needed for this example
            }

            override fun afterTextChanged(s: Editable?) {
                val number = s.toString().trim()
                if (TextUtils.isEmpty(number) || !isValidNumber(number)) {
                    etNumber.error = "Please enter a number between $minValue and $maxValue"
                } else {
                    etNumber.error = null
                }
            }
        })
    }

    private fun isValidNumber(number: String): Boolean {
        return try {
            val value = number.toInt()
            value in minValue..maxValue
        } catch (e: NumberFormatException) {
            false
        }
    }
}
```

Phone number validation

In this example, we have an EditText with an id etPhoneNumber in the XML layout file.
The inputType attribute is set to “phone” to allow the user to enter a phone number. This input type helps in displaying a numeric keyboard with additional symbols commonly used in phone numbers.

In the Kotlin code, we add a TextWatcher to the EditText to perform the validation. In the afterTextChanged() method, we retrieve the entered text, trim any leading or trailing spaces, and check if it is a valid phone number using the isValidPhoneNumber() function.

The isValidPhoneNumber() function uses PhoneNumberUtils.isGlobalPhoneNumber() to validate the phone number. It checks if the provided string is a valid global phone number according to the rules of the country or region the user is currently in.

If the entered value is not a valid phone number or is empty, we set an error message using setError() on the EditText to indicate that a valid phone number is required. If the input is a valid phone number, we clear the error.

```
<EditText

   android:id="@+id/etPhoneNumber"

   android:layout_width="match_parent"

   android:layout_height="wrap_content"

   android:hint="Enter a phone number"

   android:inputType="phone"

   android:imeOptions="actionDone"

   />
```

```
import android.os.Bundle

import android.telephony.PhoneNumberUtils

import android.text.Editable

import android.text.TextUtils

import android.text.TextWatcher

import android.widget.EditText

import androidx.appcompat.app.AppCompatActivity


class MainActivity : AppCompatActivity() {

   private lateinit var etPhoneNumber: EditText


   override fun onCreate(savedInstanceState: Bundle?) {
```

```
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        etPhoneNumber = findViewById(R.id.etPhoneNumber)

        // Perform validation using a TextWatcher
        etPhoneNumber.addTextChangedListener(object : TextWatcher {
            override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, after: Int) {
                // Not needed for this example
            }

            override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int) {
                // Not needed for this example
            }

            override fun afterTextChanged(s: Editable?) {
                val phoneNumber = s.toString().trim()
                if (TextUtils.isEmpty(phoneNumber) || !isValidPhoneNumber(phoneNumber)) {
                    etPhoneNumber.error = "Please enter a valid phone number"
                } else {
                    etPhoneNumber.error = null
                }
            }
        })
    }

    private fun isValidPhoneNumber(phoneNumber: String): Boolean {
        return PhoneNumberUtils.isGlobalPhoneNumber(phoneNumber)
    }
}
```

Note that you could also use a patterns matcher to validate the phone number. The code would look then something like the following.

```
import android.os.Bundle
import android.text.Editable
import android.text.TextUtils
import android.text.TextWatcher
import android.util.Patterns
import android.widget.EditText
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {
    private lateinit var etPhoneNumber: EditText

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        etPhoneNumber = findViewById(R.id.etPhoneNumber)

        // Perform validation using a TextWatcher
        etPhoneNumber.addTextChangedListener(object : TextWatcher {
            override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, after: Int) {
                // Not needed for this example
            }

            override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int) {
                // Not needed for this example
            }

            override fun afterTextChanged(s: Editable?) {
                val phoneNumber = s.toString().trim()
```

```
            if (TextUtils.isEmpty(phoneNumber) || !isValidPhoneNumber(phoneNumber)) {
                etPhoneNumber.error = "Please enter a valid phone number"
            } else {
                etPhoneNumber.error = null
            }
        }
    })
}

private fun isValidPhoneNumber(phoneNumber: String): Boolean {
    return Patterns.PHONE.matcher(phoneNumber).matches()
}
}
```

EasyValidation Library

The EasyValidation Library is a powerful and intuitive validation framework for Kotlin developers. It simplifies the process of validating user input by providing a fluent and declarative API. With EasyValidation, you can easily define validation rules and apply them to various data types, such as strings, numbers, and dates. It supports a wide range of validation rules, including required fields, minimum and maximum length, numeric ranges, regular expressions, and more. Here's an example code snippet in Kotlin that demonstrates how to use the EasyValidation Library:

```
// Import the EasyValidation library
import com.easyvalidation.EasyValidation

// Create a validation instance
val validation = EasyValidation()

// Define validation rules for a string input
validation.addRule("username", "Please enter a valid username")
    .required()
    .minLength(6)
    .maxLength(20)

// Validate the input
```

```
val input = "john_doe"
val validationResult = validation.validate(input)


// Check if the input is valid
if (validationResult.isValid()) {
    println("Input is valid")
} else {
    val errorMessages = validationResult.getErrorMessages()
    println("Input is invalid. Errors: $errorMessages")
}
```

In this example, we create a validation instance using EasyValidation(). We then define validation rules for a string input called “username”. The rules specify that the username is required, must have a minimum length of 6 characters, and a maximum length of 20 characters. We validate the input using validation.validate(input), which returns a ValidationResult object. We can check if the input is valid using validationResult.isValid(). If it’s not valid, we can retrieve the error messages using validationResult.getErrorMessages().

Bottom Navigation Bar in Android Using Kotlin

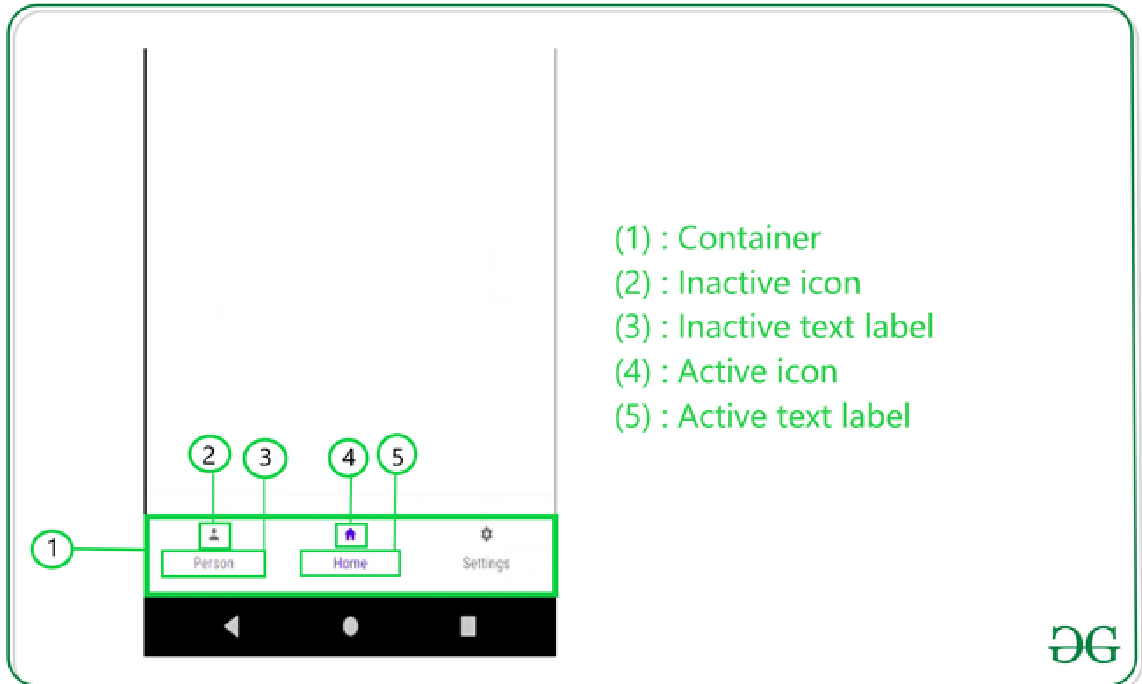
We all have come across apps that have a Bottom Navigation Bar. Some popular examples include Instagram, WhatsApp, etc. In this article, we will learn about bottom navigation using Fragments. We will learn it by making a project in android studio. Here we will be using Kotlin as the language for development.

To implement it in Java please refer to: [Bottom Navigation Bar in Android using Java]

Why do we need a Bottom Navigation Bar?

- It allows the user to navigate to different activities/fragments easily.
- It makes the user aware of the different screens available in the app.
- The user is able to check which screen are they on at the moment.

The following is an anatomy diagram for the Bottom Navigation Bar:



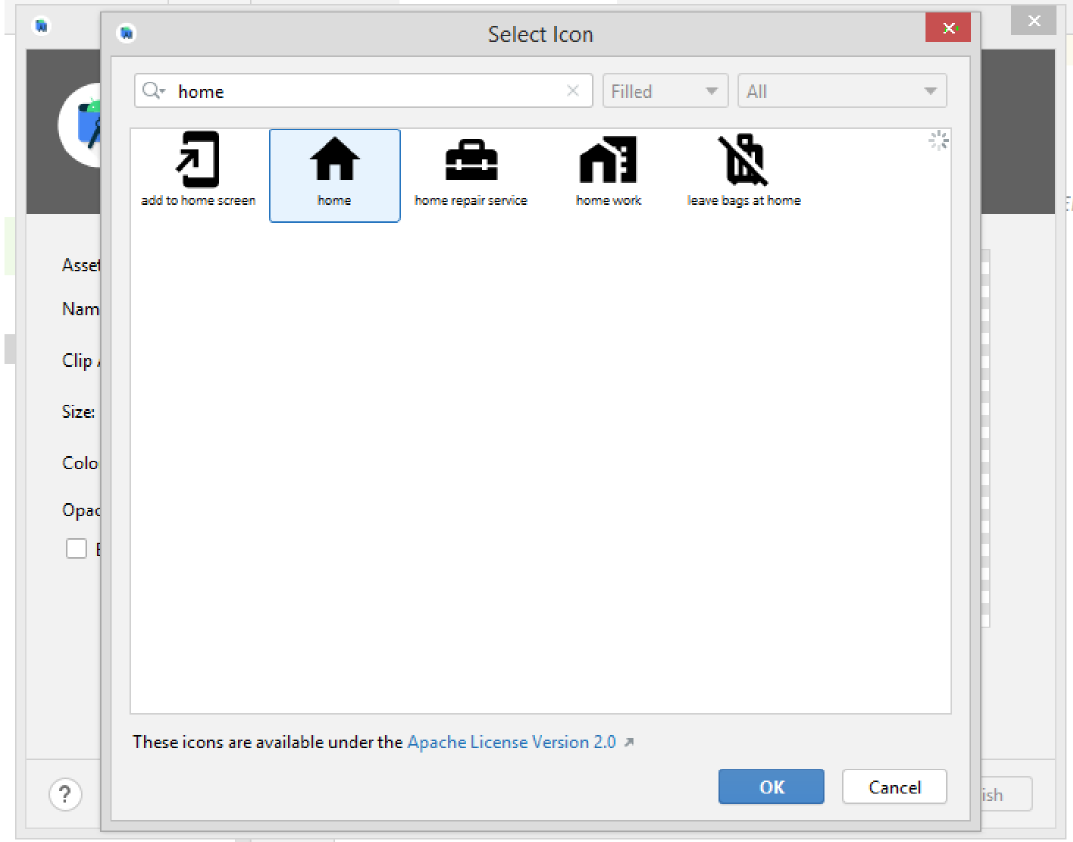
Step by Step Implementation

Step 1: Create a New Project

To create a new project in Android Studio please refer to [Create a new project in android studio in kotlin.]

Step 2: Add a vector asset in the drawable to use as an icon for the menu

To add a vector asset go to: **app > res > drawable > new(right-click) > vector asset**



Step 3: Working with the nav_menu.xml file

Create a menu directory and then add a new resource file in the menu for the popup menu. To create a menu in Android Studio please refer to [here](). Here we need to add the item that we need to show in the menu. We need to specify there's id, icon reference, and title. Here is the code for **nav_menu.xml**

- XML

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/home"
    android:icon="@drawable/ic_home_24"
    android:title="Home"  />
  <item
    android:id="@+id/message"
    android:icon="@drawable/ic_baseline_message_24"
    android:title="Message" />
```

```
    <item
        android:id="@+id/settings"
        android:icon="@drawable/ic_settings_24"
        android:title="Setting" />
</menu>
```

Step 4: For each menu, we need to create a fragment. As there are 3 menus so we need to create 3 fragments. To create a fragment in Android Studio please refer to [How to Create a New Fragment in Android Studio](). So three of our fragments are:

- Home Fragment
- Menu Fragment
- Settings Fragment

Here as of now for keeping it simple, We would not change anything in the fragment's Kotlin file. But to differentiate their's UI we will change the XML file.

XML file for the Home fragment:

- XML

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".HomeFragment">

    <!-- TODO: Update blank fragment layout -->
    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Home fragment" />

</FrameLayout>
```

XML file for the Chat fragment:

- XML

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".ChatFragment">

    <!-- TODO: Update blank fragment layout -->
    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Message fragment" />

</FrameLayout>
```

XML file for the Setting fragment:

- XML

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".SettingFragment">

    <!-- TODO: Update blank fragment layout -->
    <TextView
```

```
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Setting fragment" />

</FrameLayout>
```

Step 5: Working with the MainActivity.kt file and activity_main.xml.

this layout file will have a BottomNavigationview component in the bottom part and the top part Would be Covered By Framelayout which will be Replaced By Fragment at Runtime.

- XML

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#ffffff"
    tools:context=".MainActivity">

    <FrameLayout
        android:id="@+id/container"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_above="@+id/bottomNav"
        />


    <com.google.android.material.bottomnavigation.BottomNavigationView
        android:id="@+id/bottomNav"
        android:layout_width="match_parent"
```

```
        android:layout_height="60dp"
        android:layout_alignParentBottom="true"
        android:layout_marginBottom="20dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toStartOf="parent"
        app:menu="@menu/nav_menu"
        android:scrollIndicators="left"/>
</RelativeLayout>
```

Go to the MainActivity.kt file and refer to the following code. Below is the code for the MainActivity.kt file.

- Kotlin

```
package com.ayush.popupmenu

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.util.Log
import android.widget.ImageView
import android.widget.PopupMenu
import android.widget.Toast
import androidx.fragment.app.Fragment
import com.google.android.material.bottomnavigation.BottomNavigationView
import java.lang.Exception

class MainActivity : AppCompatActivity() {

    lateinit var bottomNav : BottomNavigationView
```

```
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        loadFragment(HomeFragment())
        bottomNav = findViewById(R.id.bottomNav) as BottomNavigationView
        bottomNav.setOnItemSelectedListener {
            when (it.itemId) {
                R.id.home -> {
                    loadFragment(HomeFragment())
                    true
                }
                R.id.message -> {
                    loadFragment(ChatFragment())
                    true
                }
                R.id.settings -> {
                    loadFragment(SettingFragment())
                    true
                }
            }
        }
    }
    private  fun loadFragment(fragment: Fragment){
        val transaction = supportFragmentManager.beginTransaction()
        transaction.replace(R.id.container,fragment)
        transaction.commit()
    }


}
```

So, our app is ready.

Fragment manager

bookmark_border

Note: We strongly recommend using the Navigation library to manage your app's navigation. The framework follows best practices for working with fragments, the back stack, and the fragment manager. For more information about Navigation, see Get started with the Navigation component and Migrate to the Navigation component.

FragmentManager is the class responsible for performing actions on your app's fragments, such as adding, removing, or replacing them and adding them to the back stack.

You might never interact with FragmentManager directly if you're using the Jetpack Navigation library, as it works with the FragmentManager on your behalf. However, any app using fragments is using FragmentManager at some level, so it's important to understand what it is and how it works.

This page covers:

- How to access the FragmentManager.
- The role of FragmentManager in relation to your activities and fragments.
- How to manage the back stack with FragmentManager.
- How to provide data and dependencies to your fragments.

Access the FragmentManager

You can access the FragmentManager from an activity or from a fragment.

FragmentActivity and its subclasses, such as AppCompatActivity, have access to the FragmentManager through the getSupportFragmentManager() method.

Fragments can host one or more child fragments. Inside a fragment, you can get a reference to the FragmentManager that manages the fragment's children through getChildFragmentManager(). If you need to access its host FragmentManager, you can use getParentFragmentManager().

Here are a couple of examples to see the relationships between fragments, their hosts, and the FragmentManager instances associated with each.

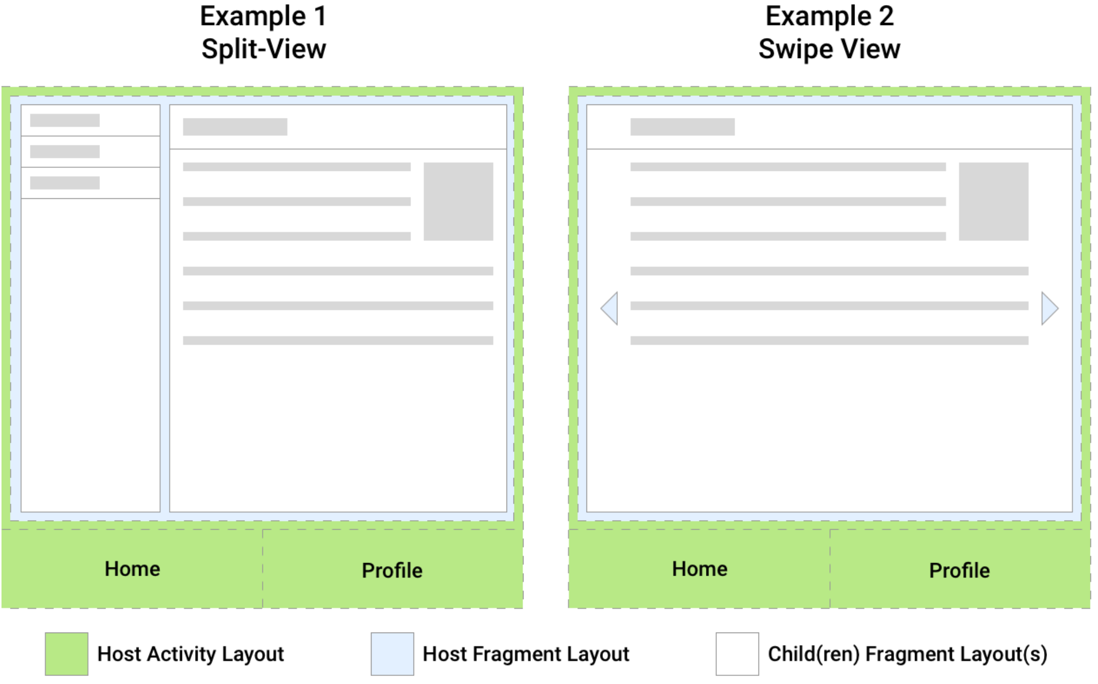


Figure 1. Two UI layout examples showing the relationships between fragments and their host activities.

Figure 1 shows two examples, each of which has a single activity host. The host activity in both of these examples displays top-level navigation to the user as a BottomNavigationView that is responsible for swapping out the host fragment with different screens in the app. Each screen is implemented as a separate fragment.

The host fragment in Example 1 hosts two child fragments that make up a split-view screen. The host fragment in Example 2 hosts a single child fragment that makes up the display fragment of a swipe view.

Given this setup, you can think about each host as having a FragmentManager associated with it that manages its child fragments. This is illustrated in figure 2 along with property mappings between supportFragmentManager, parentFragmentManager, and childFragmentManager.

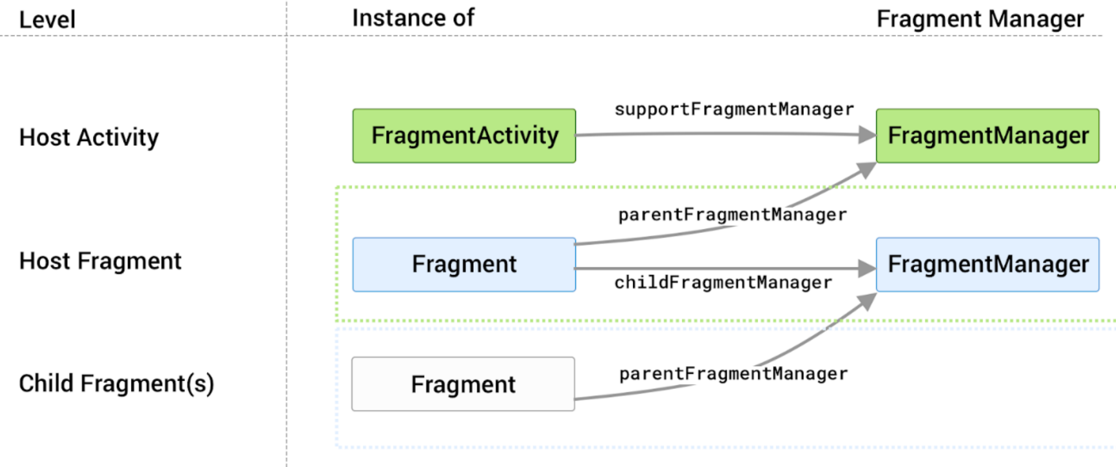


Figure 2. Each host has its own FragmentManager associated with it that manages its child fragments.

The appropriate FragmentManager property to reference depends on where the callsite is in the fragment hierarchy along with which fragment manager you are trying to access.

Once you have a reference to the FragmentManager, you can use it to manipulate the fragments being displayed to the user.

Child fragments

Generally speaking, your app consists of a single or small number of activities in your application project, with each activity representing a group of related screens. The activity might provide a point to place top-level navigation and a place to scope ViewModel objects and other view-state between fragments. A fragment represents an individual destination in your app.

If you want to show multiple fragments at once, such as in a split-view or a dashboard, you can use child fragments that are managed by your destination fragment and its child fragment manager.

Other use cases for child fragments are the following:

- Screen slides, using a ViewPager2 in a parent fragment to manage a series of child fragment views.
- Sub-navigation within a set of related screens.
- Jetpack Navigation uses child fragments as individual destinations. An activity hosts a single parent NavHostFragment and fills its space with different child destination fragments as users navigate through your app.

Use the FragmentManager

The FragmentManager manages the fragment back stack. At runtime, the FragmentManager can perform back stack operations like adding or removing fragments in response to user interactions. Each set of changes is committed together as a single unit called a FragmentTransaction. For a more in-depth discussion about fragment transactions, see the fragment transactions guide.

When the user taps the Back button on their device, or when you call FragmentManager.popBackStack(), the top-most fragment transaction pops off of the stack. If there are no more fragment transactions on the stack, and if you aren't using child fragments, the Back event bubbles up to the activity. If you *are* using child fragments, see special considerations for child and sibling fragments.

When you call addToBackStack() on a transaction, the transaction can include any number of operations, such as adding multiple fragments or replacing fragments in multiple containers.

When the back stack is popped, all these operations reverse as a single atomic action. However, if you committed additional transactions prior to the popBackStack() call, and if you *didn't* use addToBackStack() for the transaction, these operations *don't* reverse. Therefore, within a single FragmentTransaction, avoid interleaving transactions that affect the back stack with those that don't.

Perform a transaction

To display a fragment within a layout container, use the FragmentManager to create a FragmentTransaction. Within the transaction, you can then perform an add() or replace() operation on the container.

For example, a simple FragmentTransaction might look like this:

KotlinJava

```
supportFragmentManager.commit {
  replace<ExampleFragment>(R.id.fragment_container)
  setReorderingAllowed(true)
  addToBackStack("name") // Name can be null
}
```

In this example, ExampleFragment replaces the fragment, if any, that is currently in the layout container identified by the R.id.fragment_container ID. Providing the fragment's class to the replace() method lets the FragmentManager handle instantiation using its FragmentFactory. For more information, see the Provide dependencies to your fragments section.

setReorderingAllowed(true) optimizes the state changes of the fragments involved in the transaction so that animations and transitions work correctly. For more information on navigating with animations and transitions, see Fragment transactions and Navigate between fragments using animations.

Calling addToBackStack() commits the transaction to the back stack. The user can later reverse the transaction and bring back the previous fragment by tapping the Back button. If you added or removed multiple fragments within a single transaction, all those operations are undone when the back stack is popped. The optional name provided in the addToBackStack() call gives you the ability to pop back to a specific transaction using popBackStack().

If you don't call addToBackStack() when you perform a transaction that removes a fragment, then the removed fragment is destroyed when the transaction is committed, and the user cannot navigate back to it. If you do call addToBackStack() when removing a fragment, then the fragment is only STOPPED and is later RESUMED when the user navigates back. Its view *is* destroyed in this case. For more information, see Fragment lifecycle.

Find an existing fragment

You can get a reference to the current fragment within a layout container by using findFragmentById(). Use findFragmentById() to look up a fragment either by the given ID when inflated from XML or by the container ID when added in a FragmentTransaction. Here's an example:

KotlinJava

```
supportFragmentManager.commit {
  replace<ExampleFragment>(R.id.fragment_container)
  setReorderingAllowed(true)
  addToBackStack(null)
}
...
val fragment: ExampleFragment =
      supportFragmentManager.findFragmentById(R.id.fragment_container) as ExampleFragment
```

Alternatively, you can assign a unique tag to a fragment and get a reference using findFragmentByTag(). You can assign a tag using the android:tag XML attribute on fragments that are defined within your layout or during an add() or replace() operation within a FragmentTransaction.

KotlinJava

```
supportFragmentManager.commit {
  replace<ExampleFragment>(R.id.fragment_container, "tag")
  setReorderingAllowed(true)
  addToBackStack(null)
}
...
val fragment: ExampleFragment =
      supportFragmentManager.findFragmentByTag("tag") as ExampleFragment
```

Special considerations for child and sibling fragments

Only one FragmentManager can control the fragment back stack at any given time. If your app shows multiple sibling fragments on the screen at the same time, or if your app uses child fragments, then one FragmentManager is designated to handle your app's primary navigation.

To define the primary navigation fragment inside of a fragment transaction, call the setPrimaryNavigationFragment() method on the transaction, passing in the instance of the fragment whose childFragmentManager has primary control.

Consider the navigation structure as a series of layers, with the activity as the outermost layer, wrapping each layer of child fragments underneath. Each layer has a single primary navigation fragment.

When the Back event occurs, the innermost layer controls navigation behavior. Once the innermost layer has no more fragment transactions from which to pop back, control returns to the next layer out, and this process repeats until you reach the activity.

When two or more fragments are displayed at the same time, only one of them is the primary navigation fragment. Setting a fragment as the primary navigation fragment removes the designation from the previous fragment. Using the preceding example, if you set the detail fragment as the primary navigation fragment, the main fragment's designation is removed.

Support multiple back stacks

In some cases, your app might need to support multiple back stacks. A common example is if your app uses a bottom navigation bar. FragmentManager lets you support multiple back stacks with the saveBackStack() and restoreBackStack() methods. These methods let you swap between back stacks by saving one back stack and restoring a different one.

Note: Alternatively, you can use the **NavigationUI** component, which automatically handles multiple back stack support for bottom navigation.

saveBackStack() works similarly to calling popBackStack() with the optional name parameter: the specified transaction and all transactions after it on the stack are popped. The difference is that saveBackStack() saves the state of all fragments in the popped transactions.

For example, suppose you previously added a fragment to the back stack by committing a FragmentTransaction using addToBackStack(), as shown in the following example:

KotlinJava

```
supportFragmentManager.commit {
  replace<ExampleFragment>(R.id.fragment_container)
  setReorderingAllowed(true)
  addToBackStack("replacement")
}
```

In that case, you can save this fragment transaction and the state of ExampleFragment by calling saveBackStack():

KotlinJava

```
supportFragmentManager.saveBackStack("replacement")
```

Note: You can use **saveBackStack()** only with transactions that call **setReorderingAllowed(true)** so that the transactions can be restored as a single, atomic operation.

You can call restoreBackStack() with the same name parameter to restore all of the popped transactions and all of the saved fragment states:

KotlinJava

```
supportFragmentManager.restoreBackStack("replacement")
```

Note: You can't use **saveBackStack()** and **restoreBackStack()** unless you pass an optional name for your fragment transactions with **addToBackStack()**.

Provide dependencies to your fragments

When adding a fragment, you can instantiate the fragment manually and add it to the FragmentTransaction.

KotlinJava

```
fragmentManager.commit {
   // Instantiate a new instance before adding
   val myFragment = ExampleFragment()
   add(R.id.fragment_view_container, myFragment)
   setReorderingAllowed(true)
}
```

When you commit the fragment transaction, the instance of the fragment you created is the instance used. However, during a configuration change, your activity and all of its fragments are destroyed and then recreated with the most applicable Android resources.
The FragmentManager handles all of this for you: it recreates instances of your fragments, attaches them to the host, and recreates the back stack state.

By default, the FragmentManager uses a FragmentFactory that the framework provides to instantiate a new instance of your fragment. This default factory uses reflection to find and invoke a no-argument constructor for your fragment. This means that you can't use this default factory to provide dependencies to your fragment. It also means that any custom constructor you used to create your fragment the first time is *not* used during recreation by default.

To provide dependencies to your fragment, or to use any custom constructor, instead create a custom FragmentFactory subclass and then override FragmentFactory.instantiate. You can then override the default factory of the FragmentManager with your custom factory, which is then used to instantiate your fragments.

Suppose you have a DessertsFragment that is responsible for displaying popular desserts in your hometown, and that DessertsFragment has a dependency on a DessertsRepository class that provides it with the information it needs to display the correct UI to your user.

You might define your DessertsFragment to require a DessertsRepository instance in its constructor.

KotlinJava

```
class DessertsFragment(val dessertsRepository: DessertsRepository) : Fragment() {
   ...
}
```

A simple implementation of your FragmentFactory might look similar to the following.

KotlinJava

```
class MyFragmentFactory(val repository: DessertsRepository) : FragmentFactory() {
   override fun instantiate(classLoader: ClassLoader, className: String): Fragment =
         when (loadFragmentClass(classLoader, className)) {
            DessertsFragment::class.java -> DessertsFragment(repository)
            else -> super.instantiate(classLoader, className)
```

```
    }
}
```

This example subclasses FragmentFactory, overriding the instantiate() method to provide custom fragment creation logic for a DessertsFragment. Other fragment classes are handled by the default behavior of FragmentFactory through super.instantiate().

You can then designate MyFragmentFactory as the factory to use when constructing your app's fragments by setting a property on the FragmentManager. You must set this property prior to your activity's super.onCreate() to ensure that MyFragmentFactory is used when recreating your fragments.

KotlinJava

```
class MealActivity : AppCompatActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    supportFragmentManager.fragmentFactory =
MyFragmentFactory(DessertsRepository.getInstance())
    super.onCreate(savedInstanceState)
  }
}
```

Setting the FragmentFactory in the activity overrides fragment creation throughout the activity's fragment hierarchy. In other words, the childFragmentManager of any child fragments you add uses the custom fragment factory set here unless overridden at a lower level.

Test with FragmentFactory

In a single activity architecture, test your fragments in isolation using the FragmentScenario class. Since you can't rely on the custom onCreate logic of your activity, you can instead pass the FragmentFactory in as an argument to your fragments test, as shown in the following example:

```
// Inside your test
val dessertRepository = mock(DessertsRepository::class.java)
launchFragment<DessertsFragment>(factory =
MyFragmentFactory(dessertRepository)).onFragment {
  // Test Fragment logic
}
```

Clean Navigation Between Screens in Android Applications

February 6, 2020 by Vasiliy

In this post I'm going to talk about navigation between screens in Android applications, architecture and Navigation Architecture Component.

On the first sight, the topic of navigation might look mundane, almost trivial. However, by the end of this post, you'll see that this first impression couldn't be farther from truth. Navigation between screens is the core architectural aspect in your app and the way you approach this task makes a huge difference in its long-term maintainability.

Starting Activities, replacing Fragments

There are two main ways to represent "screens" in Android apps: Activity-per-screen and Fragment-per-screen. There are other approaches as well, but they are much less popular. Therefore, I'll concentrate just on these two in the context of our discussion here.

If you'd like to navigate to a screen represented by Activity, you'd do it like this:

```
Intent intent = new Intent(context, TargetActivity.class);

context.startActivity(intent);
```

With Fragments, you need to write a bit more code:

```
SomeFragment fragment = new SomeFragment();

fragmentManager

.beginTransaction()

.replace(R.id.fragment_container, fragment)

.commit();
```

But that's not all. When you start a new Activity, the old one is automatically added to the backstack by default. That's not the case with Fragments. To enable back-navigation with Fragments, you need to explicitly add transactions to the backstack:

```
TargetFragment fragment = new TargetFragment();

fragmentManager

.beginTransaction()

.addToBackStack(null)

.replace(R.id.fragment_container, fragment)

.commit();
```

All in all, not that difficult, right? Let's move on.

Activity extras and Fragment arguments

In some cases, you'll want to pass data to the destination screen. For example, imagine that you have a list of products, and when the user clicks on one of them, you want to show a new screen with that product's description. To make this work, you'll need to pass either the entire data structure representing the product, or, at least, product's ID to the next screen.

To pass data into Activities, you use so-called "Intent extras":

```
Intent intent = new Intent(context, TargetActivity.class);

intent.putExtra(TargetActivity.INTENT_EXTRA_PRODUCT, product);

context.startActivity(intent);
```

The destination TargetActivity will then extract this data from the intent:

```
Product product = (Product) getIntent().getExtras().getSerializable(INTENT_EXTRA_PRODUCT);
```

With Fragments, you'd use so-called "Fragment arguments":

```
TargetFragment fragment = new TargetFragment();
Bundle args = new Bundle();
args.putSerializable(TargetFragment.ARG_PRODUCT, product);
fragment.setArguments(args);
fragmentManager
.beginTransaction()
.addToBackStack(null)
.replace(R.id.fragment_container, fragment)
.commit();
```

The destination TargetFragment will then extract the data from its arguments:

```
Product product = (Product) getArguments().getSerializable(ARG_PRODUCT);
```

Please note that I rely on automatic Java serialization (using Serializable marker interface) to put data into intents and arguments. That's my preferred way to pass non-primitive data structures. It's simple to use and maintain. However, many Android developers don't like this approach and prefer to use Parcelables. If that's your preference too, be my guest.

Coupling between screens

Let's talk about the coupling between different screens in your application a bit.

Despite the fact that the term "coupling" is usually used in a very negative context, there is really nothing wrong with it. In fact, coupling is what you do when you interconnect multiple components to work together. However, there are different types and different degrees of coupling, and not all of them created equal.

In the very first examples in this article, I coupled the current screen to the next one either through a reference to Activity class, or a call to Fragment constructor. This kind of coupling is alright.

However, in the examples that demonstrated exchange of data, the coupling became much stronger. In addition to dependency on the high-level details of the target screen, I also made the current screen coupled to the constants defined inside that screen. In fact, the current screen is not just coupled to the constants, but it's also "assuming" that the target screen will use these constants to retrieve the data from either intent or arguments. But even that isn't the entire story. See, using this seemingly simple and innocent approach, I managed to couple the target screen to the current one as well. If I'll forget to provide data that the target screen expects, or provide incorrect types of data, then I can break the target screen, even if it worked fine until this exact moment!

In relatively small applications, such a coupling might not become an issue. But in larger codebases which need to be maintained for years, it might lead to serious problems.

For example, in older codebases I saw screens which were navigated to from multiple other screens and used more than ten different constants for data exchange. These constants would be combined in different ways upon navigation from different screens. When you jump into such code and need to add additional route to that screen, you basically face a very complex puzzle because you need to figure out which constants are mandatory to use, which types of data should be used, whether there are mutually exclusive constants that shouldn't be used together, etc. Since this contract isn't enforced by the compiler, you discover your mistakes only when you run the application. And if the application has multiple flavors which use the data injected into that screen in different ways, then good luck to you.

However, since that's the approach Android framework forces us to use if we want to pass data between screens, there is seemingly no alternative. We should just accept the inevitability of bad design, right? Not on my watch!

Static factory methods to the rescue

To address the issues described in the previous section, I usually use static factory methods.

If the navigation target is Activity, then I'll add this method to its public API:

```
public static void start(Context context, Product product) {

Intent intent = new Intent(context, TargetActivity.class);

intent.putExtra(INTENT_EXTRA_PRODUCT, product);

context.startActivity(intent);

}
```

Then the clients that want to navigate to that Activity will simply do the following:

```
TargetActivity.start(context, product);
```

Note: static methods for starting new Activities don't really qualify as factories because they don't instantiate new objects. However, I still call them static factories for convenience.

If the target is Fragment, then, similarly, I'd do the following:

```
public static TargetFragment newInstance(Product product) {

TargetFragment fragment = new TargetFragment();

Bundle args = new Bundle();

args.putSerializable(ARG_PRODUCT, product);

fragment.setArguments(args);

return fragment;

}
```

And then call this method from clients:

```
TargetFragment fragment = TargetFragment.newInstance(product);

fragmentManager
```

```
.beginTransaction()
.addToBackStack(null)
.replace(R.id.fragment_container, fragment)
.commit();
```

Nothing complex here, right? Just moving a bit of code from one place to another. However, this simple design trick brings enormous benefits:

1. The clients no longer depend on the constants from target screens.
2. The number and the type of parameters is enforced by the compiler through method signature.
3. All the details about mapping of constants to parameters are encapsulated within the target class. The constants are private.
4. If you need to navigate to the same screen from multiple places, then, instead of code duplication, you just call the same method.

In more complex scenarios, when different clients need to pass different sets of data to target screens, I simply add multiple factory methods. I give these methods descriptive names to convey information about their intent to future readers of my code. This way, I explicitly define all valid combinations of data, document their usage and let the compiler enforce that.

It's impossible to overstate the impact of this technique on your codebase. I find it especially beneficial in big and complex applications, but it'll make your life easier in any project beyond the most trivial "hello world". Since it's that simple, you should probably always use it. As far as I can tell, there are no downsides at all.

Screen navigator abstraction

If you think about it, handling navigation within your app is a standalone responsibility. For example, in case of Fragments navigation, the clients shouldn't care about the low level mechanics of FragmentTransaction and whatnot. They just need to state which screen should be shown next and provide the respective parameters (if required).

Therefore, according to Single Responsibility Principle, you should extract this functionality into a standalone component.

So, let's create ScreenNavigator abstraction:

```
public class ScreenNavigator {
private final Activity mActivity;
private final FragmentManager mFragmentManager;
public ScreenNavigator(Activity activity, FragmentManager fragmentManager) {
mActivity = activity;
mFragmentManager = fragmentManager;
}
```

```
public void toProductDetailsScreen(Product product) {
TargetFragment fragment = TargetFragment.newInstance(product);
replaceFragment(fragment);
}
private void replaceFragment(Fragment fragment) {
mFragmentManager
.beginTransaction()
.addToBackStack(null)
.replace(R.id.fragment_container, fragment)
.commit();
}
public void toProductDetailsScreen2(Product product) {
TargetActivity.start(mActivity, product);
}
}
```

Now, whenever you need to navigate to another screen, you just call ScreenNavigator's methods:

```
mScreenNavigator.toProductDetailsScreen(product);
```

In essence, all I did was just extracting the code from individual clients into this new class. I'm pretty sure you aren't mind blown by that. But you should be. See, with this humble change I introduce crucially important architectural boundary into my application.

Note that the clients no longer depend on individual Activity and Fragment classes at all. In fact, the clients don't even know whether a call to ScreenNavigator will result in a new Activity being shown, or a new Fragment (or any other component, really). These implementation details are now abstracted out and I can even change my approach in the future without affecting the existing code. In addition, elimination of dependencies on Android classes in my clients will allow me to unit test them, including their interaction with ScreenNavigator.

Also note that once you have ScreenNavigator in your codebase, it becomes much less important how you actually handle the navigation. Since all these details are encapsulated in a single class now, you can easily switch between manual approach and different external libraries. In essense, the choice of navigation mechanism ceases to be part of your architecture and becomes implementation detail of ScreenNavigator.

I have ScreenNavigator abstractions in all my apps and it's among the first classes I add in clients' codebases. So far, I implemented Activity-per-screen, Activity-per-flow, Fragment-per-screen and Fragment-per-screen-with-bottom-tabs approaches using ScreenNavigator and all of them worked like charm.

Back navigation

When you add ScreenNavigator to your application, also add navigateBack() method to its API and delegate onBackPressed() from all Activities to this method. There is a bit of nuance here, so let me just copy-paste code example from one of my projects:

```
public class ScreenNavigator {

...

public boolean navigateBack() {

if(mFragNavController.isRootFragment()) {

return false;

} else {

mFragNavController.popFragment();

return true;

}

}

}
```

This specific project uses Single-Activity approach and FragNav library to handle Fragments, so I can use FragNavController class instead of hacking with FragmentManager. Modify the implementation of navigateBack() according to your implementation details.

Then, inside your Activities, do the following (if you have many Activities, consider extracting this logic into a base class):

```
@Override

public void onBackPressed() {

if (!mScreensNavigator.navigateBack()) {

super.onBackPressed();

}

}
```

Just in case you've got some other action to do on back press, like closing the nav drawer if it's open, or dismissing a dialog, you can modify this code like this:

```
@Override

public void onBackPressed() {

if (mViewMvc.isDrawerVisible()) {

mViewMvc.closeDrawer();

} else if (!mScreensNavigator.navigateBack()) {

super.onBackPressed();
```

}

}

In addition to handling back button clicks by the user, you can now call navigateBack() in your code to get back to the previous screen. Magic!

Lab activity

Understanding different layout types

a) Linear Layout**:** A Linear Layout is a fundamental layout in Android that organizes its child views in either a horizontal or vertical orientation. Views are arranged one after the other in a sequential manner.

Attributes:

Orientation: Specifies whether the layout is arranged horizontally

(android:orientation="horizontal") or vertically (android:orientation="vertical").

Gravity: Defines how child views are aligned within the layout (e.g., android:gravity="center" centers the child views).

Use Cases:

Linear Layouts are excellent for creating straightforward interfaces, especially when you need elements to be displayed in a single line or column.

Code-Level Example:

Let's create a simple Android layout using a Linear Layout to display a button and a text field in a vertical arrangement.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center">

    <Button
        android:id="@+id/myButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click Me" />

    <EditText
        android:id="@+id/myTextField"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:hint="Enter Text Here" />

</LinearLayout>
```

In this example, we have:

Defined a Linear Layout with vertical orientation.

Added a Button (myButton) and an EditText (myTextField) as child views.

Specified layout attributes like width, height, and text.

b) Relative Layout:

A Relative Layout is a versatile layout in Android that allows you to position UI elements relative to each other or relative to the parent layout. Views are positioned based on their relationships to other views or the parent layout.

Use Cases:

Relative Layouts are particularly useful when designing complex interfaces where elements' positions depend on the location of other views.

Code-Level Example:

Let's create a simple Android layout using a Relative Layout to display a button, an image, and a text view.

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">

    <Button
        android:id="@+id/myButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click Me"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"/>

    <ImageView
        android:id="@+id/myImage"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/ic_launcher_foreground"
        android:layout_below="@id/myButton"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="16dp"  />

    <TextView
        android:id="@+id/myTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, World!"
        android:layout_below="@id/myImage"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="16dp"  />

</RelativeLayout>
```

- The ImageView is positioned below the Button and centered horizontally, with some margin from the top.
- The TextView is positioned below the ImageView and centered horizontally, with some margin from the top.

c) Constraint Layout:

A Constraint Layout is a powerful layout in Android that enables you to create complex user interfaces by defining relationships (constraints) between UI elements. It's particularly well-suited for building responsive and adaptive interfaces.

Features:

Constraints: Define spatial relationships between UI elements (e.g., align to parent, align to another view, set margins).

Guidelines: Allow for precise alignment by providing horizontal and vertical reference lines.

Chains: Control the positioning and sizing of multiple UI elements as a group.

Use Cases:

Constraint Layouts excel in scenarios where you need to create intricate, responsive UIs that adapt well to various screen sizes and orientations.

Code-Level Example:

Let's create a simple Android layout using a Constraint Layout to display a button, an image, and a text view.

```
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/myButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click Me"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        android:layout_margin="16dp"/>

    <ImageView
        android:id="@+id/myImage"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/ic_launcher_foreground"
        app:layout_constraintTop_toBottomOf="@id/myButton"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        android:layout_marginTop="16dp" />

    <TextView
        android:id="@+id/myTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, World!"
        app:layout_constraintTop_toBottomOf="@id/myImage"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        android:layout_marginTop="16dp" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Explanation:

- We've defined a Constraint Layout as the root layout element.
- The Button is constrained to the top, start, and end of the parent layout, ensuring it spans the full width with a margin.
- The ImageView is positioned below the Button and spans the full width as well.
- The TextView is positioned below the ImageView, also spanning the full width.

d)Frame Layout

A Frame Layout is a simple layout in Android that is designed to hold a single child view. By default, the child view is positioned at the top-left corner. This layout is often used when you want to display a single item or layer multiple views on top of each other.

Use Cases:

Single Item Display:

When you have a single item, like an image or a video, that you want to display.

Layering Views:

It's commonly used when you want to overlay multiple views, like buttons, images, or even fragments.

Code-Level Example:

Let's create a simple Android layout using a Frame Layout to display an image.

```
<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">

    <ImageView
        android:id="@+id/myImage"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/ic_launcher_foreground"  />

</FrameLayout>
```

Explanation:

- We've defined a Frame Layout as the root layout element. It spans the entire screen due to match_parent for both layout_width and layout_height.
- Inside the Frame Layout, we have an ImageView, which is the single child view. It's set to wrap its content, which means it will adjust its size based on the dimensions of the image.

- The android:src attribute is set to @drawable/ic_launcher_foreground, which will display the application's launcher icon as the image.

e) Table Layout:

A Table Layout is a ViewGroup in Android that arranges its children in rows and columns, similar to an HTML table. It's an effective way to present data in a tabular format, making it easy for users to understand and compare information.

Attributes:

Stretch Columns and Rows: You can define which columns and rows should stretch to fill available space using attributes like android:stretchColumns and android:stretchRows.

Shrink Columns and Rows: Conversely, you can set specific columns and rows to shrink and occupy minimum space using android:shrinkColumns and android:shrinkRows.

Use Cases:

Tabular Data Display:

Use Table Layout when you have data that logically fits into rows and columns.

Form-like Structures:

It's useful for creating forms or input layouts where data is entered in a structured manner.

Code-Level Example:

Let's create a simple Android layout using a Table Layout to display a form with labels and input fields.

How layouts affect the appearance of your app.

Layouts play a crucial role in determining the visual structure and organization of user interface elements in an Android app. Here's how different layouts can impact the appearance of your app:

Creating Views:

Creating views is a fundamental aspect of Android app development, as views are the building blocks of the user interface. Here's a detailed explanation of creating views:

Adding UI Elements:

Buttons:

Description: Buttons are used to trigger actions or navigate within the app.

Code Example:

```
<Button
    android:id="@+id/myButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Click Me" />
```

Text Fields:

Description: Text fields allow users to input text or numeric data.

Code Example:

```
<EditText                                         android"
    android:id="@+id/myTextField"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:hint="Enter Text Here" />

    <!-- First Row -->
    <TableRow>
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Name:" />

        <EditText
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Enter your name" />
    </TableRow>

    <!-- Second Row -->
    <TableRow>
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Email:" />

        <EditText
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Enter your email" />
    </TableRow>

</TableLayout>
```

Labels (TextViews):

Description: Labels display static or dynamic text.

Code Example:

```
<TextView
    android:id="@+id/myLabel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello, World!" />
```

Images (ImageViews):

Description: Image views display images or icons.

Code Example:

```
<ImageView
    android:id="@+id/myImage"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/my_image" />
```

Lists (ListView or RecyclerView):

Description: Lists display a scrollable list of items.

Code Example (using RecyclerView):

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/myRecyclerView"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Checkboxes and Radio Buttons:

Description: Checkboxes allow users to select multiple options, while radio buttons allow only one selection.

Code Example (Checkbox):

```
<CheckBox
    android:id="@+id/myCheckBox"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Accept Terms and Conditions" />
```

Spinners:

Description: Spinners provide a dropdown list of options.

Code Example:

```
<Spinner
    android:id="@+id/mySpinner"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

.

Day 3: Working with Navigation and Menus

Navigation Basics:

Setting up fragment managers (Android).

Navigating between different screens or views.

Navigation Architecture Component

Now it's time to address the elephant in the room: which parts of what I wrote in this post are still relevant given there is an official Navigation Architecture Component from Google?

If you think about it, conceptually, Navigation Component attempts to address the same issue that ScreenNavigator addressed: we'd like to have centralized management of navigation inside Android application and decouple the clients from the details of navigation. Sure, Navigation Component also has "navigation editor" (copy of "storyboards" from Xcode), but it has nothing to do with software design and architecture, so it isn't that important in the context of our discussion.

Now, unlike ScreenNavigator, Navigation Component is horribly complex:

- You need to use XMLs to define your navigation graphs.
- If you want to pass any parameters between the origin and the destination in a type-safe and decoupled way, it looks like your only option is to use some Gradle plugin that generates code.
- If you don't like code generation, then back to coupling through Bundles and code duplication.
- You can't have navigation graph that spans multiple activities.

There are probably more complexities and limitations, but the above list is already pretty bad.

I guess many developers will think at this point: "This guy is complaining about XMLs. It's not really that big of a deal!". Well, not exactly. I don't mind XMLs where they make sense, but in this case, it was a poor choice.

Just think about this trivial use case: you want to find all places in code that navigate to a specific screen. With ScreenNavigator, this amounts to just finding usages of one or more simple methods defined in ScreenNavigator. With Navigation Component, it's more nuanced and requires to look "under the hood" of the abstraction, making it so called "leaky abstraction". Furthermore, if you used the aforementioned plugin to pass parameters to the next screen in a type-safe manner, this task will require completely different approach.

In my opinion, Google took a very wrong turn with Navigation component. When I look at this library, I see Loaders all over again: extremely complex library that addressed already solvable problem in a very complex and inconvenient way. My prediction is that it'll also share the fate of Loaders: waste millions of man-hours of developers' time and then fade into irrelevance (unless Google somehow forces us to use this monster, of course). I just hope that all this waste will be

accompanied by some second-order positive impact. For example, maybe Google will finally make Fragment transitions simple and reliable.

Therefore, all in all, my recommendation is to stay away from Navigation Component. If you absolutely want to use it, then, at the very least, wrap this nonsense in ScreenNavigator abstraction. This way you'll be able to refactor it out of your codebase relatively easily in the future.

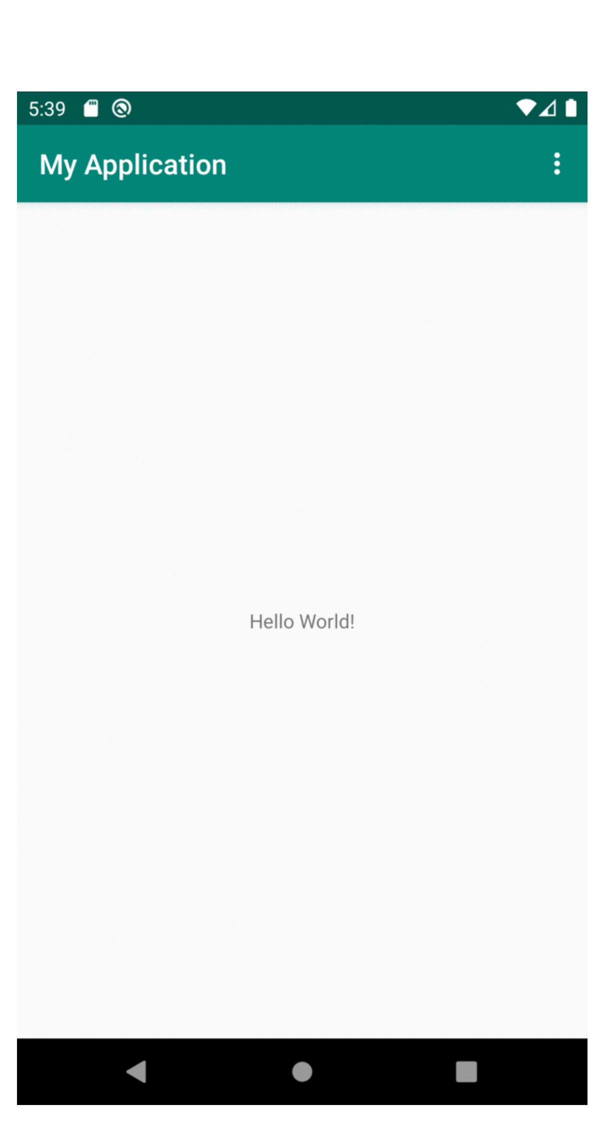
Creating Menus:

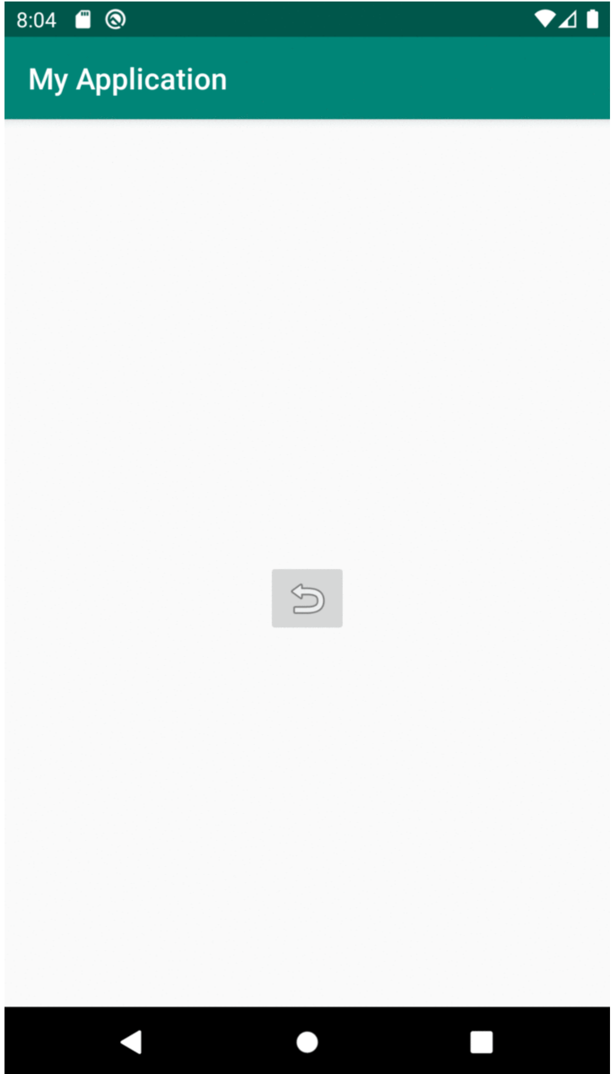
Implementing menus for actions or navigation options.

Contextual menus for specific views or elements.

Create a menu for your Android app

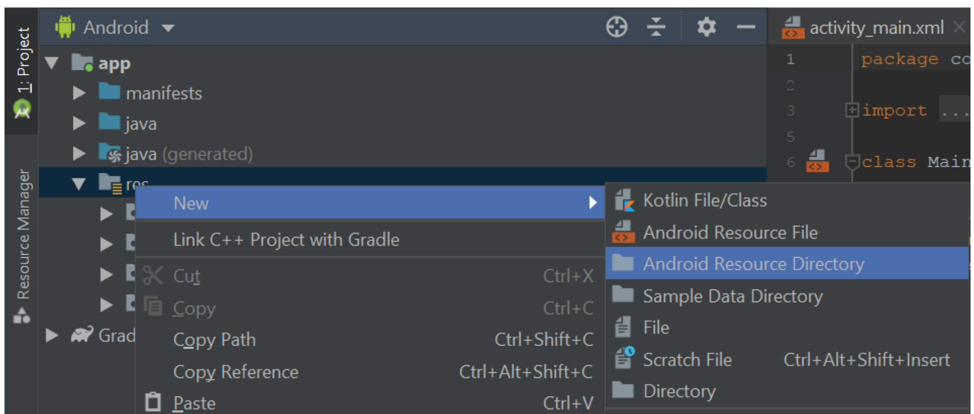
This tutorial will cover how to create and add menus to your Android app. You will also learn how to create submenus, dynamically add or remove items from the menu, and create a pop-up menu.


5:39
My Application
Hello World!

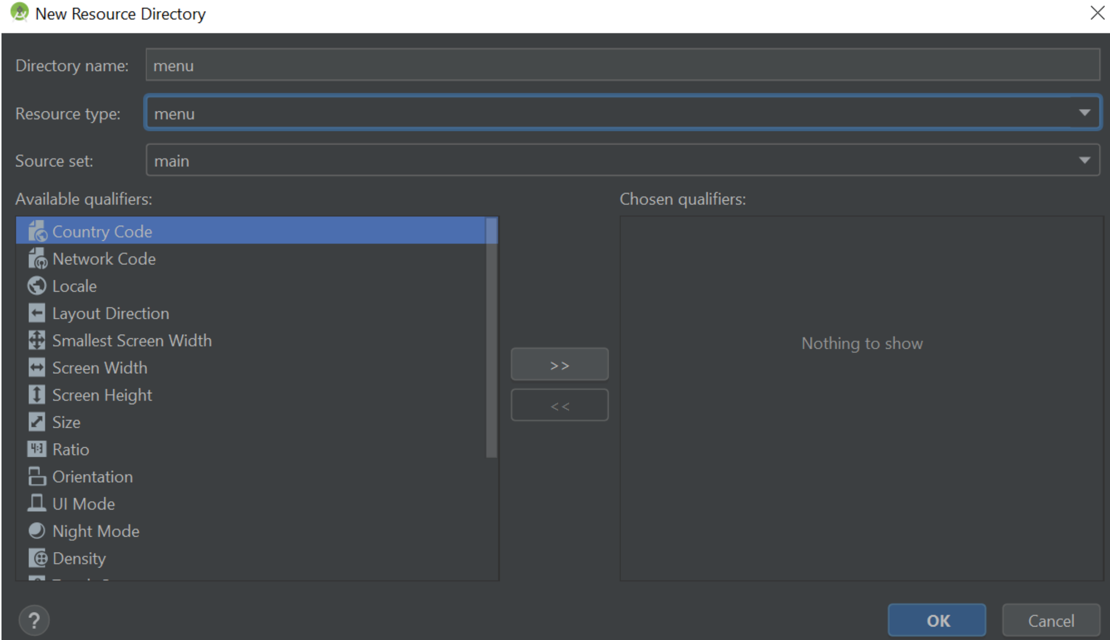


The menu resource directory

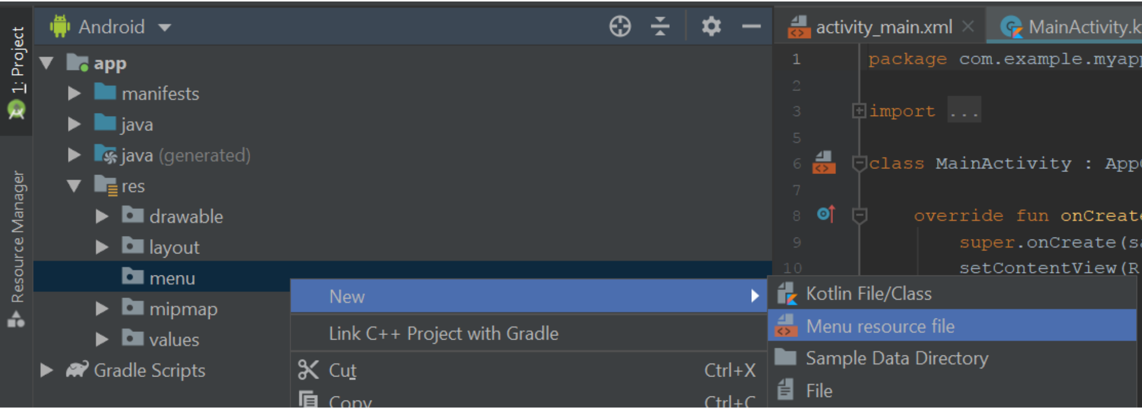
Each menu will require a layout resource file to coordinate its contents. Layout resource files are written using XML and stored inside a dedicated directory called menu. To create the menu directory, right-click the **res** folder (found by navigating through **Project** > **app**) and select **New** > **Android Resource Directory**



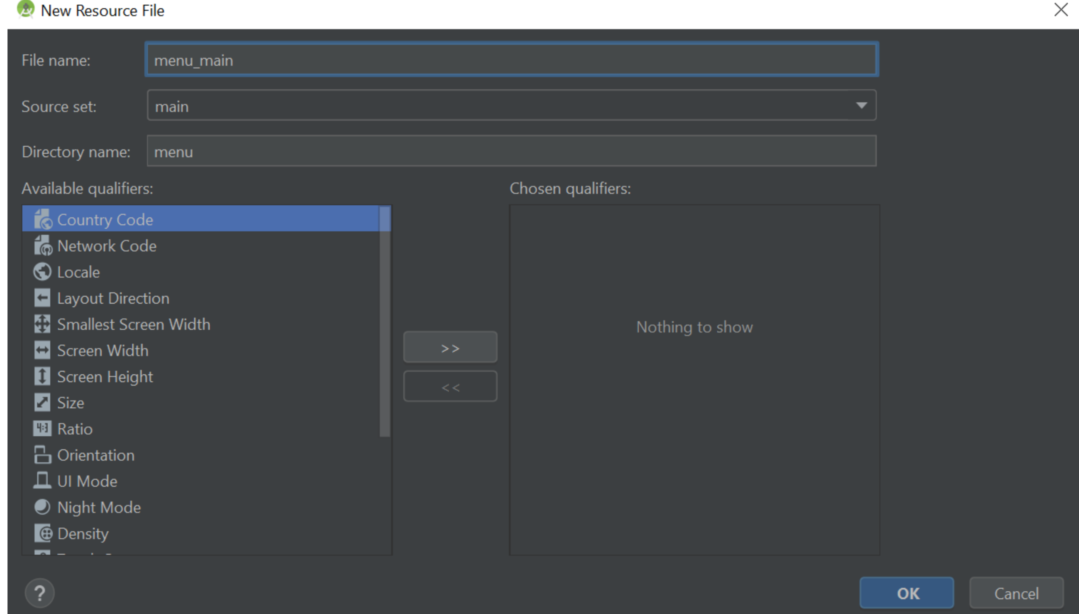
Set both the directory name and resource type to menu then press OK.



Once the menu directory is in place, you can create the layout. To create a new layout resource, right-click the menu directory then select **New** > **Menu resource file**



Set the file name to menu_main then press OK.



A layout file called **menu_main.xml** should then open in the editor. In the next section, we will populate the layout with items and create a menu.

The basic structure of the menu

Open the **menu_main.xml** layout in Code view and edit the file so it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>

<menu xmlns:android="http://schemas.android.com/apk/res/android"

    xmlns:app="http://schemas.android.com/apk/res-auto">


    <item android:id="@+id/menu_settings"

        android:title="@string/menu_settings"

        app:showAsAction="never" />


</menu>
```

Copy

In the above code, we define the menu resource and add an item element to it. The item will display the text which is encoded in a string resource called menu_settings. Currently, the menu_settings string resource does not exist. Let's fix that by opening up the strings resource file. You can find this file by navigating through **Project** > **app** > **res** > **values**

Next, add the following string to the file:

```
<string name="menu_settings">Settings</string>
```

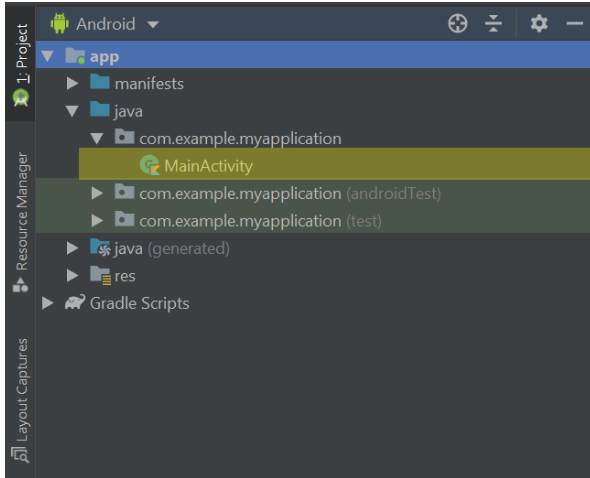
Copy

The menu item code also includes the line **app:showAsAction="never"**. This line defines an attribute called showAsAction, which determines how the menu item should be displayed. It will accept the following values:

- **never** - This option means the item will never be immediately visible in the action bar. Instead, the user will have to press the overflow icon.
- **always** - The item will always be visible in the action bar.
- **ifRoom** - The item will only be visible if there is sufficient space.
- **withText** - This attribute only applies if the menu item contains an icon image as well. It ensures both the icon and title are displayed together.

Inflating the menu

To display the menu to the user ,we must 'inflate' the menu resource file. To do this, open the **MainActivity.kt** file by navigating through **Project** > **app** > **java** > **folder with your project name**



Next, enter the following code below the onCreate function:

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
    val inflater = menuInflater
    inflater.inflate(R.menu.menu_main, menu)
    return super.onCreateOptionsMenu(menu)
}
```

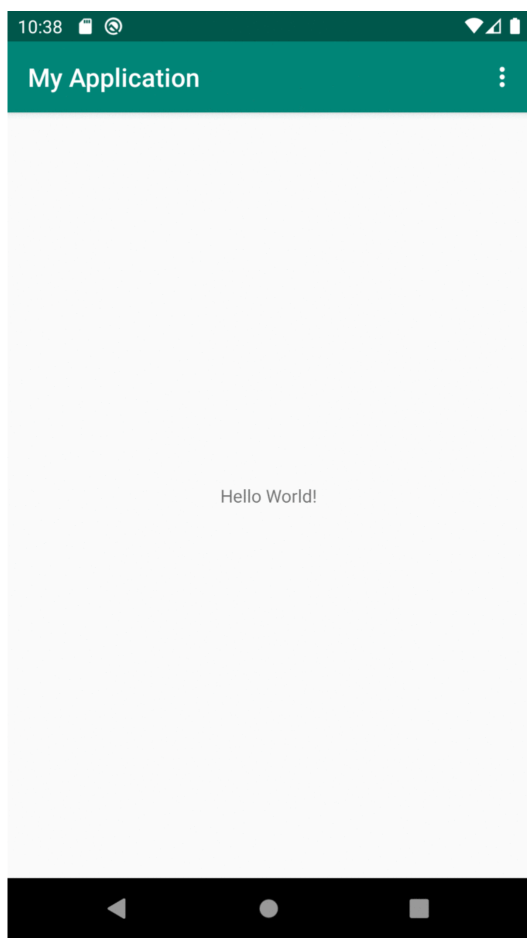
Copy

Note you may also need to add the following statement to the top of the **MainActivity.kt** file to import the Menu class:

```
import android.view.Menu
```

Copy

In the above code, we override the **onCreateOptionsMenu** function, which is a readymade function that Android activities use to display menus. We override the method because we want to provide additional instructions for opening the menu that we made. For example, the above code uses the menuInflater class to import the contents of the **menu_main.xml** file into the action bar. The app is now ready to display our menu. For the remainder of this tutorial, we will explore how to format the menu and make the menu functional.



Creating a submenu and grouping items

Submenus

If you want to create a submenu then simply add another menu element inside the item that will open the submenu. For example, you could add two submenu items to the menu like this:

```
<?xml version="1.0" encoding="utf-8"?>

<menu xmlns:android="http://schemas.android.com/apk/res/android"

  xmlns:app="http://schemas.android.com/apk/res-auto">


  <item android:id="@+id/menu_settings"

    android:title="@string/menu_settings"

    app:showAsAction="never">
```

```
        <!-- The submenu starts here -->
        <menu>
            <item android:id="@+id/submenu_item1"
                android:title="@string/submenu_item1" />
            <item android:id="@+id/submenu_item2"
                android:title="@string/submenu_item2" />
        </menu>
    </item>
</menu>
```

Copy

Remember to define string resources in the strings.xml file for each submenu item's title.

```
<string name="submenu_item1">Item 1</string>
<string name="submenu_item2">Item 2</string>
```

Copy

Grouping menu items

In addition to creating a submenu, you can also group menu items together so they can be worked on collectively.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">


  <item android:id="@+id/menu_settings"
    android:title="@string/menu_settings"
    app:showAsAction="never">


    <!-- Grouped items go here -->
    <group android:id="@+id/group_items1">
      <item android:id="@+id/group_item1"
        android:title="@string/group_item1"
        app:showAsAction="ifRoom" />
      <item android:id="@+id/group_item2"
        android:title="@string/group_item2"
        app:showAsAction="ifRoom" />
    </group>
  </item>
</menu>
```

Copy

Any items in the same group that share a **showAsAction** attribute of 'ifRoom' will either all be displayed in the action bar together or not at all. You won't get one item showing without the other.

Another useful reason to group menu items is that you can apply methods to all of them with a single command. For example, you can use:

- **setGroupVisible()** - Hides/reveals all group items.
- **setGroupEnabled()** - Enables/disables all group items
- **setGroupCheckable()** - Determines whether the group items can display a checkmark

You can apply these methods in your Kotlin code so they are initiated when a function is called. For example, you could run the methods when the menu is loaded:

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
    val inflater = menuInflater
    inflater.inflate(R.menu.menu_main, menu)

    // Set whether the group items are visible or not using 'true' and 'false'
    menu.setGroupVisible(R.id.group_items1, true)

    // Determine whether the group items can be pressed or not (true or false)
    menu.setGroupEnabled(R.id.group_items1, true)

    // Render group items checkable (true or false) and determine whether only one item or all items can be checked at once (exclusive: true or false)
    menu.setGroupCheckable(R.id.group_items1, true, false)

    return super.onCreateOptionsMenu(menu)
}
```

Copy

Modifying the menu based on user activity

There are times when you may wish to change the layout of the menu based on user action. For example, maybe you want a 'Download' button to appear when the user presses a button. To make a menu item appear or disappear, open the Kotlin file that manages the menu (e.g. **MainActivity.kt** and declare the following variables:

```
private var downloadItem = 1

private var showDownloadItem = false
```

Copy

Together these variables will help us create a menu item that is hidden by default because the boolean value is set to false.

Next, add the following line to the **onCreateOptionsMenu** function to incorporate the Download item into the menu:

```
menu.add(0, downloadItem, 0, R.string.download_item)
```

Copy

Note you may need to define a string called download_item in your strings.xml file:

```
<string name="download_item">Download</string>
```

Copy

Next, add a function called onPrepareOptionsMenu to your Kotlin file. The onPrepareOptionsMenu function can modify the menu's items. In this case, we set the isVisible property of the Download item to the value of the showDownloadItem variable. As it stands, the value of the showDownloadItem variable is false, so the Download item will be hidden by default.

```
override fun onPrepareOptionsMenu(menu: Menu?): Boolean {
    var menuItem = menu!!.findItem(downloadItem)
    menuItem.isVisible = showDownloadItem
    menuItem.setShowAsAction(MenuItem.SHOW_AS_ACTION_ALWAYS)
    return super.onPrepareOptionsMenu(menu)
}
```

Copy

Note you may need to add the following statement to the top of the Kotlin file to import the MenuItem class.

```
import android.view.MenuItem
```

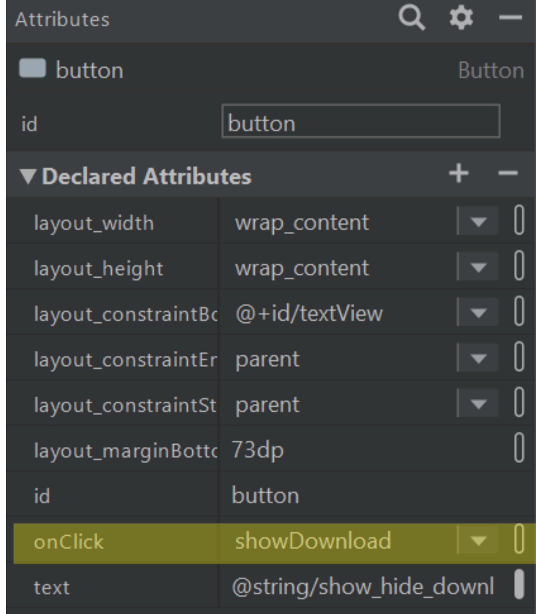
Copy

Next, add a function called showDownload to your Kotlin file using the code shown below. The showDownload function toggles the value of the showDownloadItem variable from false to true (or vice versa) then runs a command called invalidateOptionsMenu. The invalidateOptionsMenu command will regenerate the menu and hide/reveal the Download item because the value of the showDownloadItem variable has changed.

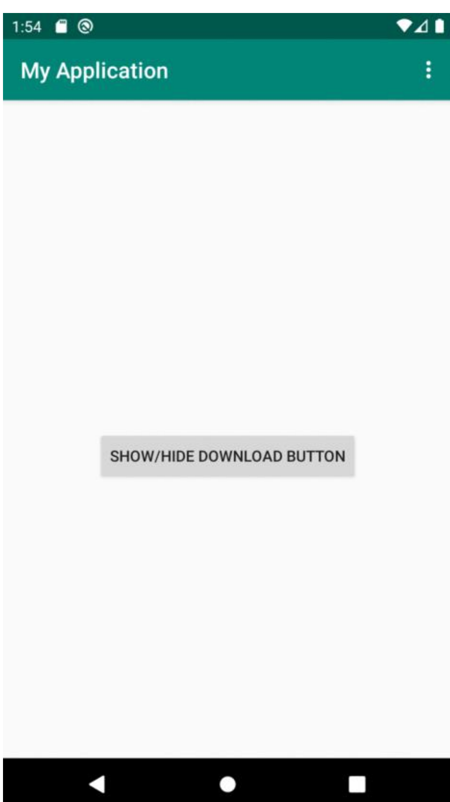
```
fun showDownload() {
    showDownloadItem = !showDownloadItem
    invalidateOptionsMenu()
}
```

Copy

The last thing to do is add a button to one of the app's user interface layout files such as **activity_main.xml**. The button should contain an onClick attribute set to showDownload, which will instruct the button to run the showDownload function and hide/reveal the Download menu item whenever the button is clicked.



Your app should now work like this:



Creating a pop-up menu

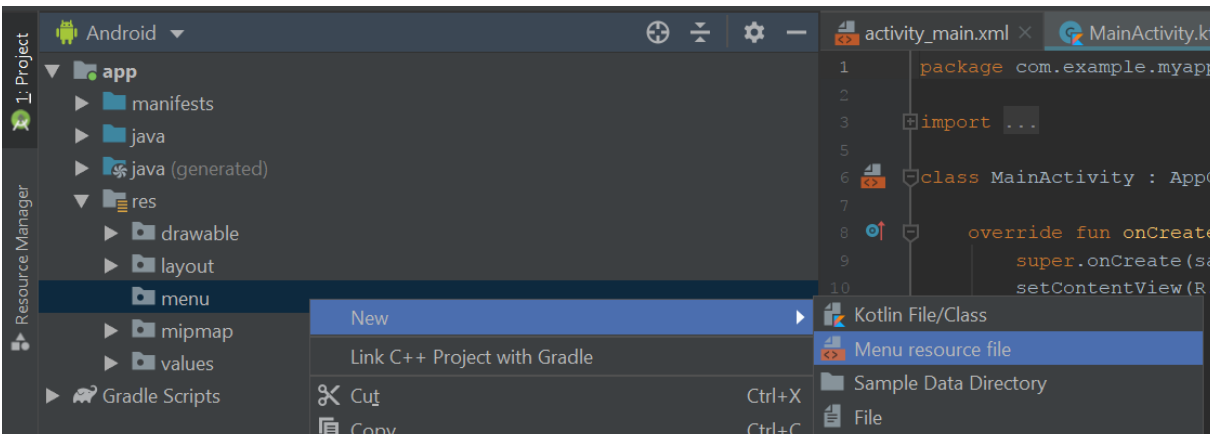
Pop-up menus are not restricted to the action bar and can be loaded in the main body of the app. For example, you might want to create a reply button that gives the user a list of options for responding to an email.

First, add the following items to your [strings.xml file] to define the string resources that we will use to populate the menu items:

```
<string name="menu_reply">Reply</string>

<string name="menu_reply_all">Reply All</string>

<string name="menu_forward">Forward</string>
```

Copy

Next, create a layout file for the pop-up menu. To do this, navigate through **Project** > **app** > **res**, right-click the **menu** folder and select **New** > **Menu resource file**.

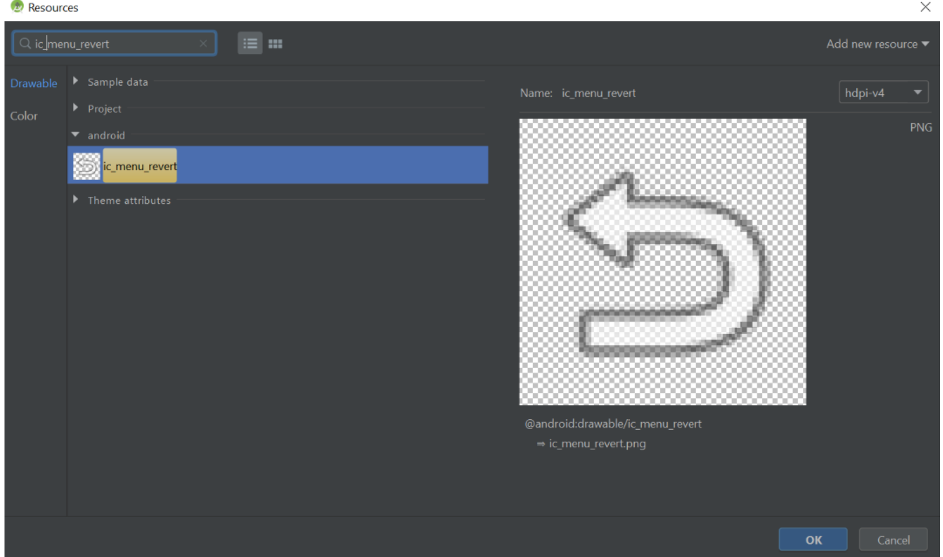


Name the file popup then press OK. Copy and paste the following code into the newly created **popup.xml** file to give the menu three items (reply, reply all and forward).

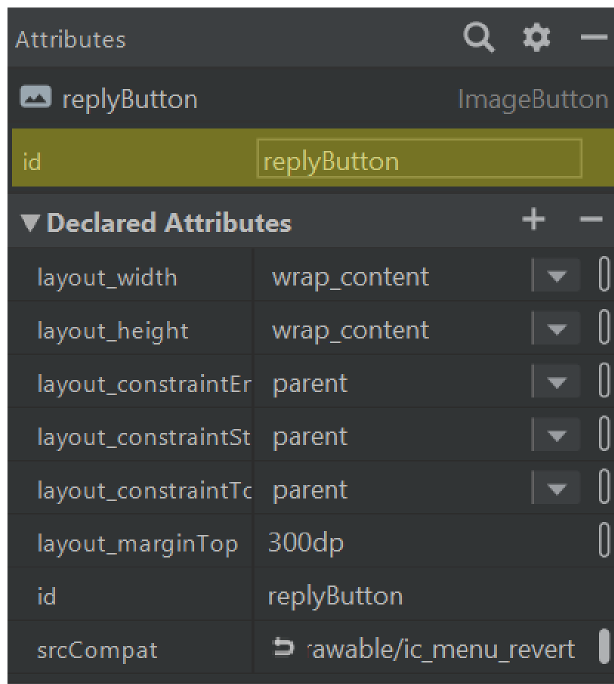
```
<?xml version="1.0" encoding="utf-8"?>

<menu xmlns:android="http://schemas.android.com/apk/res/android">

    <item android:id="@+id/menu_reply"

        android:title="@string/menu_reply" />

    <item android:id="@+id/menu_reply_all"

        android:title="@string/menu_reply_all" />

    <item android:id="@+id/menu_forward"

        android:title="@string/menu_forward" />

</menu>
```

Copy

We will use an ImageButton to load the popup menu. Add an ImageButton widget to your layout of choice (e.g. **activity_main.xml**) by dragging and dropping the widget from the Palette. A resources window should open and invite you to set the icon that should be used for the button. Set the icon to ic_menu_revert then press OK.



Set the id of the ImageButton to replyButton so you can refer to the button in your Kotlin code.



We'll now write the code which will allow the ImageButton to load the pop-up menu when pressed. Open your main Kotlin file (e.g. MainActivity.kt) and add the following code to the onCreate function:

```
val clickListener = View.OnClickListener { view ->
    when (view.id) {
        R.id.replyButton -> {
            showPopup(view)
        }
    }
```

```
}


replyButton.setOnClickListener(clickListener)
```

Copy

The above code calls the showPopup function each time the ImageButton is clicked. We have not written this function yet and so it may be coloured in red. Note you may also need to import the layout which holds the ImageButton. If the ImageButton is located in the **activity_main.xml** file then the import statement will look like this:

```
import kotlinx.android.synthetic.main.activity_main.*
```

Copy

All we need to do now to get the ImageButton working is write the showPopup function. This function will inflate the **popup.xml** menu layout whenever the user presses the ImageButton:

```
private fun showPopup(view: View) {
    val popup = PopupMenu(this, view)
    popup.inflate(R.menu.popup)


    popup.show()
}
```

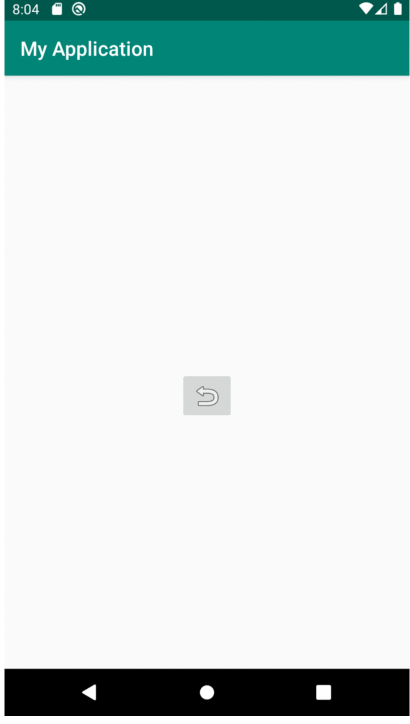
Copy

Note you may need to import the PopupMenu class.

```
import androidx.appcompat.widget.PopupMenu
```

Copy

And that's it! You have now created a pop-up menu!



Add menus

bookmark_border

Try the Compose way

Jetpack Compose is the recommended UI toolkit for Android. Learn how to add components in Compose.

TopAppBar →

Menus are a common user interface component in many types of apps. To provide a familiar and consistent user experience, use the Menu APIs to present user actions and other options in your activities.

Note: For a better user experience, see Material Design Menus.

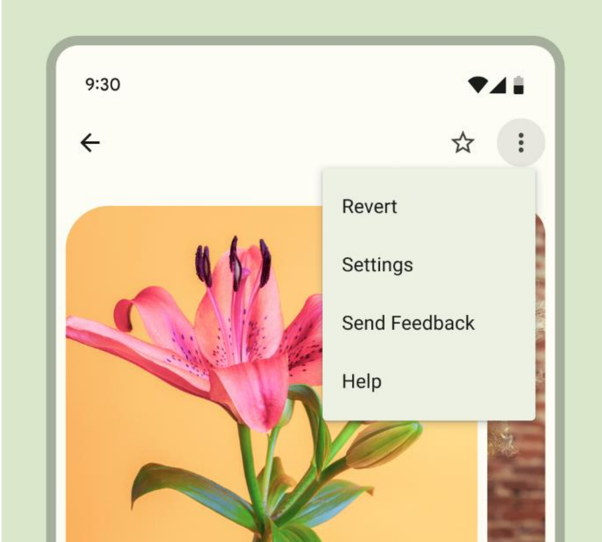


Figure 1. A menu triggered by an icon tap, appearing below the overflow menu icon.

This document shows how to create the three fundamental types of menus or action presentations on all versions of Android:

Options menu and app bar

The options menu is the primary collection of menu items for an activity. It's where you place actions that have a global impact on the app, such as "Search," "Compose email," and "Settings."

See the Create an options menu section.

Context menu and contextual action mode

A context menu is a floating menu that appears when the user performs a touch & hold on an element. It provides actions that affect the selected content or context frame.

The contextual action mode displays action items that affect the selected content in a bar at the top of the screen and lets the user select multiple items.

See the Create a contextual menu section.

Popup menu

A popup menu displays a vertical list of items that's anchored to the view that invokes the menu. It's good for providing an overflow of actions that relate to specific content or to provide options for the second part of a command. Actions in a popup menu don't directly affect the corresponding content—that's what contextual actions are for. Rather, the popup menu is for extended actions that relate to regions of content in your activity.

Define a menu in XML

For all menu types, Android provides a standard XML format to define menu items. Instead of building a menu in your activity's code, define a menu and all its items in an XML menu resource.

You can then inflate the menu resource—loading it as a Menu object—in your activity or fragment.

Using a menu resource is good practice for the following reasons:

- It's easier to visualize the menu structure in XML.
- It separates the content for the menu from your app's behavioral code.
- It lets you create alternative menu configurations for different platform versions, screen sizes, and other configurations by leveraging the app resources framework.

To define a menu, create an XML file inside your project's res/menu/ directory and build the menu with the following elements:

<menu>

Defines a Menu, which is a container for menu items. A <menu> element must be the root node for the file, and it can hold one or more <item> and <group> elements.

<item>

Creates a MenuItem, which represents a single item in a menu. This element can contain a nested <menu> element to create a submenu.

<group>

An optional, invisible container for <item> elements. It lets you categorize menu items so they share properties, such as active state and visibility. For more information, see the Create a menu group section.

Here's an example menu named game_menu.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
   <item android:id="@+id/new_game"
        android:icon="@drawable/ic_new_game"
        android:title="@string/new_game"
        app:showAsAction="ifRoom"/>
   <item android:id="@+id/help"
        android:icon="@drawable/ic_help"
        android:title="@string/help" />
</menu>
```

The <item> element supports several attributes you can use to define an item's appearance and behavior. The items in the preceding menu include the following attributes:

android:id

A resource ID that's unique to the item, which lets the app recognize the item when the user selects it.

android:icon

A reference to a drawable to use as the item's icon.

android:title

A reference to a string to use as the item's title.

android:showAsAction

The specification for when and how this item appears as an action item in the app bar.

These are the most important attributes you use, but there are many more available. For information about all the supported attributes, see the Menu resource documentation.

You can add a submenu to an item in any menu by adding a <menu> element as the child of an <item>. Submenus are useful when your app has a lot of functions that can be organized into topics, like items in a PC app's menu bar—such as **File**, **Edit**, and **View**. See the following example:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
   <item android:id="@+id/file"
        android:title="@string/file" >
      <!-- "file" submenu -->
      <menu>
         <item android:id="@+id/create_new"
              android:title="@string/create_new" />
         <item android:id="@+id/open"
              android:title="@string/open" />
      </menu>
   </item>
</menu>
```

To use the menu in your activity, _inflate_ the menu resource, converting the XML resource into a programmable object using MenuInflater.inflate(). The following sections show how to inflate a menu for each menu type.

Create an options menu

The options menu, like the one shown in figure 1, is where you include actions and other options that are relevant to the current activity context, such as "Search," "Compose email," and "Settings."

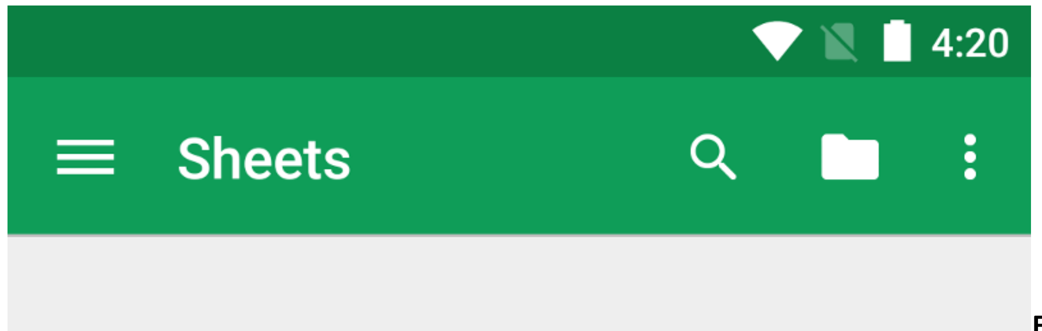


Figure 2. The Google Sheets app, showing several buttons, including the action overflow button.

You can declare items for the options menu from your Activity subclass or a Fragment subclass. If both your activity and your fragments declare items for the options menu, the items are combined in the UI. The activity's items appear first, followed by those of each fragment, in the order in which the fragments are added to the activity. If necessary, you can reorder the menu items with the android:orderInCategory attribute in each <item> you need to move.

To specify the options menu for an activity, override onCreateOptionsMenu(). Fragments provide their own onCreateOptionsMenu() callback. In this method, you can inflate your menu resource, defined in XML, into the Menu provided in the callback. This is shown in the following example:

KotlinJava

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
    val inflater: MenuInflater = menuInflater
    inflater.inflate(R.menu.game_menu, menu)
    return true
}
```

You can also add menu items using add() and retrieve items with findItem() to revise their properties with MenuItem APIs.

Handle click events

When the user selects an item from the options menu, including action items in the app bar, the system calls your activity's onOptionsItemSelected() method. This method passes the MenuItem selected. You can identify the item by calling getItemId(), which returns the unique ID for the menu item, defined by the android:id attribute in the menu resource or with an integer given to the add() method. You can match this ID against known menu items to perform the appropriate action.

KotlinJava

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    // Handle item selection.
    return when (item.itemId) {
        R.id.new_game -> {
            newGame()
            true
        }
        R.id.help -> {
            showHelp()
            true
        }
        else -> super.onOptionsItemSelected(item)
    }
}
```

When you successfully handle a menu item, return true. If you don't handle the menu item, call the superclass implementation of onOptionsItemSelected(). The default implementation returns false.

If your activity includes fragments, the system first calls onOptionsItemSelected() for the activity, then for each fragment in the order the fragments are added, until one returns true or all fragments are called.

Tip: If your app contains multiple activities and some of them provide the same options menu, consider creating an activity that implements only the **onCreateOptionsMenu()** and **onOptionsItemSelected()** methods. Then extend this class for each activity that shares the same options menu. This way, you can manage one set of code for handling menu actions, and each descendant class inherits the menu behaviors. If you want to add menu items to one of the descendant activities, override **>onCreateOptionsMenu()** in that activity. Call **super.onCreateOptionsMenu(menu)** so the original menu items are created, then add new menu items with **menu.add()**. You can also override the superclass's behavior for individual menu items.

Change menu items at runtime

After the system calls onCreateOptionsMenu(), it retains an instance of the Menu you populate and doesn't call onCreateOptionsMenu() again unless the menu is invalidated. However, use onCreateOptionsMenu() only to create the initial menu state and not to make changes during the activity lifecycle.

If you want to modify the options menu based on events that occur during the activity lifecycle, you can do so in the onPrepareOptionsMenu() method. This method passes you the Menu object as it currently exists so you can modify it, such as by adding, removing, or disabling items. Fragments also provide an onPrepareOptionsMenu() callback.

The options menu is considered always open when menu items are presented in the app bar. When an event occurs and you want to perform a menu update, call invalidateOptionsMenu() to request that the system call onPrepareOptionsMenu().

Note: Never change items in the options menu based on the **View** in focus. When in touch mode—when the user isn't using a trackball or D-pad—views can't take focus, so never use focus as the basis for modifying items in the options menu. If you want to provide menu items that are context-sensitive to a **View**, use a contextual menu as described in the following section.

Create a contextual menu

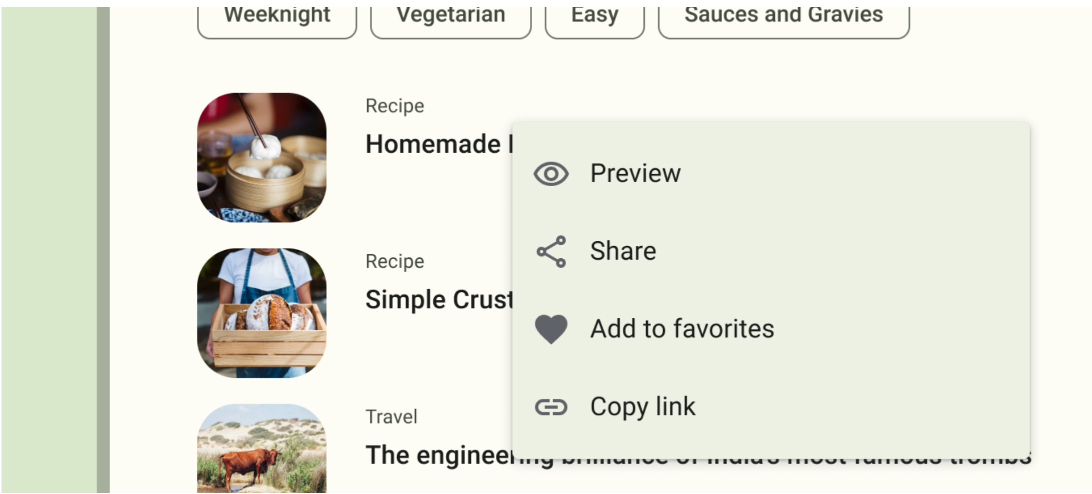


Figure 3. A floating context menu.

A contextual menu offers actions that affect a specific item or context frame in the UI. You can provide a context menu for any view, but they are most often used for items in a RecylerView or other view collections in which the user can perform direct actions on each item.

There are two ways to provide contextual actions:

- In a floating context menu. A menu appears as a floating list of menu items, similar to a dialog, when the user performs a touch & hold on a view that declares support for a context menu. Users can perform a contextual action on one item at a time.
- In the contextual action mode. This mode is a system implementation of ActionMode that displays a *contextual action bar*, or CAB, at the top of the screen with action items that affect the selected item(s). When this mode is active, users can perform an action on multiple items at once, if your app supports that.

Create a floating context menu

To provide a floating context menu, do the following:

1. Register the View the context menu is associated with by calling registerForContextMenu() and passing it the View.

If your activity uses a RecyclerView and you want each item to provide the same context menu, register all items for a context menu by passing the RecyclerView to registerForContextMenu().

2. Implement the onCreateContextMenu() method in your Activity or Fragment.

When the registered view receives a touch & hold event, the system calls your onCreateContextMenu() method. This is where you define the menu items, usually by inflating a menu resource, as in the following example:

KotlinJava

```
override fun onCreateContextMenu(menu: ContextMenu, v: View,
                        menuInfo: ContextMenu.ContextMenuInfo) {
    super.onCreateContextMenu(menu, v, menuInfo)
    val inflater: MenuInflater = menuInflater
    inflater.inflate(R.menu.context_menu, menu)
```

```
}
```

MenuInflater lets you inflate the context menu from a menu resource. The callback method parameters include the View that the user selects and a ContextMenu.ContextMenuInfo object that provides additional information about the item selected. If your activity has several views that each provide a different context menu, you might use these parameters to determine which context menu to inflate.

3. Implement onContextItemSelected(), as shown in the following example. When the user selects a menu item, the system calls this method so you can perform the appropriate action.

KotlinJava

```
override fun onContextItemSelected(item: MenuItem): Boolean {
    val info = item.menuInfo as AdapterView.AdapterContextMenuInfo
    return when (item.itemId) {
        R.id.edit -> {
            editNote(info.id)
            true
        }
        R.id.delete -> {
            deleteNote(info.id)
            true
        }
        else -> super.onContextItemSelected(item)
    }
}
```

The getItemId() method queries the ID for the selected menu item, which you assign to each menu item in XML using the android:id attribute, as shown in Define a menu in XML.

When you successfully handle a menu item, return true. If you don't handle the menu item, pass the menu item to the superclass implementation. If your activity includes fragments, the activity receives this callback first. By calling the superclass when unhandled, the system passes the event to the respective callback method in each fragment, one at a time, in the order each fragment is added, until true or false is returned. The default implementations for Activity and android.app.Fragment return false, so always call the superclass when unhandled.

Use the contextual action mode

The contextual action mode is a system implementation of ActionMode that focuses user interaction toward performing contextual actions. When a user enables this mode by selecting an item, a *contextual action bar* appears at the top of the screen to present actions the user can perform on the selected items. While this mode is enabled, the user can select multiple items, if your app supports that, and can deselect items and continue to navigate within the activity. The action mode is disabled and the contextual action bar disappears when the user deselects all items, taps the Back button, or taps the **Done** action on the left side of the bar.

Note: The contextual action bar isn't necessarily associated with the app bar. They operate independently, although the contextual action bar visually overtakes the app bar position.

For views that provide contextual actions, you usually invoke the contextual action mode when one or both of these two events occurs:

- The user performs a touch & hold on the view.
- The user selects a checkbox or similar UI component within the view.

How your app invokes the contextual action mode and defines the behavior for each action depends on your design. There are two designs:

- For contextual actions on individual, arbitrary views.
- For batch contextual actions on groups of items in a RecyclerView, letting the user select multiple items and perform an action on them all.

The following sections describe the setup required for each scenario.

Enable the contextual action mode for individual views

If you want to invoke the contextual action mode only when the user selects specific views, do the following:

1. Implement the ActionMode.Callback interface as shown in the following example. In its callback methods, you can specify the actions for the contextual action bar, respond to click events on action items, and handle other lifecycle events for the action mode.

KotlinJava

```
private val actionModeCallback = object : ActionMode.Callback {
    // Called when the action mode is created. startActionMode() is called.
    override fun onCreateActionMode(mode: ActionMode, menu: Menu): Boolean {
        // Inflate a menu resource providing context menu items.
        val inflater: MenuInflater = mode.menuInflater
        inflater.inflate(R.menu.context_menu, menu)
        return true
    }

    // Called each time the action mode is shown. Always called after
    // onCreateActionMode, and might be called multiple times if the mode
    // is invalidated.
    override fun onPrepareActionMode(mode: ActionMode, menu: Menu): Boolean {
        return false // Return false if nothing is done
    }

    // Called when the user selects a contextual menu item.
    override fun onActionItemClicked(mode: ActionMode, item: MenuItem): Boolean {
        return when (item.itemId) {
            R.id.menu_share -> {
                shareCurrentItem()
                mode.finish() // Action picked, so close the CAB.
```

```
            true
          }
          else -> false
        }
      }

      // Called when the user exits the action mode.
      override fun onDestroyActionMode(mode: ActionMode) {
        actionMode = null
      }
    }
```

These event callbacks are almost exactly the same as the callbacks for the options menu, except that each of these also passes the ActionMode object associated with the event. You can use ActionMode APIs to make various changes to the CAB, such as revising the title and subtitle with setTitle() and setSubtitle(), which is useful to indicate how many items are selected.

The preceding sample sets the actionMode variable to null when the action mode is destroyed. In the next step, see how it's initialized and how saving the member variable in your activity or fragment can be useful.

2. Call startActionMode() when you want to show the bar, such as when the user performs a touch & hold on the view.

KotlinJava

```
someView.setOnLongClickListener { view ->
  // Called when the user performs a touch & hold on someView.
  when (actionMode) {
    null -> {
      // Start the CAB using the ActionMode.Callback defined earlier.
      actionMode = activity?.startActionMode(actionModeCallback)
      view.isSelected = true
      true
    }
    else -> false
  }
}
```

When you call startActionMode(), the system returns the ActionMode created. By saving this in a member variable, you can make changes to the contextual action bar in response to other events. In the preceding sample, the ActionMode is used to ensure that the ActionMode instance isn't recreated if it's already active, by checking whether the member is null before starting the action mode.

Create a popup menu

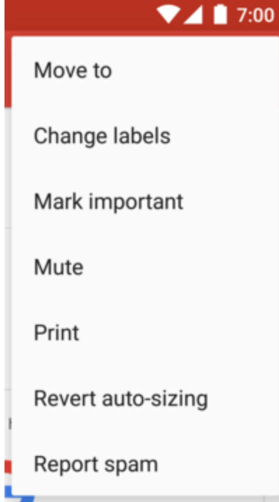


Figure 4. A popup menu in the Gmail app, anchored to the overflow button in the top-right corner.

A PopupMenu is a modal menu anchored to a View. It appears below the anchor view if there is room, or above the view otherwise. It's useful for the following:

- Providing an overflow-style menu for actions that *relate to* specific content, such as Gmail's email headers, shown in figure 4.**Note:** This isn't the same as a context menu, which is generally for actions that *affect* selected content. For actions that affect selected content, use the contextual action mode or floating context menu.
- Providing a second part of a command sentence, such as a button marked **Add** that produces a popup menu with different **Add** options.
- Providing a menu similar to a Spinner that doesn't retain a persistent selection.

If you define your menu in XML, here's how you can show the popup menu:

1. Instantiate a PopupMenu with its constructor, which takes the current app Context and the View to which the menu is anchored.
2. Use MenuInflater to inflate your menu resource into the Menu object returned by PopupMenu.getMenu().
3. Call PopupMenu.show().

For example, here's a button that shows a popup menu:

```
<ImageButton
    android:id="@+id/dropdown_menu"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:contentDescription="@string/descr_overflow_button"
    android:src="@drawable/arrow_drop_down" />
```

The activity can then show the popup menu like this:

KotlinJava

```
findViewById<ImageButton>(R.id.dropdown_menu).setOnClickListener {
    val popup = PopupMenu(this, it)
    val inflater: MenuInflater = popup.menuInflater
    inflater.inflate(R.menu.actions, popup.menu)
    popup.show()
}
```

The menu is dismissed when the user selects an item or taps outside the menu area. You can listen for the dismiss event using PopupMenu.OnDismissListener.

Handle click events

To perform an action when the user selects a menu item, implement the PopupMenu.OnMenuItemClickListener interface and register it with your PopupMenu by calling setOnMenuItemclickListener(). When the user selects an item, the system calls the onMenuItemClick() callback in your interface.

This is shown in the following example:

KotlinJava

```
fun showMenu(v: View) {
    PopupMenu(this, v).apply {
        // MainActivity implements OnMenuItemClickListener.
        setOnMenuItemClickListener(this@MainActivity)
        inflate(R.menu.actions)
        show()
    }
}

override fun onMenuItemClick(item: MenuItem): Boolean {
    return when (item.itemId) {
        R.id.archive -> {
            archive(item)
            true
        }
        R.id.delete -> {
            delete(item)
            true
        }
        else -> false
    }
}
```

Create a menu group

A menu group is a collection of menu items that share certain traits. With a group, you can do the following:

- Show or hide all items using setGroupVisible().

- Enable or disable all items using setGroupEnabled().
- Specify whether all items are checkable using setGroupCheckable().

You can create a group by nesting <item> elements inside a <group> element in your menu resource or by specifying a group ID with the add() method.

Here's an example of a menu resource that includes a group:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/menu_save"
      android:icon="@drawable/menu_save"
      android:title="@string/menu_save" />
  <!-- menu group -->
  <group android:id="@+id/group_delete">
    <item android:id="@+id/menu_archive"
        android:title="@string/menu_archive" />
    <item android:id="@+id/menu_delete"
        android:title="@string/menu_delete" />
  </group>
</menu>
```

The items that are in the group appear at the same level as the first item—all three items in the menu are siblings. However, you can modify the traits of the two items in the group by referencing the group ID and using the preceding methods. The system also never separates grouped items. For example, if you declare android:showAsAction="ifRoom" for each item, they both appear in the action bar or both appear in the action overflow.

Use checkable menu items

Figure 5. A submenu with checkable items.

A menu can be useful as an interface for turning options on and off, using a checkbox for standalone options, or radio buttons for groups of mutually exclusive options. Figure 5 shows a submenu with items that are checkable with radio buttons.

Note: Menu items in an options menu can't display a checkbox or radio button. If you make items in an options menu checkable, manually indicate the checked state by swapping the icon or text, or both, each time the state changes.

You can define the checkable behavior for individual menu items using the android:checkable attribute in the <item> element, or for an entire group with the android:checkableBehavior attribute in the <group> element. For example, all items in this menu group are checkable with a radio button:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <group android:checkableBehavior="single">
    <item android:id="@+id/red"
        android:title="@string/red" />
    <item android:id="@+id/blue"
```

```
        android:title="@string/blue" />
    </group>
</menu>
```

The android:checkableBehavior attribute accepts one of the following:

single

Only one item from the group can be checked, resulting in radio buttons.

all

All items can be checked, resulting in checkboxes.

none

No items are checkable.

You can apply a default checked state to an item using the android:checked attribute in the <item> element and change it in code with the setChecked() method.

When a checkable item is selected, the system calls your respective item-selected callback method, such as onOptionsItemSelected(). This is where you set the state of the checkbox, because a checkbox or radio button doesn't change its state automatically. You can query the current state of the item—as it was before the user selected it—with isChecked() and then set the checked state with setChecked(). This is shown in the following example:

KotlinJava

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {
        R.id.vibrate, R.id.dont_vibrate -> {
            item.isChecked = !item.isChecked
            true
        }
        else -> super.onOptionsItemSelected(item)
    }
}
```

If you don't set the checked state this way, then the visible state of the checkbox or radio button doesn't change when the user selects it. When you do set the state, the activity preserves the checked state of the item so that when the user opens the menu later, the checked state that you set is visible.

Day 3-Lab Activity: Working with Navigation and Menus

Navigation Basics:

The essentials of navigation and menus in Android development. This includes setting up fragment managers, navigating between screens, and creating both standard and contextual menus.

Setting up Fragment Managers (Android):

In Android, FragmentManager is responsible for handling fragments and fragment transactions. It manages the addition, removal, and replacement of fragments within an activity.

```
val fragmentManager = supportFragmentManager
val fragmentTransaction = fragmentManager.beginTransaction()

// Adding a fragment
val fragment = MyFragment()
fragmentTransaction.add(R.id.fragment_container, fragment)
fragmentTransaction.commit()
```

Navigating between different screens or views.

Navigation involves moving between different destinations in an app, such as activities or fragments. This is typically handled using a navigation component or manually with intents.

```
val navController = findNavController(R.id.nav_host_fragment)
navController.navigate(R.id.action_firstFragment_to_secondFragment)
```

Creating Menus:

Menus in Android provide a way for users to perform actions or navigate within the app. They can be implemented in the app's toolbar or as overflow menus.

1. **Create a Menu Resource**:

Create a new XML file in the res/menu directory (e.g., menu_main.xml) and define the menu items.

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".MainActivity">

    <item
        android:id="@+id/action_search"
        android:icon="@drawable/ic_search"
        android:title="Search"
        app:showAsAction="always" />
    <!-- Add more items as needed -->
</menu>
```

1. **Inflate the Menu in Your Activity:**

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
    menuInflater.inflate(R.menu.menu_main, menu)
    return true
}
```

3. Handle Menu Item Clicks:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.action_search -> {
            // Handle search action
            return true
        }
        // Add more cases for other menu items as needed
        else -> return super.onOptionsItemSelected(item)
    }
```

Day 4: Advanced UI Components (Optional)

Using Advanced UI Components:

Implementing features like image galleries, maps, etc.

Create an Android app that displays a map by using the Google Maps template for Android Studio. If you have an existing Android Studio project that you'd like to set up, see Project Configuration.

Set up the development environment

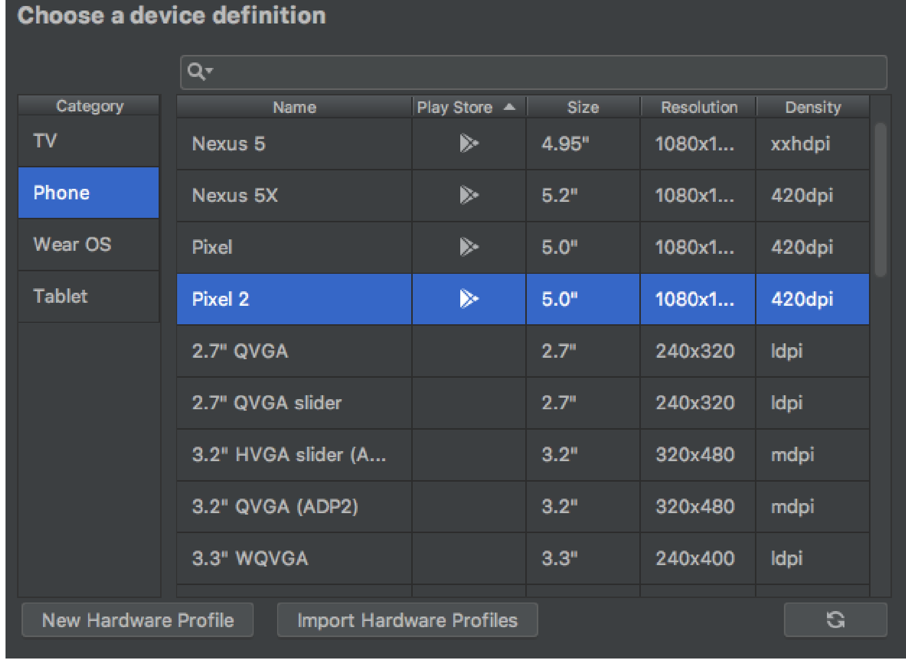
Android Studio Arctic Fox or later is required. If you haven't already done so, download and install it. Ensure that you are using the Android Gradle plugin version 7.0 or later in Android Studio.

Set up an Android device

To run an app that uses the Maps SDK for Android, you must deploy it to an Android device or Android emulator that is based on Android 4.0 or higher and includes the Google APIs.

To use an Android device, follow the instructions at Run apps on a hardware device.

To use an Android emulator, you can create a virtual device and install the emulator by using the Android Virtual Device (AVD) Manager that comes with Android Studio.**Note:** If you choose to use an Android emulator, ensure that you choose a device with the **Play icon**, , displayed under the **Play Store** column. This icon indicates that these profiles are fully CTS compliant and may use system images that include the Play Store app:



Create a Google Maps project in Android Studio

The procedure to create a Google Maps project in Android Studio was changed in the Flamingo and later releases of Android Studio. Make sure to follow the steps below for your specific version of Android Studio.

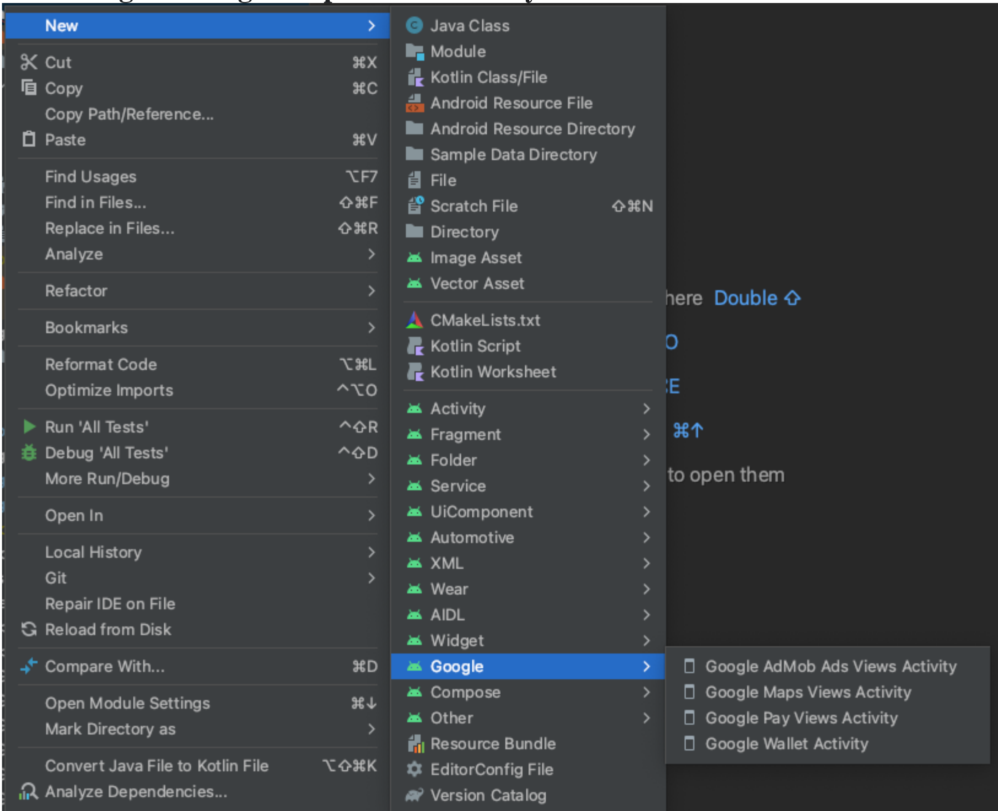
Open Android Studio, and click **Create New Project** in the **Welcome to Android Studio** window.

In the **New Project** window, under the **Phone and Tablet** category, select the **Empty Activity**, and then click **Next**.

Note: If you are using Android Studio Electric Eel or earlier, select the **Google Maps Activity** option.

Complete the **Google Maps Activity** form:

Set **Language** to Java or Kotlin. Both languages are fully supported by the Maps SDK for Android. To learn more about Kotlin, see Develop Android apps with Kotlin.
Set **Minimum SDK** to an SDK version compatible with your test device. You must select a version greater than the minimum version required by the Maps SDK for Android version 18.0.x, which is Android API Level 19 (Android 4.4, KitKat) or higher. See the Release Notes for the latest information on the SDK version requirements.
Note: If you are unsure of what SDK version to choose, select the **Help me choose** link to display distribution information for supported Android SDKs. For example, if you choose **API 19: Android 4.4 (KitKat)**, your app should run on over 99% of Android devices.
Click **Finish**.
Android Studio starts Gradle and builds the project. This may take some time.
When the build is finished, Android Studio opens the AndroidManifest.xml and MapsActivity files. Your activity may have a different name, but it is the one you configured during setup.
Add the **Google Maps Views Activity**:
Note: Skip this step if you are using **Android Studio Electric Eel or earlier** because you selected the **Google Maps Activity** in step 2.
Right-click on the package where you would like to add the Google Maps Views Activity.
Select **New > Google > Google Maps Views Activity**.



For more information, see Add code from a template
The AndroidManifest.xml file contains instructions on getting a Google Maps API key and then adding it to your local.properties file. **Do not add your API key to the AndroidManifest.xml file.** Doing so stores your API key less securely. Instead, follow the instructions in the next sections to create a Cloud project and configure an API key.
Set up your Google Cloud project
Complete the required Cloud Console setup steps by clicking through the following tabs:
In the Google Cloud Console, on the project selector page, click **Create Project** to begin creating a new Cloud project.
Go to the project selector page
Make sure that billing is enabled for your Cloud project. Confirm that billing is enabled for your project.

Google Cloud offers a $0.00 charge trial. The trial expires at either end of 90 days or after the account has accrued $300 worth of charges, whichever comes first. Cancel anytime. Google Maps Platform features a recurring $200 monthly credit. For more information, see Billing account credits and Billing.
Note: If you don't plan to keep the resources that you create in a learning exercise, create a project instead of selecting an existing project. After you finish the exercise, you can delete the project, removing all resources associated with the project.

Add the API key to your app

This section describes how to store your API key so that it can be securely referenced by your app. You should not check your API key into your version control system, so we recommend storing it in the local.properties file, which is located in the root directory of your project. For more information about the local.properties file, see Gradle properties files.

To streamline this task, we recommend that you use the Secrets Gradle Plugin for Android. To install the plugin and store your API key:

In Android Studio, open your project-level build.gradle file and add the following code to the dependencies element under buildscript.

```
buildscript {
    dependencies {
        classpath "com.google.android.libraries.mapsplatform.secrets-gradle-plugin:secrets-gradle-
plugin:2.0.1"
    }
}
```

Next, open your module-level build.gradle file and add the following code to the plugins element.

```
plugins {
    // ...
    id 'com.google.android.libraries.mapsplatform.secrets-gradle-plugin'
}
```

Save the file and sync your project with Gradle.

Open the local.properties in your project level directory, and then add the following code.
Replace YOUR_API_KEY with your API key.

```
MAPS_API_KEY=YOUR_API_KEY
```

Save the file.

In your AndroidManifest.xml file, go to com.google.android.geo.API_KEY and update the android:value attribute as follows:

```
<meta-data
    android:name="com.google.android.geo.API_KEY"
    android:value="${MAPS_API_KEY}" />
```

Note: As shown above, **com.google.android.geo.API_KEY** is the recommended metadata name for the API key. A key with this name can be used to authenticate to multiple Google Maps-based APIs on the Android platform, including the Maps SDK for Android. For backwards compatibility, the API also supports the name **com.google.android.maps.v2.API_KEY**. This legacy name allows authentication to the Android Maps API v2 only. An application can specify only one of the API key metadata names. If both are specified, the API throws an exception.

Look at the code

Examine the code supplied by the template. In particular, look at the following files in your Android Studio project.

Maps activity file

The maps activity file is the main activity for the app, and contains the code to manage and display the map. By default, the file that defines the activity is named MapsActivity.java or if you set Kotlin as the language for your app, MapsActivity.kt.

The main elements of the maps activity:

The SupportMapFragment object manages the life cycle of the map and is the parent element of the app's UI.

The GoogleMap object provides access to the map data and view. This is the main class of the Maps SDK for Android. The Map Objects guide describes the SupportMapFragment and GoogleMap objects in more detail.
The moveCamera function centers the map at the LatLng coordinates for Sydney Australia. The first settings to configure when adding a map are usually the map location and camera settings; such as viewing angle, map orientation, and zoom level. See the Camera and View guide for details.
The addMarker function adds a marker to the coordinates for Sydney. See the Markers guide for details.
The maps activity file contains the following code:
JavaKotlin

```
import android.os.Bundle;
import androidx.appcompat.app.AppCompatActivity;
import com.google.android.gms.maps.CameraUpdateFactory;
import com.google.android.gms.maps.GoogleMap;
import com.google.android.gms.maps.OnMapReadyCallback;
import com.google.android.gms.maps.SupportMapFragment;
import com.google.android.gms.maps.model.LatLng;
import com.google.android.gms.maps.model.MarkerOptions;

public class MapsActivity extends AppCompatActivity implements OnMapReadyCallback {

    private GoogleMap mMap;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_maps);
        // Obtain the SupportMapFragment and get notified when the map is ready to be used.
        SupportMapFragment mapFragment = (SupportMapFragment) getSupportFragmentManager()
                .findFragmentById(R.id.map);
        mapFragment.getMapAsync(this);
    }

    /**
     * Manipulates the map once available.
     * This callback is triggered when the map is ready to be used.
     * This is where we can add markers or lines, add listeners or move the camera. In this case,
     * we just add a marker near Sydney, Australia.
     *
     * If Google Play services is not installed on the device, the user will be prompted to install
     * it inside the SupportMapFragment. This method will only be triggered once the user has
     * installed Google Play services and returned to the app.
     */
    @Override
    public void onMapReady(GoogleMap googleMap) {
        mMap = googleMap;

        // Add a marker in Sydney and move the camera
        LatLng sydney = new LatLng(-34, 151);
        mMap.addMarker(new MarkerOptions()
                .position(sydney)
                .title("Marker in Sydney"));
        mMap.moveCamera(CameraUpdateFactory.newLatLng(sydney));
    }
}
```

Module Gradle file

The Module build.gradle file includes the following maps dependency, which is required by the Maps SDK for Android.

```
dependencies {
    implementation 'com.google.android.gms:play-services-maps:18.1.0'
    // ...
}
```

Note: The exact version can change based on the current version of the Maps SDK for Android.

To learn more about managing the Maps dependency, see Versioning.

XML layout file

The activity_maps.xml file is the XML layout file that defines the structure of the app's UI. The file is located in the res/layout directory. The activity_maps.xml file declares a fragment that includes the following elements:

tools:context sets the default activity of the fragment to MapsActivity, which is defined in the maps activity file.

android:name sets the class name of the fragment to SupportMapFragment, which is the fragment type used in the maps activity file.

The XML layout file contains the following code:

```
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/map"
    tools:context=".MapsActivity"
    android:name="com.google.android.gms.maps.SupportMapFragment" />
```

Deploy and run the app

When you run the app successfully, it will display a map that is centered on Sydney Australia with a marker on the city as seen in the following screenshot.
To deploy and run the app:
In Android Studio, click the **Run** menu option (or the play button icon) to run your app.
When prompted to choose a device, choose one of the following options:
Select the Android device that's connected to your computer.
Alternatively, select the **Launch emulator** radio button and choose the virtual device that you set up.
Click **OK**. Android Studio will start Gradle to build your app, and then display the results on your device or emulator. It can take several minutes before the app launches.
Next steps
Set up a map: This topic describes how to set up the initial and runtime settings for your map, such as the camera position, map type, UI components, and gestures.
Add a map to your Android app (Kotlin): This codelab walks you through an app that demonstrates some additional features of the Maps SDK for Android.
Use the Maps Android KTX library: This Kotlin extensions (KTX) library allows you to take advantage of several Kotlin language features while using the Maps SDK for Android.

Advanced UI components to enhance

Advanced UI components to enhance the user experience in your Android app. These components offer specialized functionalities and can greatly enrich the interactivity and visual appeal of your application.

RecyclerView:

Description: RecyclerView is a flexible view group that efficiently displays large sets of data. It's commonly used for displaying lists or grids of items.

Code Example (Kotlin):

```
// In your activity/fragment
val recyclerView = findViewById<RecyclerView>(R.id.recycler_view)
recyclerView.layoutManager = LinearLayoutManager(this)
val adapter = MyAdapter(dataList)
recyclerView.adapter = adapter
```

DatePickerDialog:

Description: DatePickerDialog provides a dialog for selecting a date.

Code Example (Kotlin):

```
// In your activity/fragment
val calendar = Calendar.getInstance()
val year = calendar.get(Calendar.YEAR)
val month = calendar.get(Calendar.MONTH)
val day = calendar.get(Calendar.DAY_OF_MONTH)

val datePickerDialog = DatePickerDialog(this, { _, selectedYear, selectedMonth,
selectedDay ->
    // Handle the selected date
}, year, month, day)

// Show the dialog when needed
datePickerDialog.show()
```

TimePickerDialog:

Description: TimePickerDialog provides a dialog for selecting a time.

Code Example (Kotlin):

```
// In your activity/fragment
val calendar = Calendar.getInstance()
val hour = calendar.get(Calendar.HOUR_OF_DAY)
val minute = calendar.get(Calendar.MINUTE)

val timePickerDialog = TimePickerDialog(this, { _, selectedHour,
selectedMinute ->
    // Handle the selected time
}, hour, minute, true)
```

Implementing features like image galleries, maps, etc.

Implementing features like image galleries and maps can greatly enhance the functionality and user experience of your Android app. Here's how you can go about it:

Implementing an Image Gallery:

1. Add Permissions:

Ensure you have the necessary permissions in your AndroidManifest.xml file to access external storage (if required).

<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />

2.Load Images:

Use a library like Glide or Picasso to efficiently load and display images from the device's storage or a remote server.

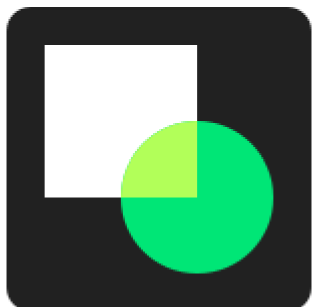
```
// In your activity/fragment

Glide.with(this)

   .load("https://example.com/image.jpg")

   .into(imageView)
```

3.Implement a Gallery View:

You can use a RecyclerView or a ViewPager to create a gallery-like interface for navigating through images.

Handle User Interactions:

Set up click listeners to allow users to interact with the images (e.g., view in full screen, open in a separate activity, etc.).



Material Components (MDC) help developers implement Material Design. Created by a team of engineers and UX designers at Google, MDC features dozens of beautiful and functional UI components and is available for Android, iOS, web and Flutter.material.io/develop

---

In codelab MDC-103, you customized the color, elevation, and typography, of Material Components (MDC) to style your app.

A component in the Material Design system performs a set of predefined tasks and has certain characteristics, like a button. However, a button is more than just a way for a user to perform an action, it's also a visual expression of shape, size, and color that lets the user know that it's interactive, and that something will happen upon touch or click.

The Material Design guidelines describe components from a designer's point of view. They describe a wide range of basic functions available across platforms, as well as the anatomic elements that make up each component. For instance, a backdrop contains a back layer and its content, the front layer and its content, motion rules, and display options. Each of these components can be customized for each app's needs, use cases, and content. These pieces are, for the most part, traditional views, controls, and functions from your platform's SDK.

While the Material Design guidelines name many components, not all of them are good candidates for reusable code and therefore aren't found in MDC. You can create these experiences yourself to achieve a customized style for your app, all using traditional code.

Day 5: Practice and Review

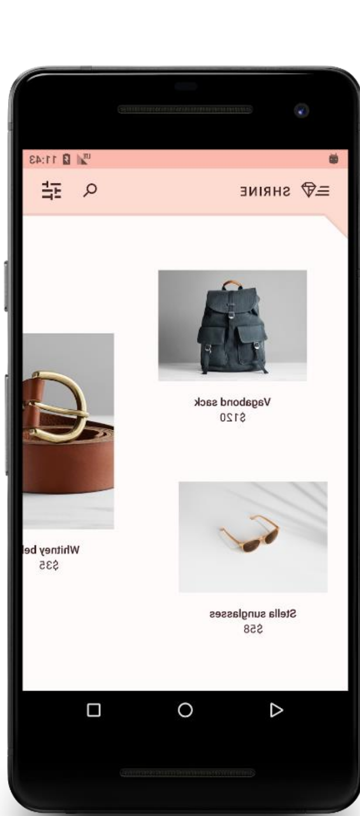
Project Activity:

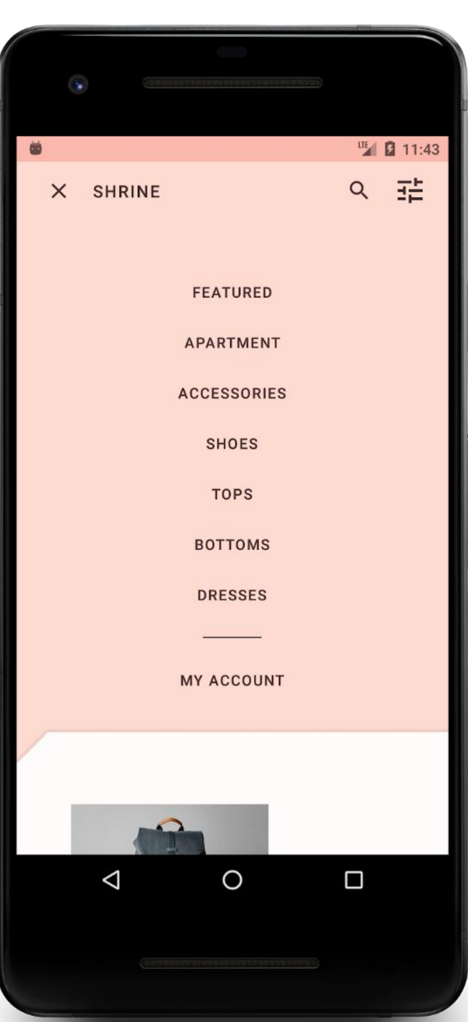
Building a small app with various UI components, input handling, and navigation.

Example for UI Components

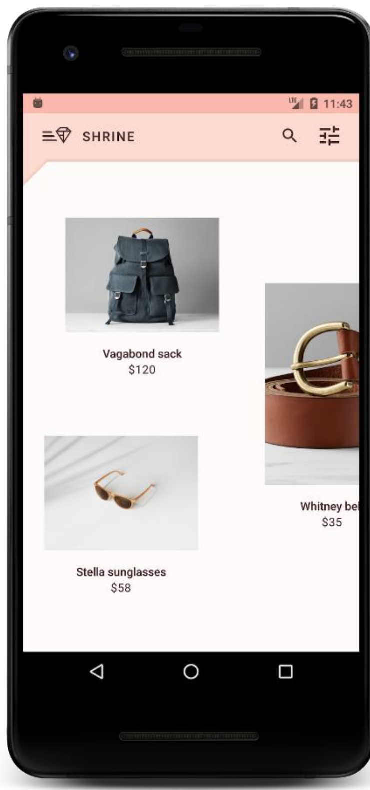
In this codelab, you'll add a backdrop to **Shrine**. It will filter the products shown in the asymmetrical grid by category. You'll use:

- Shape
- Motion
- Traditional Android SDK classes





This is the last of 4 codelabs that guide you through building an app for a product called Shrine. We recommend that you do all of the codelabs in order as they progress through tasks step-by-step.The related codelabs can be found at:



MDC-Android components in this codelab

- Shape

What you'll need

- Basic knowledge of Android development
- **Android Studio**
- An Android emulator or device (available through Android Studio)
- The sample code you may add according to your project requirements

Week 6 – Working with Data

Week 6 – Working with Data

Day 1: Introduction to Data Handling

Understanding Data in Apps:

Data handling in Android refers to the process of managing and manipulating data within Android applications. Android applications often need to deal with various types of data, including user inputs, database entries, network responses, and more. Efficient data handling is crucial for creating responsive and user-friendly applications.

Different types of data (e.g., text, images, user inputs).

Here's an overview of key aspects of data handling in Android:

User Inputs: Android applications receive data from users through various UI elements like text fields, buttons, checkboxes, etc. This data needs to be captured, validated, and processed.

Persistence: Android apps often need to store data so that it can be accessed later, even after the app is closed. This can be achieved through different mechanisms such as SharedPreferences for simple key-value pairs, SQLite databases for structured data, or external storage for files.

Networking: Many Android apps interact with remote servers to fetch or send data. This involves making HTTP requests, handling responses, and processing data received from the server.

Content Providers: Android provides Content Providers as a mechanism for sharing data between different applications. Content Providers allow applications to securely access data stored by other applications.

AsyncTasks and Threads: Since Android applications run on a single main thread, long-running operations like network requests should be performed in separate threads or AsyncTask to prevent blocking the UI and provide a smoother user experience.

Data Validation and Sanitization: Data received from users or external sources should be validated and sanitized to prevent issues like SQL injection, code injection, or other security vulnerabilities.

Data Binding: Android's data binding library allows for easier and more efficient connection between UI components and the data source. It helps automate the process of keeping UI elements updated with the latest data.

Serialization and Deserialization: When passing data between different components of an app or between different apps, data needs to be serialized (converted into a format that can be easily transmitted) and deserialized (converted back to its original form).

Caching: Caching allows for storing frequently accessed data in memory or on disk to reduce the need for expensive operations like network requests or complex computations.

Data Security and Encryption: Sensitive data should be handled with care and, if necessary, encrypted to protect it from unauthorized access.

Importance of efficient data management.

Efficient data management is of paramount importance in various domains and industries due to its numerous benefits and impacts on operations, decision-making, compliance, and overall organizational success. Here are some key reasons why efficient data management is crucial:

Improved Decision-Making: Accurate and timely data enables organizations to make informed decisions. It provides the foundation for analytics, business intelligence, and data-driven insights, allowing for better strategic planning and resource allocation.

Enhanced Productivity: Well-organized data systems and efficient data retrieval processes save time and effort. This leads to increased productivity as employees can quickly access the information, they need to perform their tasks.

Cost Efficiency: Efficient data management reduces costs associated with data storage, processing, and maintenance. Unnecessary duplication, storage of outdated information, and inefficient data handling practices can be minimized.

Compliance and Regulatory Requirements: Many industries have strict regulations regarding data handling and privacy (e.g., GDPR in Europe, HIPAA in healthcare). Efficient data management ensures compliance with these regulations, reducing legal and financial risks.

Customer Satisfaction: Effective data management allows organizations to provide better services to their customers. Timely access to accurate information leads to faster responses to inquiries and improved customer experiences.

Data Security and Privacy: Proper data management includes security measures to protect sensitive information from unauthorized access, breaches, or cyber threats. This helps build trust with customers and partners.

Risk Management: Efficient data management practices help in identifying and mitigating risks associated with data loss, corruption, or unauthorized access. This is crucial for business continuity and disaster recovery planning.

Innovation and Research: Well-maintained data repositories facilitate research and innovation. Scientists, analysts, and researchers can access high-quality data for analysis and experimentation, driving innovation within the organization.

Competitive Advantage: Organizations that effectively manage their data have a competitive edge. They can respond more quickly to market changes, customer demands, and emerging trends based on real-time insights.

Business Continuity and Disaster Recovery: Properly managed data includes backup and recovery strategies. In the event of a system failure, natural disaster, or cyber-attack, organizations can recover critical data to ensure business continuity.

Efficient Collaboration: When data is organized and easily accessible, it fosters collaboration among team members and departments. It allows for seamless sharing and integration of information across the organization.

Scalability and Growth: As organizations grow, so does the volume of data they handle. Efficient data management ensures that systems can scale to accommodate increased data loads without sacrificing performance.

Basic Data Storage:

Introduction to key-value pairs, file storage, etc.

Storing data locally on a device is a fundamental aspect of application development. It allows apps to retain information even when they are closed or the device is restarted. Here are some basic methods for local data storage:

Key-Value Pairs:

Overview: This is one of the simplest ways to store small pieces of data. It involves associating a unique key with a corresponding value. This can be used for settings, preferences, or any small piece of information.

Usage in Android (SharedPreferences): In Android, SharedPreferences is used for storing key-value pairs. It allows you to persistently store simple data types like strings, integers, booleans, etc. It's often used for app settings and preferences.

File Storage:

Overview: This method involves saving data in files on the device's internal or external storage. It's suitable for various types of data, including text files, images, audio, and more.

Usage in Android (File I/O): In Android, you can use classes like File, FileOutputStream, and FileInputStream to perform file operations. This allows you to read from and write to files on the device's storage.

Day 2: Introduction to Databases

Database Basics:

Databases are organized collections of structured information, or data, stored electronically in a computer system. They are designed to efficiently manage, store, retrieve, and update data. Here are some basic concepts and components of databases:

Data:

Data refers to the information that is stored in a database. It can be in various forms, such as text, numbers, dates, images, and more.

Database Management System (DBMS):

A DBMS is a software that provides tools and services for creating, managing, and interacting with databases. It acts as an intermediary between the user, applications, and the actual data stored in the database.

Tables:

Tables are the fundamental structure within a database. They organize data into rows and columns, where each row represents a unique record, and each column represents a specific attribute or field.

Fields/Attributes:

Fields or attributes are the individual pieces of information that make up a record. For example, in a database of employees, fields might include "Name," "Age," "Salary," etc.

Records/Tuples:

A record, also known as a tuple, is a single entry in a table. It contains values for each of the fields defined in the table.

Primary Key:

A primary key is a unique identifier for each record in a table. It ensures that each record can be uniquely identified and helps maintain the integrity of the data.

Relations (in Relational Databases):

A relational database organizes data into one or more tables, and the relationships between these tables are defined based on common fields. For example, in a database for a library, there might be a "Books" table and an "Authors" table, with a relationship based on the author's ID.

SQL (Structured Query Language):

SQL is a domain-specific language used for managing and querying databases. It allows users to interact with the database by writing commands to retrieve, insert, update, and delete data.

Indexes:

Indexes are data structures that provide a quick lookup of data based on specific columns. They improve the performance of data retrieval operations but may increase the time it takes to perform write operations.

Normalization:

Normalization is the process of organizing data in a database to minimize redundancy and dependency. This helps improve data integrity and reduce storage space.

Importance of structured data.

Structured data is the foundation of organized information in a digital format. It plays a pivotal role in numerous facets of data management and analysis, offering a framework that enables efficient storage, retrieval, and interpretation of data. Below are comprehensive reasons why structured data is import:

Clear Organization: Structured data is organized in a systematic and easily comprehensible manner. It typically employs tables, where rows represent individual records, and columns represent specific attributes or characteristics of those records. This organization allows for easy navigation and management of data, making it accessible and understandable for users.

Facilitates Retrieval and Querying: One of the primary advantages of structured data is its ease of retrieval. With the aid of querying languages like SQL (Structured Query Language), users can extract specific subsets of data based on defined criteria. This capability is crucial for quickly accessing relevant information from large datasets.

Supports Data Integrity and Consistency: Structured data allows for the application of constraints and validations. These rules ensure that data remains accurate, reliable, and consistent over time. Constraints prevent incorrect or invalid data from being entered, maintaining data integrity.

Enhances Data Analysis and Reporting: Structured data is well-suited for data analysis tasks. Its organized format facilitates the use of various tools, such as spreadsheets, statistical software, and business intelligence platforms. This enables users to derive meaningful insights, generate reports, and perform complex analyses.

Enables Automation and Integration: Structured data is conducive to automation. It can be processed by automated systems, algorithms, and scripts more easily. This is crucial for tasks like data validation, transformation, and integration across different systems and applications.

Simplifies Data Sharing and Interoperability: Structured data is more readily shared and integrated across diverse platforms and applications. Its standardized format allows for smoother data exchange between different software systems, promoting interoperability and seamless collaboration.

Optimizes Storage and Retrieval Efficiency: The structured format of data allows for the use of indexing and other optimization techniques. This leads to faster retrieval times and more efficient utilization of storage resources. Indexing provides a quick lookup mechanism, similar to an index in a book, making data retrieval more efficient.

Supports Data Validation and Constraints: Structured data enables the implementation of rules and constraints on the data. These rules ensure that data adheres to predefined criteria, preventing incorrect or invalid values from being entered. This safeguards the accuracy and reliability of the data.

Enhances Data Security Measures: The structured format of data allows for the implementation of robust security measures. Access controls, encryption, and authentication mechanisms can be more effectively applied to protect sensitive information from unauthorized access or breaches.

Facilitates Data Governance and Compliance: Structured data is more amenable to governance and compliance efforts. It allows for the implementation of policies, procedures, and auditing mechanisms to ensure that data handling adheres to industry regulations and data protection laws.

Forms the Basis for Data Modeling and Predictive Analytics: Structured data often serves as the foundation for creating models that can make predictions or decisions based on historical data patterns. This is employed in fields like machine learning, where algorithms learn from structured data to make accurate predictions.

Improves Collaboration and Communication: Structured data with well-defined schemas fosters effective collaboration among multiple teams and stakeholders. Everyone involved understands the format and meaning of the data, reducing misunderstandings and errors in communication.

Types of Databases:

Overview of different database options (SQLite, Firebase, etc.).

Here's an overview of some popular database options, including SQLite, Firebase, MongoDB, and MySQL

SQLite:

Type: Embedded Database (RDBMS)

Description: SQLite is a lightweight, serverless, self-contained database engine. It's perfect for mobile apps or small-scale applications where a standalone database is needed. It doesn't require a separate server process and is stored directly on the device or application.

Firebase Realtime Database:

Type: NoSQL Database (JSON-based)

Description: Firebase Realtime Database is a cloud-hosted NoSQL database that allows real-time synchronization and updates. It's designed for building web and mobile applications where real-time updates and collaborative features are essential.

Firebase Firestore:

Type: NoSQL Database (Document-Oriented)

Description: Firestore is another offering from Firebase, providing a more scalable and powerful NoSQL database compared to the Realtime Database. It's designed for scalable, high-performance applications with a document-oriented data model.

MongoDB:

Type: NoSQL Database (Document-Oriented)

Description: MongoDB is a popular open-source NoSQL database known for its flexibility, scalability, and ease of use. It uses a document-oriented data model, making it suitable for handling complex and dynamic data.

MySQL:

Type: Relational Database Management System (RDBMS)

Description: MySQL is one of the most widely used open-source relational databases. It's known for its speed, reliability, and scalability. It's used in a wide range of applications, from small websites to large-scale enterprise systems.

PostgreSQL:

Type: Relational Database Management System (RDBMS)

Description: PostgreSQL is another powerful open-source RDBMS known for its robustness, extensibility, and support for advanced features like JSON data types and full-text search. It's often used in applications where complex data types and advanced querying capabilities are required.

Microsoft SQL Server:

Type: Relational Database Management System (RDBMS)

Description: Microsoft SQL Server is a comprehensive RDBMS developed by Microsoft. It offers a wide range of features including support for business intelligence, data warehousing, and integration with other Microsoft technologies.

Oracle Database:

Type: Relational Database Management System (RDBMS)

Description: Oracle Database is a powerful and widely used commercial RDBMS known for its high performance, scalability, and security features. It's commonly used in large enterprise applications.

Amazon DynamoDB:

Type: NoSQL Database (Key-Value and Document-Oriented)

Description: DynamoDB is a managed NoSQL database service provided by Amazon Web Services (AWS). It's designed for high availability, scalability, and low-latency access to data. It supports both key-value and document-oriented data models.

Pros and cons of each.

SQLite:

Pros:

- Lightweight and serverless, making it easy to set up and use.
- Requires no separate server process.
- Suitable for small-scale applications and mobile devices.
- Single-file database, making it easy to transfer and backup.

Cons:

- Not suitable for large-scale or high-concurrency applications.
- Limited support for complex queries and transactions.
- Lacks built-in user authentication and access control mechanisms.

Firebase Realtime Database:

Pros:

- Provides real-time synchronization, making it ideal for collaborative applications.
- Offers seamless integration with other Firebase services like authentication and hosting.
- Suitable for applications requiring real-time updates, such as chat apps and gaming.

Cons:

- Limited querying capabilities compared to more traditional databases.
- Limited support for complex data querying and relationships.
- May lead to higher costs as usage scales up.

Firebase Firestore:

Pros:

- Offers more advanced querying capabilities compared to the Realtime Database.
- Provides scalable, serverless architecture with real-time synchronization.
- Supports more complex data structures and relationships.

Cons:

- May lead to higher costs as usage scales up.
- Limited support for transactions compared to some other databases.
- Not as suitable for applications with extremely high concurrency requirements.

MongoDB:

Pros:

- Offers high flexibility, allowing for dynamic schema design.

- Scalable and suitable for handling large volumes of data.
- Supports rich query language and indexing for complex data retrieval.

Cons:

- May require more careful schema design to prevent redundancy and ensure data integrity.
- Can be memory-intensive for certain operations.
- Limited support for transactional operations compared to some RDBMS.

MySQL:

Pros:

- Well-established and widely used, with a large community and extensive documentation.
- Supports complex transactions, making it suitable for applications with high data integrity requirements.
- Provides strong data security features and supports encryption.

Cons:

- May require more effort in setting up and managing compared to some NoSQL databases.
- Limited flexibility in handling unstructured or semi-structured data.
- Performance may degrade in extremely high-concurrency scenarios.

PostgreSQL:

Pros:

- Offers advanced features like support for JSON data types, full-text search, and complex data types.
- Highly extensible, allowing for the creation of custom functions and data types.
- Strong support for data integrity through constraints and transactions.

Cons:

- May have a steeper learning curve for beginners compared to some other databases.
- Requires more memory and resources compared to some lightweight databases.
- May not perform as well as some other databases in certain scenarios.

Microsoft SQL Server:

Pros:

- Offers a comprehensive suite of tools and features, including support for business intelligence and reporting services.
- Strong integration with other Microsoft technologies and products.
- High level of security and support for complex transactions.

Cons:

- Licensing costs may be prohibitive for smaller organizations or projects.
- Primarily designed for Windows environments, which may limit cross-platform compatibility.
- Requires more resources compared to some lighter-weight databases.

Oracle Database:

Pros:

- Extremely robust and scalable, suitable for large enterprise applications with high concurrency requirements.
- Offers a wide range of advanced features, including support for parallel processing and advanced analytics.
- Strong support for data security and compliance.

Cons:

- High licensing and maintenance costs, making it less accessible for smaller organizations.
- Resource-intensive, requiring powerful hardware and infrastructure.
- May have a steeper learning curve compared to some other databases.

Amazon DynamoDB:

Pros:

- Managed service with automatic scaling and high availability.
- Designed for high performance, low-latency access to data, making it suitable for high-throughput applications.
- Supports both key-value and document data models.

Cons:

- Limited flexibility in query capabilities compared to some other databases.
- Costs can increase significantly as data scales up.
- May not be as suitable for applications with complex relationships or complex querying requirements.

Day 3: Working with SQLite (or chosen Database)

Setting Up SQLite:

Working with SQLite involves setting up a database, creating tables, and performing operations like inserting, updating, and querying data. Below are steps to set up SQLite:

Install SQLite:

Linux:

SQLite is often pre-installed on Linux systems. To check if it's installed, open a terminal and type sqlite3. If you get a prompt, SQLite is installed.

If not installed, you can install it using your package manager. For example, on Ubuntu, you can use: sudo apt-get install sqlite3.

Windows:

Download the SQLite command-line shell from the official website (https://sqlite.org/download.html) and follow the installation instructions.

macOS:

SQLite is pre-installed on macOS. You can access it via the terminal by typing sqlite3.

Integrating SQLite into your mobile app project.

Step by Step Implementation

Step 1: Create a New Project

Step 2: Adding permissions to access the storage in the AndroidManifest.xml file

Navigate to the app > AndroidManifest.xml and add the below code to it.

XML:

```
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

Step 3: Working with the activity_main.xml file

Navigate to the app > res > layout > activity_main.xml and add the below code to that file. Below is the code for the activity_main.xml file.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <!--Edit text to enter course name-->
    <EditText
        android:id="@+id/idEdtCourseName"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="10dp"
        android:hint="Enter course Name" />
```

```xml
<!--edit text to enter course duration-->
<EditText
    android:id="@+id/idEdtCourseDuration"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="10dp"
    android:hint="Enter Course Duration" />

<!--edit text to display course tracks-->
<EditText
    android:id="@+id/idEdtCourseTracks"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="10dp"
    android:hint="Enter Course Tracks" />

<!--edit text for course description-->
<EditText
    android:id="@+id/idEdtCourseDescription"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="10dp"
    android:hint="Enter Course Description" />

<!--button for adding new course-->
<Button
    android:id="@+id/idBtnAddCourse"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
```

```
        android:layout_margin="10dp"

        android:text="Add Course"

        android:textAllCaps="false" />


</LinearLayout>
```

Step 4: Creating a new Java class for performing SQLite operations.

Navigate to the app > java > your app's package name > Right-click on it > New > Java class and name it as DBHandler and add the below code to it. Comments are added inside the code to understand the code in more detail.

```
import android.content.ContentValues
import android.content.Context
import android.database.sqlite.SQLiteDatabase
import android.database.sqlite.SQLiteOpenHelper

class DBHandler(context: Context) : SQLiteOpenHelper(context, DB_NAME, null, D
B_VERSION) {

    companion object {
        private const val DB_NAME = "coursedb"
        private const val DB_VERSION = 1
        private const val TABLE_NAME = "mycourses"
        private const val ID_COL = "id"
        private const val NAME_COL = "name"
        private const val DURATION_COL = "duration"
        private const val DESCRIPTION_COL = "description"
        private const val TRACKS_COL = "tracks"
    }

    override fun onCreate(db: SQLiteDatabase) {
        val query = "CREATE TABLE $TABLE_NAME (" +
                "$ID_COL INTEGER PRIMARY KEY AUTOINCREMENT, " +
                "$NAME_COL TEXT," +
                "$DURATION_COL TEXT," +
                "$DESCRIPTION_COL TEXT," +
                "$TRACKS_COL TEXT)"

        db.execSQL(query)
    }

    fun addNewCourse(courseName: String, courseDuration: String, courseDescrip
tion: String, courseTracks: String) {
```

```
        val db = this.writableDatabase
        val values = ContentValues().apply {
            put(NAME_COL, courseName)
            put(DURATION_COL, courseDuration)
            put(DESCRIPTION_COL, courseDescription)
            put(TRACKS_COL, courseTracks)
        }
        db.insert(TABLE_NAME, null, values)
        db.close()
    }

    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: In
t) {
        db.execSQL("DROP TABLE IF EXISTS $TABLE_NAME")
        onCreate(db)
    }
}
```

Step 5: Working with the MainActivity.kt

Go to the MainActivity. kt file and refer to the following code. Below is the code for the MainActivity. kt file. Comments are added inside the code to understand the code in more detail.

```
import android.os.Bundle
import android.view.View
import android.widget.Button
import android.widget.EditText
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {

    private lateinit var courseNameEdt: EditText
    private lateinit var courseTracksEdt: EditText
    private lateinit var courseDurationEdt: EditText
    private lateinit var courseDescriptionEdt: EditText
    private lateinit var addCourseBtn: Button
    private lateinit var dbHandler: DBHandler

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        courseNameEdt = findViewById(R.id.idEdtCourseName)
        courseTracksEdt = findViewById(R.id.idEdtCourseTracks)
        courseDurationEdt = findViewById(R.id.idEdtCourseDuration)
        courseDescriptionEdt = findViewById(R.id.idEdtCourseDescription)
```

```
        addCourseBtn = findViewById(R.id.idBtnAddCourse)

        dbHandler = DBHandler(this)

        addCourseBtn.setOnClickListener {
            val courseName = courseNameEdt.text.toString()
            val courseTracks = courseTracksEdt.text.toString()
            val courseDuration = courseDurationEdt.text.toString()
            val courseDescription = courseDescriptionEdt.text.toString()

            if (courseName.isEmpty() && courseTracks.isEmpty() && courseDurati
on.isEmpty() && courseDescription.isEmpty()) {
                Toast.makeText(this, "Please enter all the data..", Toast.LENG
TH_SHORT).show()
                return@setOnClickListener
            }

            dbHandler.addNewCourse(courseName, courseDuration, courseDescripti
on, courseTracks)

            Toast.makeText(this, "Course has been added.", Toast.LENGTH_SHORT)
.show()

            courseNameEdt.setText("")
            courseDurationEdt.setText("")
            courseTracksEdt.setText("")
            courseDescriptionEdt.setText("")
        }
    }
}
```

Day 4: Working with Firebase (or chosen Database)

Setting Up Firebase:

Step 1: Create a Firebase Project

Follow the steps mentioned earlier to create a Firebase project in the Firebase Console.

Step 2: Add Your App to the Project

Register your app in the Firebase Console.

Download the google-services.json file and place it in your app module's app directory.

Add the Firebase SDK to your build.gradle file:

implementation 'com.google.firebase:firebase-database:22.0.0' // For Firebase Realtime Database

// or

implementation 'com.google.firebase:firebase-firestore:24.0.0' // For Firebase Firestore

Certainly! Here's the equivalent code in Kotlin:

Step 1: Create a Firebase Project

Follow the steps mentioned earlier to create a Firebase project in the Firebase Console.

Step 2: Add Your App to the Project

Register your app in the Firebase Console.

Download the google-services.json file and place it in your app module's app directory.

Add the Firebase SDK to your build.gradle file:

gradle

Copy code

implementation 'com.google.firebase:firebase-database:22.0.0' // For Firebase Realtime Database

// or

implementation 'com.google.firebase:firebase-firestore:24.0.0' // For Firebase Firestore

Step 3: Initialize Firebase in Your App

In your app's main activity or application class, add the following initialization code:

```
// For Realtime Database
FirebaseDatabase.getInstance().setPersistenceEnabled(true)

// Initialize Firebase
FirebaseApp.initializeApp(this)
```

Using Firebase for Data Storage:

Firebase Realtime Database:

Writing Data:

```
val databaseRef = FirebaseDatabase.getInstance().getReference("users")

val user = User("Ahmed ", "Ahmed@example.com")
databaseRef.child("userId123").setValue(user)
```

Reading Data:

```
databaseRef.addValueEventListener(object : ValueEventListener {
    override fun onDataChange(dataSnapshot: DataSnapshot) {
        for (userSnapshot in dataSnapshot.children) {
            val user = userSnapshot.getValue(User::class.java)
            // Process user data
        }
    }

    override fun onCancelled(databaseError: DatabaseError) {
```

```
            // Handle error
        }
})
```

Day 5 Uploading, retrieving, and managing data with Firebase Firestore

Writing Data:

```
val db = FirebaseFirestore.getInstance()

val user = hashMapOf(
    "name" to "Ali",
    "email" to "Ali@example.com"
)


db.collection("users")
    .document("userId123")
    .set(user)
```

Reading Data:

```
val db = FirebaseFirestore.getInstance()

db.collection("users")
    .get()
    .addOnSuccessListener { documents ->
        for (document in documents) {
            val user = document.toObject(User::class.java)
            // Process user data
        }
    }
    .addOnFailureListener { exception ->
        // Handle error
    }
```

Week 7- Handling Features and Sensors

Day 1: Introduction to Device Features

Overview of Device Features:

Device features refer to the functionalities and capabilities integrated into a device. These features can vary widely depending on the type of device, its intended use, and its target audience. Some common examples include:

Sensors: These are components that detect changes in the environment and convert them into electrical signals. Sensors can measure things like temperature, light, pressure, motion, and more.

Connectivity: Devices can have various connectivity options like Wi-Fi, Bluetooth, NFC, or cellular capabilities. These allow devices to communicate with each other or with external networks.

Input/Output Methods: This includes touchscreens, keyboards, buttons, cameras, and microphones. These features enable users to interact with the device.

Processing Power: The CPU (Central Processing Unit) and GPU (Graphics Processing Unit) determine a device's computing capabilities. These are vital for running applications and processing data.

Storage: This includes both RAM (Random Access Memory) for temporary data storage and permanent storage like SSDs (Solid State Drives) or HDDs (Hard Disk Drives) for long-term data retention.

Operating System and Software: These define the user interface and the range of applications a device can run. Different operating systems (e.g., iOS, Android, Windows, Linux) have unique features and capabilities.

Power Source: Devices may be powered by batteries, external power sources, or a combination of both. Understanding power management is crucial for optimizing battery life.

Form Factor and Design: The physical design and layout of a device greatly influence its usability and user experience.

Importance of Device Features

Understanding and harnessing device features is crucial for several reasons:

Optimized User Experience: Well-designed features enhance usability and provide a seamless experience for users.

Functional Efficiency: Leveraging features effectively ensures that the device performs its intended tasks efficiently and accurately.

Competitive Advantage: Devices with innovative or advanced features often gain a competitive edge in the market.

Compliance and Standards: Certain industries have specific standards and regulations that devices must meet. Understanding features helps ensure compliance.

Security and Privacy: Many features, such as biometric sensors or encryption capabilities, are critical for ensuring the security and privacy of user data.

Customization and Personalization: Knowing how to work with device features allows for tailoring experiences to individual user preferences.

Understanding the capabilities of mobile devices (camera, GPS, accelerometer, etc).

Camera:

Capabilities: Mobile cameras use image sensors to capture visual data. They can have various specifications like megapixels, lens type (wide-angle, telephoto, etc.), and features like optical image stabilization (OIS), autofocus, and HDR.

Applications: Besides basic photography and videography, modern cameras support advanced features like depth sensing for portrait mode, night mode for low-light photography, and AI-driven scene recognition for optimized settings.

GPS (Global Positioning System):

Capabilities: GPS uses signals from multiple satellites to triangulate the device's precise location on Earth. This includes latitude, longitude, altitude, and speed data.

Applications: GPS is used extensively in navigation apps like Google Maps, location-based services (e.g., finding nearby businesses), geotagging in photos, and tracking fitness activities.

Accelerometer:

Capabilities: This sensor measures acceleration forces, including gravity. It provides information about the device's movement in three dimensions.

Applications: Apart from screen rotation and pedometer functions, accelerometers are vital in gaming for tilt-based controls and in detecting gestures for various tasks.

Gyroscope:

Capabilities: A gyroscope complements the accelerometer by providing information about the device's orientation in three-dimensional space, specifically angular velocity.

Applications: It's crucial in applications where precise rotational movements are needed, like in VR and AR experiences.

Magnetometer (Compass):

Capabilities: The magnetometer detects the Earth's magnetic field, which allows the device to determine its orientation relative to magnetic north.

Applications: Beyond basic compass applications, it's essential for navigation, especially when used in conjunction with GPS.

Proximity Sensor:

Capabilities: Proximity sensors detect when an object is close to the device without direct contact. They typically use infrared or ultrasonic technology.

Applications: This sensor is used during calls to turn off the screen and prevent unintended touch inputs when the device is close to the user's face.

Ambient Light Sensor:

Capabilities: It measures the intensity of light in the device's environment, allowing for automatic adjustment of display brightness for optimal visibility and power efficiency.

Applications: This sensor is responsible for adaptive brightness, which ensures that the screen is neither too bright nor too dim in different lighting conditions.

Fingerprint Sensor/Biometric Sensors:

Capabilities: These sensors capture and authenticate unique biological features like fingerprints, facial features, or iris patterns.

Applications: Biometric sensors are primarily used for unlocking devices, securing apps, and authorizing secure transactions.

How these features can enhance app functionality.

Camera:

Augmented Reality (AR): AR apps can overlay digital information onto the real world captured by the camera, creating interactive and immersive experiences.

Visual Search: Utilizing the camera for object recognition allows users to search for information about real-world objects.

Document Scanning: Apps can use the camera to scan and digitize documents, receipts, or business cards.

GPS (Global Positioning System):

Location-Based Services (LBS): Apps can provide personalized content or services based on the user's location, such as finding nearby restaurants, stores, or events.

Geofencing: This allows apps to trigger actions or notifications when a user enters or exits a predefined geographical area.

Accelerometer and Gyroscope:

Gaming: Tilt controls and motion gestures can enhance gaming experiences, providing a more interactive and immersive gameplay.

Fitness and Health: These sensors can track physical activities, count steps, and measure movements for accurate fitness tracking.

Magnetometer (Compass):

Navigation: Apps can provide accurate compass-based navigation, particularly useful in outdoor and adventure applications.

Proximity Sensor:

Call Management: During calls, the proximity sensor ensures that the screen turns off to prevent unintended touch inputs while the phone is close to the user's face.

Ambient Light Sensor:

Adaptive UI: Apps can adjust their interface brightness and color schemes based on ambient lighting conditions, enhancing visibility and reducing eye strain.

Fingerprint Sensor/Biometric Sensors:

Security: Biometric sensors provide secure authentication methods for unlocking devices, accessing sensitive apps, or making secure transactions.

Personalization: Apps can use biometrics for user-specific customization and preferences.

Day 2: Working with the Camera

Camera Integration:

Setting up camera access in your app.

First, we need to set Permissions enabled in order to use camera of phone, to set permission enabled:

// AndroidManifest.xml

<uses-feature android:name="android.hardware.camera" />

<uses-permission android:name="android.permission.CAMERA" />

// Request camera permission at runtime (for Android 6.0+)

if (ContextCompat.checkSelfPermission(this, Manifest.permission.CAMERA) != PackageManager.PERMISSION_GRANTED) {

ActivityCompat.requestPermissions(this, arrayOf(Manifest.permission.CAMERA), CAMERA_PERMISSION_REQUEST)

}

Capturing photos and videos.

```
// Taking Photos
private fun takePhoto() {
    val intent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
    startActivityForResult(intent, REQUEST_IMAGE_CAPTURE)
}

// Handling the result
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?
) {
    super.onActivityResult(requestCode, resultCode, data)
    if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode == Activity.RESULT_
OK) {
        val imageBitmap = data?.extras?.get("data") as Bitmap
        // Process the captured photo (if needed)
    }
}

// Recording Videos (requires additional setup)
private fun recordVideo() {
    val intent = Intent(MediaStore.ACTION_VIDEO_CAPTURE)
    startActivityForResult(intent, REQUEST_VIDEO_CAPTURE)
}
```

Processing Media:

Editing, filtering, or applying effects to captured media.

```
// Applying a filter using BitmapFactory (for simplicity, consider using a lib
rary for real applications)
private fun applyFilter(originalBitmap: Bitmap): Bitmap {
    val filteredBitmap = Bitmap.createBitmap(originalBitmap.width, originalBit
map.height, originalBitmap.config)
    val canvas = Canvas(filteredBitmap)
```

```
    val paint = Paint()
    val colorMatrix = ColorMatrix()
    colorMatrix.setSaturation(0f) // Grayscale
    val filter = ColorMatrixColorFilter(colorMatrix)
    paint.colorFilter = filter

    canvas.drawBitmap(originalBitmap, 0f, 0f, paint)
    return filteredBitmap
}

// Saving the processed media
private fun saveProcessedMedia(processedBitmap: Bitmap) {
    val filePath = "${Environment.getExternalStorageDirectory()}/processed_ima
ge.jpg"
    try {
        val out = FileOutputStream(filePath)
        processedBitmap.compress(Bitmap.CompressFormat.JPEG, 100, out)
        out.close()
    } catch (e: IOException) {
        e.printStackTrace()
    }
}
```

Day 3: Implementing GPS and Location Services

Location Services Integration:

To implement GPS and Location Services in an Android application, you'll need to integrate the necessary permissions, set up a LocationManager, and handle location updates. Below is a step-by-step guide to help you achieve this:

Location Services Integration:

Requesting Permissions:

Add the necessary permissions to your AndroidManifest.xml file:

<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />

Request runtime permissions in activity:

```
// Inside working  Activity
if (ContextCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LO
CATION) != PackageManager.PERMISSION_GRANTED) {
    ActivityCompat.requestPermissions(this, arrayOf(Manifest.permission.ACCESS
_FINE_LOCATION), LOCATION_PERMISSION_REQUEST)
}
```

Accessing GPS data in your app.

Initializing Location Services:

Create a LocationManager instance, set up a LocationListener, and request location updates:

```
// Inside your Activity
val locationManager = getSystemService(Context.LOCATION_SERVICE) as LocationMa
nager

val locationListener = object : LocationListener {
    override fun onLocationChanged(location: Location) {
        // Handle location updates here
        val latitude = location.latitude
        val longitude = location.longitude
        // Use latitude and longitude for your application's purpose
    }

    override fun onStatusChanged(provider: String, status: Int, extras: Bundle
) {}

    override fun onProviderEnabled(provider: String) {}

    override fun onProviderDisabled(provider: String) {}
}

locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, MIN_TIME_
BETWEEN_UPDATES, MIN_DISTANCE_CHANGE_FOR_UPDATES, locationListener)
```

Handling Location Updates:

Implement the logic to handle location updates inside the onLocationChanged method of your LocationListener:

```
override fun onLocationChanged(location: Location) {
    val latitude = location.latitude
    val longitude = location.longitude


}
```

Retrieving current location and tracking movement

Retrieving Current Location:

You can also request the last known location to get an immediate fix:

```
val lastKnownLocation = locationManager.getLastKnownLocation(LocationManager.G
PS_PROVIDER)
if (lastKnownLocation != null) {
    val latitude = lastKnownLocation.latitude
    val longitude = lastKnownLocation.longitude
    // Use latitude and longitude for your application's purpose
}
```

Day 4: Utilizing the Accelerometer and Sensors

Understanding Sensors:

The accelerometer sensor measures acceleration along the three physical axes (x, y, and z). It provides information about changes in motion and orientation of the device.

How to access and utilize sensor data.

How to Access and Utilize Accelerometer Sensor Data:

Initialize Sensor Manager:

Get an instance of the SensorManager to access the device's sensors.

```
val sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
```

Get the Accelerometer Sensor:

Retrieve the accelerometer sensor.

```
val accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)
```

Register Sensor Listener:

Register a SensorEventListener to listen for changes in the accelerometer data.

```
sensorManager.registerListener(sensorEventListener, accelerometer, SensorManag
er.SENSOR_DELAY_NORMAL)
```

Implement Sensor Event Handling:

Override the onSensorChanged method to receive and process accelerometer data.

```
val sensorEventListener = object : SensorEventListener {
    override fun onSensorChanged(event: SensorEvent) {
        if (event.sensor.type == Sensor.TYPE_ACCELEROMETER) {
            val x = event.values[0]
            val y = event.values[1]
            val z = event.values[2]

            // Process accelerometer data (e.g., detect motion, orientation ch
anges)
        }
    }

    override fun onAccuracyChanged(sensor: Sensor?, accuracy: Int) {
        // Handle accuracy changes (if needed)
    }
}
```

Unregister Sensor Listener:

Remember to unregister the listener when it's no longer needed to conserve resources.

```
override fun onPause() {
    super.onPause()
    sensorManager.unregisterListener(sensorEventListener)
}
```

Implementing Motion-Based Interactions:

Example: Tilt-Based Interaction

```
class TiltActivity : AppCompatActivity() {

    private lateinit var sensorManager: SensorManager
    private lateinit var accelerometer: Sensor

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_tilt)

        sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorMana
ger
        accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMET
ER)
    }

    override fun onResume() {
        super.onResume()
        sensorManager.registerListener(sensorEventListener, accelerometer, Sen
sorManager.SENSOR_DELAY_NORMAL)
    }

    override fun onPause() {
        super.onPause()
        sensorManager.unregisterListener(sensorEventListener)
    }

    private val sensorEventListener = object : SensorEventListener {
        override fun onSensorChanged(event: SensorEvent) {
            if (event.sensor.type == Sensor.TYPE_ACCELEROMETER) {
                val x = event.values[0]
                val y = event.values[1]
                val z = event.values[2]

                // Implement tilt-
based interaction based on accelerometer data
                // e.g., adjust UI elements, control a game, etc.
            }
        }
```

```
        override fun onAccuracyChanged(sensor: Sensor?, accuracy: Int) {
            // Handle accuracy changes (if needed)
        }
    }
}
```

In this example, the accelerometer data is used to implement a tilt-based interaction. As the device is tilted, you can adjust UI elements or perform other actions based on the accelerometer data.

Remember to replace the placeholder action in the onSensorChanged method with the specific functionality you want to implement.

Additional Considerations:

Sensor Delay:

Adjust the sensor delay (e.g., SENSOR_DELAY_NORMAL) based on your app's requirements. Higher delays conserve battery but provide less frequent updates.

Battery Considerations:

Be mindful of power consumption when continuously using sensors. Unregister listeners when they're no longer needed.

Sensor Availability:

Check if the accelerometer sensor is available on the device before attempting to use it.

Day 5: Device Permissions and Security

Requesting Permissions:

Requesting permissions is a crucial aspect of Android app development, ensuring that your app has the necessary access to device resources while respecting user privacy and security. Here's a step-by-step guide on how to request permissions in your Android app:

Requesting Permissions:

Declare Permissions in the Manifest:

Open your AndroidManifest.xml file.

Add the necessary permissions inside the <manifest> element. For example, to request access to the camera, you would add:

```
<uses-permission android:name="android.permission.CAMERA" />
```

Why You Need the Permission:

It's important to provide a clear explanation to the user about why your app needs a particular permission. This can be done using a dialog or a message.

Handle Denied Permission:

If the user denies permission, ensure your app handles it gracefully. This might involve providing an alternative method or explaining the benefits of granting permission.

Request Multiple Permissions (if needed):

If your app requires multiple permissions, you can request them in a single request.

Week 7- Handling Features and Sensors

Day 1: Introduction to Device Features

Overview of Device Features:

Understanding the capabilities of mobile devices (camera, GPS, accelerometer, etc.).

How these features can enhance app functionality.

Choosing Relevant Features:

Identifying which device features are applicable to your app's purpose.

Day 2: Working with the Camera

Camera Integration:

Setting up camera access in your app.

Capturing photos and videos.

Processing Media:

Editing, filtering, or applying effects to captured media.

Day 3: Implementing GPS and Location Services

Location Services Integration:

Accessing GPS data in your app.

Retrieving current location and tracking movement.

Day 4: Utilizing the Accelerometer and Sensors

Understanding Sensors:

Introduction to different sensors (accelerometer, gyroscope, etc.).

How to access and utilize sensor data.

Implementing Motion-Based Interactions:

Using sensor data for interactive elements in your app.

Day 5: Device Permissions and Security

Requesting Permissions:

How to ask for user consent to access device features.

Best practices for handling permissions.

Security Considerations:

Ensuring secure handling of sensitive data and device features.

Week 8- Mobile Application Development Tools and Frameworks

Day 1: Introduction to Mobile App Development Frameworks

Understanding Mobile App Development Frameworks

Mobile application development involves creating software applications that are designed to run on mobile devices like smartphones and tablets. There are several tools and frameworks available to developers that streamline the process and help create high-quality mobile apps.

Overview of popular cross-platform frameworks (e.g., React Native).

Cross-platform frameworks allow developers to build mobile applications that can run on multiple platforms (such as iOS and Android) using a single codebase. Here's an overview of two popular cross-platform frameworks: React Native and Flutter.

React Native:

Language: JavaScript (React)

Developed By: Facebook

Key Features:

Native-Like Performance: React Native allows for near-native performance by using native components. It also provides a bridge to interact with native modules.

Reusable Components: Developers can reuse a significant portion of their code across iOS and Android, which leads to faster development.

Large Community and Ecosystem: There's a thriving community of developers and a rich ecosystem of libraries and plugins available.

Hot Reloading: Allows developers to see the result of code changes in real-time without restarting the application.

React Integration: If you're familiar with React, the learning curve for React Native is smoother.

Drawbacks:

Limited Access to Native Modules: For complex functionalities, you might need to write native modules in Objective-C or Java.

Less Control Over Native Components: In some cases, you might need to dive into native code for fine-grained control.

Examples of Apps Built with React Native:

Facebook

Instagram

Airbnb

UberEats

Flutter:

Language: Dart

Developed By: Google

Key Features:

Highly Customizable UI: Flutter allows developers to create highly customized and interactive UIs through its rich set of widgets.

Fast Development and Compilation: Flutter's "hot reload" feature allows developers to instantly see the effects of code changes.

Consistent Performance: Since Flutter compiles to native ARM code, it provides high performance and near-native speed.

Strong Community and Growing Ecosystem: While not as large as React Native's community, Flutter's community is vibrant and growing.

Material Design and Cupertino Widgets: Flutter provides both Material Design and Cupertino widgets for a consistent look on both Android and iOS.

Drawbacks:

Learning Curve: Dart may be less familiar to developers compared to JavaScript or other languages.

Limited Native Modules: Like React Native, you may need to write native code for complex functionalities.

Examples of Apps Built with Flutter:

Google Ads

Alibaba

Reflectly

Hamilton Musical App

Choosing Between React Native and Flutter:

React Native might be a better choice if you have a team that's already familiar with JavaScript and React, and if you value the large community and extensive library ecosystem.

Flutter might be a better choice if you want a high level of customization for your UI, and if you're interested in the fast development and consistent performance it offers.

Pros and cons of using cross-platform frameworks.

Using cross-platform frameworks for mobile app development comes with its own set of advantages and disadvantages. Here's a breakdown of the pros and cons:

Pros of Using Cross-Platform Frameworks:

Code Reusability:

Pro: Developers can write a single codebase that works on multiple platforms (iOS, Android, etc.), which saves time and effort.

Faster Development:

Pro: Development is generally faster compared to building separate native apps for each platform. Features like hot-reloading can speed up the development process.

Cost-Effectiveness:

Pro: Since one codebase serves multiple platforms, it's often more cost-effective than building separate native apps.

Consistent UI/UX:

Pro: Cross-platform frameworks often provide tools to ensure a consistent look and feel across different devices and platforms.

Larger Developer Pool:

Pro: Developers with expertise in cross-platform frameworks can work on both iOS and Android projects, providing a larger pool of potential talent.

Access to Plugins and Libraries:

Pro: Many cross-platform frameworks have a rich ecosystem of plugins and libraries that can be used to add functionality to the app.

Easier Maintenance:

Pro: With a single codebase, maintenance and updates can be more streamlined and efficient.

Cons of Using Cross-Platform Frameworks:

Performance Overheads:

Con: Cross-platform apps may not always perform as well as native apps, especially for graphics-intensive or CPU-intensive tasks.

Limited Access to Native Features:

Con: Some advanced or platform-specific features may not be directly accessible through the cross-platform framework and may require native code.

Learning Curve:

Con: Developers may need to learn the specific language or framework used by the cross-platform tool, which can take time.

Less Control Over Platform-Specific Issues:

Con: Platform updates or changes may not be immediately reflected in the cross-platform framework, potentially leading to compatibility issues.

Dependence on Framework's Development Cycle:

Con: The progress and updates of the cross-platform framework can impact your project. If the framework is not well-maintained, it may lead to issues.

Potential for Larger App Size:

Con: Cross-platform apps may have a larger file size due to the inclusion of the framework itself.

Platform Limitations:

Con: Some platform-specific features may not be easily replicable in a cross-platform environment, requiring workarounds or native code.

Choosing Between Cross-Platform and Native:

Cross-Platform is a good choice for:

- Apps with relatively straightforward UI/UX requirements.
- Projects with time and budget constraints.
- Apps where code sharing and rapid development are crucial.

Native may be preferred for:

- High-performance apps (e.g., games, complex simulations).
- Apps that rely heavily on platform-specific features.
- Projects with a focus on platform-specific user experiences.

Choosing the Right Framework:

Choosing the right framework for your mobile app development is a critical decision that can significantly impact the success and efficiency of your project. Here's a step-by-step guide to help you make an informed choice:

Define Project Requirements:

Clearly outline the features, functionalities, and performance expectations of your app. Consider factors like UI complexity, real-time updates, geolocation, and hardware integration.

Target Platforms:

Determine which platforms you want to deploy on (iOS, Android, or both). Some frameworks specialize in specific platforms, while others are cross-platform.

Consider Developer Skillset:

Assess the skills and expertise of your development team. If they are already proficient in a specific language or framework, it may be more efficient to leverage their existing knowledge.

Performance Requirements:

If your app requires high performance (e.g., gaming or complex simulations), native development might be more suitable. Cross-platform frameworks like Flutter and React Native can also achieve good performance, but may not match native speeds in all cases.

UI/UX Complexity:

Evaluate the complexity of your app's user interface. If you need highly customized and platform-specific designs, frameworks like Flutter or native development might be preferred.

Community and Ecosystem:

Consider the size and activity level of the framework's community. A robust community means more resources, libraries, and support available.

Access to Native Features:

Determine if your app requires extensive access to platform-specific features (e.g., camera, GPS, sensors). Some cross-platform frameworks may have limitations in this regard.

Development Speed:

Assess how quickly you need to bring your app to market. Cross-platform frameworks like React Native and Flutter are known for their rapid development capabilities.

Maintenance and Long-Term Support:

Evaluate the framework's maintenance history and whether it has long-term support. This is crucial for ensuring the longevity and stability of your app.

Cost Considerations:

Factor in the costs associated with development, maintenance, and potential licensing fees for certain frameworks. Cross-platform development can be more cost-effective due to code reusability.

Previous Project Success:

Research case studies or examples of successful apps built with the framework you're considering. This can provide valuable insights into its capabilities and limitations.

Prototyping and Testing:

Consider how easy it is to create prototypes and conduct testing with the chosen framework. A good framework should facilitate efficient testing and debugging.

Scalability:

If your app is expected to grow in user base or features, consider how well the chosen framework supports scalability.

Day 2: Exploring React Native

React Native is an open-source framework developed by Facebook for building mobile applications using JavaScript and React. It allows developers to use familiar web technologies like JavaScript and React to build native mobile apps for iOS and Android platforms.

Here are some key points about React Native:

Cross-Platform Development: With React Native, you can write code once and deploy it on both iOS and Android platforms. This saves a significant amount of development time compared to writing separate codebases for each platform.

Native Components: React Native allows you to use native components written in Objective-C, Swift (for iOS), and Java, Kotlin (for Android). These components are rendered using native APIs, providing a performance similar to apps built with platform-specific languages.

Hot Reloading: This feature enables developers to see the results of their code changes in real-time without having to recompile the entire app. It speeds up the development process and makes it easier to experiment with UI changes.

Large Community and Ecosystem: React Native has a large and active community of developers. This means there are a lot of libraries, plugins, and resources available to help with common tasks and functionalities.

Support for Third-Party Plugins: You can use native modules or create your own in native languages to interact with device functionalities not covered by React Native out of the box.

Performance Optimization: While React Native provides good performance out of the box, there are ways to optimize performance further by writing native code for specific parts of the app.

Support for Popular IDEs: You can use a variety of popular Integrated Development Environments (IDEs) such as Visual Studio Code, Atom, and others to write React Native applications.

Community and Documentation: React Native has a vibrant community, and there are extensive resources available including documentation, tutorials, and forums.

Setting Up the Development Environment:

Installing React Native and required dependencies.

Node.js and npm

React Native requires Node.js and npm (Node Package Manager) to be installed on your machine. Visit the official Node.js website and download the latest LTS version.

Node's official website.

Java Development Kit (JDK)

Android Studio (for Android Development)

Creating a New React Native Project

Now that your environment is set up, you can create a new React Native project using either Expo or the React Native CLI:

Setting up a new project.

Building a Simple App:

Creating a basic app using React Native.

```
npx react-native init MyProject
cd MyProject
npx react-native run-android (for Android)
npx react-native run-ios (for iOS)
```

Testing on Devices or Emulators

Understanding component-based architecture.

Component-based architecture is a design approach used in software development where a program is organized into reusable, self-contained units known as components. Each component encapsulates a specific piece of functionality or user interface element and can be composed together to build more complex applications.

Here are the key principles and concepts of component-based architecture:

1. Reusability

Components are designed to be reused in different parts of an application or even in different applications altogether. This promotes code modularity and reduces redundancy, as you can leverage existing components rather than writing the same code from scratch.

2. Encapsulation

Components encapsulate their functionality, meaning that the internal workings of a component are hidden from the outside world. This allows for better abstraction and separation of concerns, making it easier to maintain and update the application.

3. Isolation

Components operate independently, which means they do not directly depend on the internal details of other components. This isolation allows for easier testing and debugging, as components can be examined and modified without affecting the rest of the application.

4. Composition

In a component-based architecture, complex applications are built by assembling or "composing" smaller, reusable components. This composition can occur at multiple levels, with higher-level components composed of lower-level components.

5. Intercommunication

Components communicate with each other through well-defined interfaces. They can pass data and trigger events, allowing them to work together to accomplish a task or present information to the user.

6. Lifecycle Events

Components often have a lifecycle with various phases, such as initialization, rendering, updating, and cleanup. These phases allow components to perform specific actions at different points in their lifespan.

7. State Management

Components can have their own internal state, which represents the data that is relevant to that specific component. This state can change over time, and when it does, the component can re-render to reflect those changes.

8. User Interface (UI) Elements

Components are often associated with user interface elements, such as buttons, forms, lists, and more. These UI elements can be customized and styled to match the application's design.

9. Separation of Concerns

By breaking down the application into smaller components, each responsible for a specific task or piece of functionality, you achieve a clean separation of concerns. This makes it easier to understand, maintain, and extend the codebase.

10. Scalability

Component-based architectures are well-suited for building large and complex applications. As the application grows, you can continue to create new components and compose them together without significantly increasing the complexity of the codebase.

Popular libraries and frameworks, like React in the web development space, implement component-based architecture. This approach is widely used and praised for its effectiveness in building modular, maintainable, and scalable applications.

Day 3: Exploring Flutter

Setting Up the Development Environment:

- Operating Systems: Windows 10 or later (64-bit), x86-64 based.
- Disk Space: 2.5 GB (does not include disk space for IDE/tools).
- Tools: Flutter depends on these tools being available in your environment.
- Windows PowerShell 5.0 or newer (this is pre-installed with Windows 10)

Get the Flutter SDK.

Download the following installation bundle to get the latest stable release of the Flutter SDK:

Download Flutter

For other release channels, and older builds, check out the SDK archive.

Extract the zip file and place the contained flutter in the desired installation location for the Flutter SDK (for example, %USERPROFILE%\flutter, D:\dev\flutter).

Update your path

If you wish to run Flutter commands in the regular Windows console, take these steps to add Flutter to the PATH environment variable:

From the Start search bar, enter ‘env’ and select Edit environment variables for your account.

Under User variables check if there is an entry called Path:

If the entry exists, append the full path to flutter\bin using ; as a separator from existing values.

If the entry doesn’t exist, create a new user variable named Path with the full path to flutter\bin as its value.

Run `flutter doctor`

```
C:\src\flutter>flutter doctor
```

Open a terminal or command prompt and navigate to the directory where you want to create your Flutter project.

Use the following command to create a new Flutter project:

Setting up a new project.

Building a Simple App:

```
flutter create my_flutter_app
```

Navigate into the newly created project directory:

Building a Simple App:

Now that you have your project set up, let's create a basic app using Flutter.

Understanding the widget-based architecture.

Flutter uses a widget-based architecture for building user interfaces. Everything in Flutter is a widget, from simple elements like text and buttons to more complex elements like containers and layouts. Here's a basic understanding:

Widgets:

In Flutter, a widget is a description of part of the user interface. It could be a button, a text input field, or even an entire screen.

Widgets can be classified into two main types:

Stateless widgets: These are immutable and cannot change after they are built. They are used for UI elements that do not change over time.

Stateful widgets can change over time in response to user actions or other factors. They maintain some form of internal state.

Widget Tree:

Flutter apps are built using a tree of widgets. This tree represents the hierarchy of UI elements in your app.

The build method in a Flutter widget is where you define the structure of your UI. It returns a widget tree that describes the entire user interface.

Hot Reload:

One of the powerful features of Flutter is hot reload. This allows you to make changes to your code and see the results instantly without restarting the app. It's a great productivity booster during development.

Layouts and Composition:

Flutter provides a variety of layout widgets that allow you to arrange other widgets in specific ways. These include Container, Column, Row, Stack, etc.

Material Design and Cupertino:

Flutter provides two sets of widgets to create apps with either Material Design (Android) or Cupertino (iOS) style. This allows you to create apps that look native on both platforms.

.

Day 4: Introduction to App Development Tools

Overview of Integrated Development Environments (IDEs):

App development tools are software programs that assist developers in creating, testing, and deploying applications. These tools include Integrated Development Environments (IDEs), code editors, compilers, emulators/simulators, and version control systems.

Integrated Development Environments (IDEs) are comprehensive software applications that provide a complete set of tools for app development. They typically include a code editor, debugger, compiler, and various automation tools integrated into a single environment. IDEs streamline the development process and enhance productivity.

Introduction to VS code for cross-platform development.

Visual Studio Code (VS Code) is a free, open-source code editor developed by Microsoft. It's highly customizable and lightweight, and it supports a wide range of programming languages. While it's not a full-fledged IDE, it provides a robust development environment with extensions that can transform it into a powerful tool for various types of development, including app development.

Understanding the features and functionalities of each IDE.

Multi-Language Support:

VS Code supports a wide array of programming languages out-of-the-box, including JavaScript, TypeScript, Python, Java, and more.

Extensions:

One of the standout features of VS Code is its extensibility. You can install extensions to add new languages, debuggers, themes, and even specific tools for various frameworks or platforms.

IntelliSense:

VS Code provides intelligent code completion suggestions based on your code context, making it easier to write code quickly and accurately.

Integrated Terminal:

VS Code includes a built-in terminal, allowing you to run commands directly within the editor.

Git Integration:

VS Code has built-in Git support, allowing you to manage version control directly within the editor.

Debugging Tools:

It provides powerful debugging tools with support for multiple languages. You can set breakpoints, inspect variables, and step through your code.

Task Automation:

You can define custom tasks and workflows to automate repetitive tasks, like building, testing, and deploying your applications.

Live Server and Hot Reloading:

For web development, extensions like Live Server provide live previews of your project, and Hot Reloading allows you to see changes instantly without manual refreshes.

Integrated Version Control:

VS Code provides Git integration and support for other version control systems. You can commit, pull, push, and manage branches without leaving the editor.

Themes and Customization:

VS Code offers a wide range of themes and customization options, allowing you to personalize the editor to suit your preferences.

Day 5: Testing and Debugging Mobile Apps

Unit Testing:

Unit testing involves testing individual units or components of your application in isolation to ensure they work as intended. In mobile app development, this typically means testing specific functions or methods within your code.

Writing and executing unit tests for your app.

Select a Testing Framework:

For iOS (Swift) and Android (Kotlin/Java), various testing frameworks are available. For example, XCTest is for iOS, and JUnit is for Android.

Create Test Cases:

Write test cases that cover different scenarios for each unit of code. These test cases should check the expected behavior against the actual behavior.

Run Tests:

Use the testing framework's tools to run your unit tests. This can be done directly within your development environment.

Analyze Results:

Review the test results to ensure that all tests pass. If a test fails, investigate the issue and adjust your code as needed.

Manual Testing:

Manual testing involves human testers interacting with the app to evaluate its functionalities, UI/UX, and overall user experience.

Exploratory testing and user acceptance testing.

Exploratory Testing:

This involves exploring the app without predefined test cases. Testers use their creativity and experience to discover potential issues.

User Acceptance Testing (UAT):

This testing phase involves real users (or representatives of the target audience) testing the app to ensure it meets their expectations and requirements.

Debugging Tools and Techniques:

Debugging is the process of identifying and fixing issues or bugs in your code.

Using Built-in Debuggers in Xcode and Android Studio:

Xcode (iOS):

Set Breakpoints: Place breakpoints in your code to pause execution and inspect variables.

Step Through Code: Use the debugger to step through your code line by line.

View Variables: Examine the current state of variables and objects.

Debugging Console: Use the console to print messages and view output.

Android Studio (Android):

Logcat: View system logs and messages from your app to identify issues.

Set Breakpoints: Similar to Xcode, set breakpoints to pause execution.

Evaluate Expressions: Inspect and evaluate expressions during runtime.

Profiler: Analyze app performance and memory usage.

Troubleshooting Common Issues:

Null Pointers (Java/Kotlin):

Pay attention to NullPointerExceptions. Ensure that variables are properly initialized before use.

Memory Leaks:

Use memory profiling tools to identify and fix memory leaks in your app.

UI Issues:

Check layout files and ensure they are correctly configured. Use layout inspectors to visualize UI hierarchies.

Network Issues:

Test the app on different network conditions using emulators or physical devices.

Compatibility Issues:

Test the app on different devices and Android versions to ensure compatibility.

Week-9 App Deployment and Distribution

Best practices for app optimization

The following best practices help optimize your app without sacrificing quality.

Use Baseline Profiles

Baseline Profiles can improve code execution speed by 30% from the first launch, and can make all user interactions—such as app startup, navigating between screens, or scrolling through content—

smoother from the first time they run. Increasing the speed and responsiveness of an app leads to more daily active users and a higher average return visit rate.

Use a startup profile

A startup profile is similar to a Baseline Profile, but it is run at compile time to optimize the DEX layout for faster app startup.

Use the App Startup library

The App Startup library lets you define component initializers that share a single content provider, instead of defining separate content providers for each component you need to initialize. This can significantly improve app startup time.

Lazily load libraries or disable auto-initialization

Apps consume many libraries, some of which might be mandatory for startup. However, there can be many libraries where initialization can be delayed until after the first frame is drawn. Some libraries have an option to disable auto-initialization on startup or have an on-demand initialization. Use this option to postpone initialization until necessary to help boost performance. For example, you can use on-demand initialization to only invoke WorkManager when the component is required.

Use ViewStubs

A ViewStub is an invisible, zero-sized View that you can use to lazily inflate layout resources at runtime. This lets you delay inflating views that aren't necessary at startup until a later time.

If you are using Jetpack Compose, you can get similar behavior to ViewStub using state to defer loading some components:

```
var shouldLoad by remember {mutableStateOf(false)}

if (shouldLoad) {
  MyComposable()
}
```

Load the composeables inside the conditional block by modifying shouldLoad:

```
LaunchedEffect(Unit) {
  shouldLoad = true
}
```

This triggers a recomposition that includes the code inside the conditional block in the first snippet.

Optimize your splash screen

Splash screens are a major part of app startup, and using a well-designed splash screen can help improve the overall app startup experience. Android 12 (API level 31) and later includes a splash screen designed to improve performance. For more information, see Splash screen.

Use scalable image types

We recommend using vector drawables for images. Where it's not possible, use WebP images. WebP is a image format that provides superior lossless and lossy compression for images on the web. You can convert existing BMP, JPG, PNG or static GIF images to WebP format using Android Studio. For more information, see Create WebP images.

Additionally, minimize the number and size of images loaded during startup.

Use Performance APIs

The performance API for media playback is available on Android 12 (API level 31) and later. You can use this API to understand device capabilities and perform operations accordingly.

Prioritize cold startup traces

A cold start refers to an app starting from scratch. Meaning, the system's process doesn't yet create the app's process. Your app typically starts cold if you launch it for the first time since the device booted or since the system force-stopped the app. Cold starts are much slower because the app and system must perform more work that isn't required on other startup types—like warm and hot starts. System tracing cold startups gives you better oversight into app performance.

Understanding the Importance of Android App Performance Optimization

App performance significantly impacts user satisfaction and retention. Understanding how poor app performance leads to frustrated users and high churn rates while optimized performance contributes to positive user experiences and improved retention is crucial. This enables enterprises to improve their brand reputation while generating greater ROI. Furthermore, optimizing app performance can positively impact your app's visibility in app stores, as search algorithms often prioritize apps with good performance metrics.

However, enterprises are often baffled with complexities while optimizing their Android app's performance. Some of the most common challenges include:

- **Device fragmentation:** Android is a fragmented platform with various devices and configurations in use. This can make optimizing apps for all devices difficult, as each device may have different hardware capabilities and software versions.
- **Code complexity:** Android apps can be complex, with a large number of code components. This often makes it difficult to identify and optimize performance bottlenecks.
- **Third-party libraries:** Android apps often rely on third-party libraries. These libraries can introduce performance problems, as they may be incompatible with the latest Android versions or may not be optimized for Android.
- **User behavior:** User behavior can also impact app performance. For example, apps used heavily for gaming or video streaming can put a strain on the device's resources, leading to performance problems.

Check out: Android vs. iOS App Performance Testing - How are These Different?

How does performance optimization benefit Android apps?

Optimizing your Android app's performance brings numerous benefits. It not only helps enhance user experience but also improves the overall reputation and marketability of the application.

Ensuring efficient resource utilization and responsiveness can increase user satisfaction, reduce app abandonment rates, and drive higher user engagement and retention.

Why are businesses focusing on Android app optimization to drive growth and revenue?

Android apps with optimized performance help drive business growth by improving user experience, which leads to increased engagement and revenue. When an app is slow or buggy, users are more likely to uninstall it or abandon it. This can lead to lost revenue from in-app purchases, subscription fees, or advertising. Optimized apps, on the other hand, are likelier to be used for longer time periods. This is because users will more likely use an app that is reliable, fast, and user-friendly.

Here are the primary reasons why every Android business prioritizes optimizing app performances for greater ROI and how it happens:

- **Increased customer satisfaction:** When apps are fast and reliable, users are more likely to be satisfied. This leads to improved customer loyalty and repeat clients.
- **Improved brand reputation:** A well-performing app can help to improve a business's brand reputation. This is because users are likelier to have a positive impression of a business that offers high-quality apps.
- **Better app store visibility:** App store algorithms consider various factors when determining the ranking and visibility of apps. Performance metrics, such as app responsiveness and crash rates, are often taken into account. Optimizing your app's performance can improve its chances of being discovered and featured by app stores, leading to increased organic downloads.
- **Increased market share:** Optimized apps can help businesses to increase their market share. This is because they can attract new users and keep existing users engaged.
- **Reduced support and maintenance costs:** A well-optimized app experiences fewer performance-related issues, resulting in reduced support and maintenance efforts. By investing in performance optimization upfront, organizations save time and resources which would otherwise be dedicated to addressing user complaints and fixing performance-related bugs.

Overall, there are many benefits to optimizing Android apps for performance. By doing so, businesses can improve user experience, increase their competitive advantage, and drive business growth.

Critical KPIs that impact app performance

To achieve success in this competitive app development space, it is essential to prioritize app performance optimization. Key Performance Indicators (KPIs) serve as critical metrics that help measure the effectiveness of your app's performance optimization efforts. Tracking and analyzing these KPIs help organizations gain valuable insights into the app's performance and make data-driven decisions to enhance its overall user experience. When it comes to optimizing app performance, here are some important KPIs to consider:

1. **App Launch time:** This metric measures the time the app takes to load and become usable after being launched. It indicates how quickly users can start interacting with the app and is critical to providing a smooth user experience.

2. **Screen rendering time:** Screen rendering time measures how long it takes for the app to display the UI elements and content on the screen. Optimizing screen rendering time ensures that the app feels responsive and avoids any lag or delays when navigating between screens.

3. **Network performance:** Network performance KPIs include metrics such as network latency, response time, and data transfer rate. Improving network performance ensures that the app efficiently communicates with servers, loads data, and delivers a seamless experience even in varying network conditions.

4. **Memory usage:** Monitoring and optimizing memory usage is crucial to prevent memory leaks and excessive memory consumption. KPIs related to memory usage include memory footprint, allocation rate, and peak memory usage. Efficient memory management ensures optimal app performance and avoids crashes or slowdowns.

5. **Battery consumption:** Battery consumption KPIs measure the impact of the app on the device's battery life. Optimizing battery consumption is important to ensure that the app doesn't drain the battery excessively, leading to a poor user experience and decreased device performance.

6. **Crash rate:** The crash rate metric tracks the frequency of app crashes. Reducing the crash rate is a key performance goal, as crashes severely impact user satisfaction and can lead to negative reviews or uninstallation. Monitoring crash reports and promptly fixing bugs and issues helps maintain app stability.

7. **Conversion and engagement metrics:** These KPIs focus on the user's interaction and engagement within the app. Metrics such as user retention, session duration, and conversion rates (e.g., sign-ups and purchases) indicate the overall success and effectiveness of the app in driving user engagement and achieving business goals.

8. **App size and download time:** Optimizing the app's size and download time is important for user acquisition and retention. Users prefer smaller app sizes that download quickly, especially in regions with limited bandwidth. Monitoring these metrics and implementing strategies to reduce app size and improve download speed can positively impact user experience.

Also check: Performance Testing Challenges Faced by Enterprises and How to Overcome Them

Strategies that drive Android app performance optimization

Android apps are becoming increasingly complex and demanding, putting a strain on the resources of mobile devices. Therefore, it is essential for developers to optimize their apps for performance. There are several optimization strategies that can be deployed to improve the performance of an Android app. Some of the most common strategies include:

- **Reduce the app size:** The larger the app, the more resources it will take to run. This can lead to performance problems on devices with limited resources. Developers can reduce the app size by using a lightweight framework or library, removing unused code and resources, and compressing images and videos.

- **Minimize app startup time:** It is critical to reduce initialization overhead by optimizing resource initialization and deferring non-essential operations until they are required. This assists in improving the app's startup time. Additionally, it is important to identify and

optimize any code blocks that contribute to increased startup time, such as heavy computations or resource-intensive operations.

- **Use efficient code:** Developers should use efficient code to avoid wasting resources. This includes using the correct data types, avoiding unnecessary loops and conditional statements, and using caching techniques to store frequently used data.
- **Ensure efficient memory management:** Effective memory management is crucial for optimizing your Android app's performance. Identifying and fixing memory leaks, which can result in excessive memory consumption and degraded app performance, is crucial to achieving this. Organizations should leverage appropriate memory profiling tools to detect and resolve these leaks effectively. Additionally, teams can optimize memory usage by implementing strategies such as object pooling, resource recycling, and efficient data structures. These techniques help minimize the memory footprint of your app, resulting in improved overall performance.
- **Testing and profiling:** Testing and profiling are essential aspects of optimizing app performance. Conducting comprehensive performance testing allows you to identify bottlenecks, measure app response times, and ensure that your app meets performance expectations. By utilizing profiling tools such as Android Profiler and third-party libraries, you can analyze CPU usage, memory allocations, and network behavior. This analysis provides valuable insights that help in optimizing performance and addressing any potential performance-related issues.
- **Use background threads for long-running tasks:** Long-running tasks, such as network requests and database queries, should be executed in background threads. This will prevent the main thread from being blocked, which can lead to performance problems.
- **Optimize across different Android versions:** To optimize your app for different Android versions, leveraging version-specific enhancements that capitalize on performance improvements and features introduced in specific Android versions is crucial. This allows you to enhance your app's performance and ensure compatibility with different OS versions. Additionally, conducting thorough compatibility testing on various Android devices and OS versions is essential to guarantee consistent and reliable performance across different configurations. By implementing these strategies, you can effectively optimize your app for diverse Android environments and provide a seamless experience for all users.
- **Optimize battery consumption:** Optimizing battery consumption in Android applications involves minimizing background operations and using power-friendly APIs. By reducing unnecessary battery-draining activities and implementing efficient scheduling mechanisms, you can extend battery life. Additionally, leverage Android's power-friendly APIs like JobScheduler to execute background tasks at optimal times, further conserving battery power. These strategies enhance battery performance and improve the overall user experience on Android devices.
- **Optimize network operations:** To optimize your app's network operations, focus on minimizing requests and implementing efficient caching techniques. To minimize network requests, organizations can employ various techniques such as data caching, request bundling, and batch processing. Implement caching with HTTP and in-memory caching to store frequently accessed data, reducing latency and improving responsiveness. Streamline network operations for a smoother user experience and optimal app performance.

- **Improve app responsiveness:** To enhance app responsiveness, optimize UI rendering through techniques like View recycling, background threading, and lazy loading. Implement asynchronous operations to prevent UI unresponsiveness and execute time-consuming tasks in the background. By separating lengthy operations from the main UI thread, users can seamlessly interact with the app while tasks are processed asynchronously, improving the overall user experience and maintaining app responsiveness.

- **Monitor performance continuously:** Continuous performance monitoring is crucial for ensuring the optimal performance of your Android app. By integrating performance monitoring tools into your app, you can gather real-time insights and effectively identify any performance degradation issues. These tools enable you to closely monitor key performance metrics and promptly detect any anomalies that may impact user experience. Furthermore, by analyzing performance reports and leveraging user feedback, you can prioritize and address performance-related issues promptly. This proactive approach allows you to optimize your app's performance continuously, ensuring a smooth and seamless user experience.

- **Leverage user feedback and testing:** Organizations can effectively optimize their app performances by gathering user feedback that helps identify performance issues that may arise in real-world scenarios. Additionally, conducting thorough testing, including load testing and stress testing, to simulate different usage scenarios and make improvements based on feedback enables companies to ensure app performance under various conditions.

Read: A Guide on Automation Testing for Mobile App Performance Optimization

Annexed to the strategies outlined above, there are a few additional tips that developers can follow to optimize their Android apps for performance. These tips include:

- **Avoid using third-party libraries unless they are absolutely necessary:** Third-party libraries can add to the size of an app and can also introduce performance problems.

- **Use caching techniques to store frequently used data:** Caching can help improve performance by reducing the number of instances data needs to be retrieved from the network or database.

- **Use a memory leak detection tool to identify and fix memory leaks:** Memory leaks can cause an app to run slowly and can eventually lead to the app crashing.

- **Keep the app updated with the latest Android versions:** New versions of Android often include performance improvements.

Leading Android app performance testing tools in the market

1. LoadNinja

LoadNinja, developed by SmartBear, is an advanced performance testing tool that simplifies the process of load testing for your web applications. With its scriptless approach, LoadNinja offers several features and advantages that help optimize testing time and enhance efficiency.

Capabilities:

- Scriptless load test creation & playback

- Real browser load test execution at scale

- VU Debugger for real-time debugging
- VU Inspector for real-time virtual user management
- Cloud-based hosting with no server machine upkeep
- Sophisticated browser-based metrics with analytics and reporting

2. NeoLoad

Tricentis NeoLoad is a robust performance and load-testing solution designed to deliver responsive, scalable, and reliable applications. With NeoLoad, you can proactively address performance issues and ensure a seamless user experience.

Capabilities:

- RealBrowser for accurate browser-based testing and capturing the end-user experience
- DevOps and Agile integration, leveraging APM tools and functional testing for precise diagnostics and metrics
- Codeless test design for faster creation of complex tests
- Actionable insights through high-level dashboards and detailed metrics

3. StressStimulus

StresStimulus is a comprehensive testing solution that targets challenging application scenarios. With its unique features, it offers efficient testing capabilities that other tools may struggle with.

Capabilities:

- Automated error fixing with proprietary autocorrelation
- User action recording and replay for realistic load testing
- End-to-end Test Wizard for a streamlined testing process
- On-premise or cloud testing with multiple load generators
- Stand-alone tool or Fiddler add-on compatibility
- Script export to Visual Studio format

4. HeadSpin

HeadSpin empowers organizations to optimize digital experiences through its powerful performance testing capabilities. With HeadSpin, you can continuously monitor, identify, resolve, and optimize performance issues across applications, devices, networks, and 3rd-party interfaces.

Capabilities:

- Performance testing capabilities for enhanced digital experiences
- Continuous monitoring and issue resolution across various platforms
- Global Device Infrastructure for real-world data on devices, networks, and locations

How HeadSpin — a global omnichannel testing platform is leveraged by businesses for optimizing app performance and perfecting the user experience

HeadSpin offers data science driven app performance testing capabilities by identifying and resolving performance issues proactively across different apps, devices, and networks. With its powerful capabilities and intuitive interface, HeadSpin enables developers, testers, and product and QA teams to assess, monitor, and improve the performance of their Android applications.

1. Data science driven performance testing

HeadSpin's data science capabilities help to continuously monitor and optimize the quality of experience. The Platform helps capture hundreds of custom KPIs that impact app performance and user experience. The Platform provides real-time performance monitoring, giving you detailed insights into various metrics, such as app startup time, screen rendering, CPU and memory usage, battery consumption, and network requests.

2. HeadSpin's global device infrastructure

With a robust device infrastructure, HeadSpin allows enterprises to test their apps on real SIM-enabled devices across 90+ locations worldwide. Enterprises can obtain real-world data and eliminate ambiguity from different devices, networks, carriers, and locations to streamline testing.

3. Proactive AI driven issue detection

HeadSpin's advanced AI capabilities help surface performance issues automatically before they impact users. The Platform further helps identify the root causes of the issues and predict issues based on historical data. With these valuable performance data, enterprises can identify bottlenecks, pinpoint areas for improvement, and prioritize optimization efforts effectively.

4. Insightful visualizations

HeadSpin provides comprehensive reporting and analytics features and generates detailed reports on performance trends, key metrics, and comparison data across different devices and network conditions. These reports help you track the progress of your Android performance optimization efforts, measure the impact of optimizations, and make informed decisions based on data

Importance of improving app performance

Introduction

According to a report by Kleiner Perkins, more than 15 % of internet traffic worldwide is due to mobile apps. There are around 1.5 billion mobile users globally. Hence, high-performing mobile apps are highly significant for your enterprise.

Mobile App is analogous to an online salesperson for your enterprise. As an ill-performing salesperson can be detrimental to the prosperity of the business, a mobile app with unsatisfactory performance and multiple user experience issues can defame your business to a great extent.

A poor-performing app can undermine your company's prospects and degrade its reputation.

Here is why app performance optimization is of utmost importance.

- 36 % of users agree that a slow app produces a "lower opinion of the company."
- 49 % of e-commerce customers expect the page to be loaded in 2 seconds or even less.

Importance of app performance

In fact, in just half a generation, technology has completely redefined the word 'app.' More than six million mobile apps currently crowd the Google Play Store and the Apple app store – the repo(s) from where most users download apps.

Statistically, reports show that around 70% of mobile app users will desert an application with a high loading time. A delay in response of as low as 1 sec can cause nearly 7% of loss in conversion. Almost 1 in every two apps gets uninstalled within its initial month of download.

These statistics confirm the notion that App performance has an indispensable impact on its user experience. Performance is synonymous with the measure of the success or failure of the app. Both user retention and prolonged usage of an app depend on its performance.

Also, reasonable people who constitute more than 40% of users will use the app created by a competitor if they encounter a poor mobile experience with any app. Simultaneously, the scenario has an even more severe impact than just uninstalling the app for lousy performance — customers are inclined toward the competitors to solve their needs. Hence, optimal app performance is highly significant.

Ways to Improve App Performance

It is a common misconception that performance is usually synonymous with only the front-end or the UX. Optimal performance is based on both the back-end and front-end of the app.

For example, consider a situation where the app initially gets loaded quickly, and the user can navigate through it smoothly — until a specific time when the user needs to make a payment. Suddenly at that moment, the app crashes; the app crashes at the peak of user activity. This is also regarded as a performance problem. Performance issues encompass many kinds of stuff like

- how slow or fast the app loads
- When and why the app crashes
- How seamlessly do certain features (like checkout) run

and many more.

Although inadequacy to offer an optimal user experience is the fundamental reason for ill-optimized front-ends, these are not the only measures of app performance.

Optimizing mobile apps using performance testing tools requires device and system at peak performance levels. Given below are a few hacks to improve app performance by optimizing both the device and system:

1. **Check network performance**
2. When it comes to mobile apps, understanding their functionalities in various internet connections like Wi-Fi or 4G, or 5G is to be primarily considered. After checking this using performance testing tools, how the existing performance can be enhanced should be taken up next. Given below are some quick ways to improve app performance.
3. · Deploy an optimum back-end server with high performance to enhance the app response time

4. Maintain a native database that guarantees the safety of user data even if the server goes down
5. Decrease the URL redirects as much as possible on your screen
6. Try to avoid app crashes by all means
7. Reduce or limit server requests by an app to complete the user request
8. Have a backup server for users to access your app even if the primary server is down
9. Check device performance

Although the consequences are manifested through the front end, the actual mess lies in the back end. Apps that are glitchy and contain bugs often consume battery life and memory. Unambiguously, they are in urgent need of optimization. However, starting by evaluating the device itself is a good practice.

- Optimize screen rendering times

It is true that users want speed, but not at the cost of rendering the app well. It is useless if the pages or the app opens fast, but users cannot engage or interact with it quickly. Hence for an optimal rendering of the app, developers should consider the following:

1. Is the app rendering accurately on multiple screen sizes or operating systems? This should be kept in mind that a page created for a computer screen will not render correctly on a mobile and vice-versa. Hence developers should scale the image appropriately for various OS and screen types and sizes
2. Developers should also check the consistency of the images and font sizes. It is one of the primary ways to improve screen rendering times.

- Limit memory and energy consumption

Battery life and memory are crucial aspects of a mobile device. Hence, an app with less memory and battery drain results in a better and improved user experience. For implementing this, the following can be considered:

1. Push notifications and memory leaks can increase memory consumption.
2. Incessant app usage drains battery life quickly. Therefore, developers should restrict unnecessary energy drains while fabricating the app. Also, users must be alerted to turn off GPS and Bluetooth when not in use. This will help preserve battery life and portray the app as user-friendly and consumer-minded.
3. Even after achieving optimal user experience, the app's back end should always be considered for further enhancements.

Improving app performance (and why it's so important)

App performance relies on launch and load times, app size, frame rate, compressing, and much more. It's essential to optimize these facets of app performance to improve the user experience ASAP, and teams **can** make an impact by focusing on a handful of tactics.

What does app performance actually look like?

Sometimes, to understand something, you must first understand what it is not.

When talking about app performance, we are not talking solely about key performance indicators (KPIs), but about *performance*. The amount of energy your app uses, the amount of time it takes to load, and/or how fast it responds to user interactions, things like that.

Consider this scenario: You leave the house without having time for your coffee. You've been reluctant to use delivery or food apps in the past because of concerns about your privacy. Today, you throw caution to the wind and install a big-brand coffee shop app so that you can hit the drive-thru and quickly pick up your mobile order – just like their commercials promise.

Here's the problem: the initial load time of the app is taking longer than being late for making that cup of coffee at home or waiting in line at the store. Most reasonable people — even when coffee deprived — will abandon a slow app, never to return.

In this scenario, application performance looks like an app's initial launch being successful and you being caffeinated. But it is not limited to just initial launch time; performance is how your app functions, how fast it **How important is app performance (really)?**

Most (70%) of mobile app users will abandon an app if it takes too long to load. A one-second delay in response can result in a 7% loss in conversion, and nearly one in every two apps are uninstalled within the first month of download.

App performance can therefore make or break its success, as it's directly tied to both prolonged usage and user retention.

So, let's say our caffeine-challenged user had a fast initial load time and is now navigating through the app. Our user finds their drink and "clicks" to order, but nothing happens. Or, in reality, it takes the app 10 full seconds to offer the user size and sweetener options.

Our user wouldn't know this, however, because they have already abandoned the app.

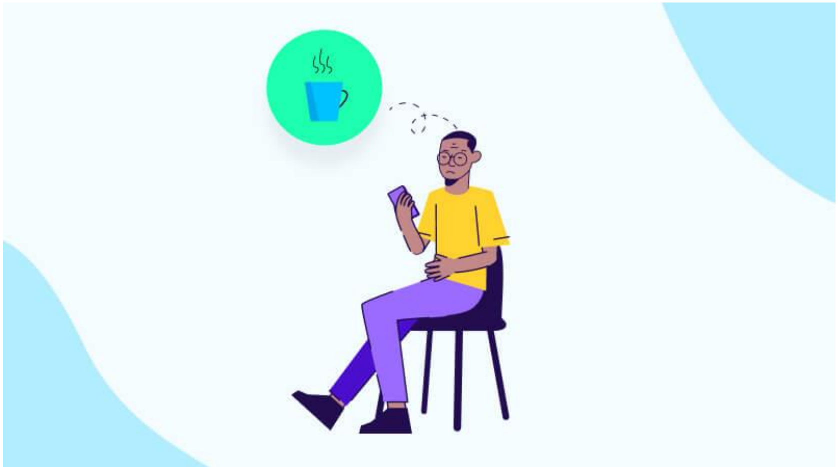
While the above is just an example, statistics show that 40% of users will jump ship in favor of a competitor after a bad mobile experience. In other words, this scenario ends with more than just abandoning an app for poor performance — users are going to competitors to solve their problems.

App performance is clearly very important.

How to improve app performance

Because we only know what we experience as users, performance is usually confused with only UX. However, optimal app performance relies on both the front-end and back-end of app development.

Think back to the coffee-less user who took a leap of faith on an app that didn't deliver because it couldn't load.



Imagine now that the app loaded, and the user navigated seamlessly — until it came time to pay. If an app crashes at peak user activity — that is also a performance issue.

Performance problems range from how fast or slow your app loads, when and if your app crashes, how smoothly certain features (like checkout) work — and so much more.

And while it is true that failing to offer a satisfactory user experience is most likely the result of poorly optimized front-ends – like suboptimal data loading or image optimizations, – these are not the only indicators of how an app is performing.

Mobile app optimization requires that both system and device are at peak performance levels. To help you improve your mobile app performance, here are a few system and device tips:

1. Check network performance

When it comes to mobile apps, the first thing to consider is performance on

functionalities like Wi-Fi, 4G, or 5G.

So, when optimizing mobile apps, a good place to start is understanding how it performs on different networks and how that performance can be improved.

GUIDE

Show me the money! Your definitive guide to app conversions

Increase conversions now

Optimize network speeds and application response times

Whether caffeinated or not, users have little-to-no patience for slow launch or load times. Developers should ensure that their apps can function across a variety of networks before issues arise and ruin their app's experience.

Good news, though — there are a few ways to immediately improve app performance:

1. Reduce the number of URL redirects on your screens.
2. Still using Flash? Stop. Find a better, mobile-friendly alternative.
3. If the back-end server response time is sluggish, your app will be sluggish. The simplest way to avoid this is to avoid reliance on free or inadequate hosting services that offer minimal-to-no support. The easy fix is to invest in a high-performance server that avoids these types of issues.
4. Keeping a native database guarantees that even if a server goes down, the user's data is not in danger. Additionally, a backup server ensures that users can access your app even if the main server is down, and it maintains the launch and load speeds users have come to expect.

Avoid app crashes by all means necessary

The devastation of an app crash at peak user experience is real. Think about our imaginary user and the caffeine highs and lows that fictional app has put them through. The ultimate front-end performance failure is an app crash.

The best way to address this is to constantly evaluate, collect, and address the data on the following:

1. Percentage of users that suffer a crash in a given amount of time
2. How often the app is crashing in a given amount of time
3. Rates of hang time and failed network error encounters

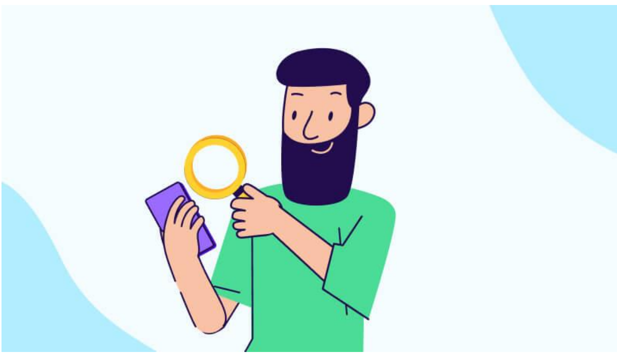
But let's pause before tackling the front-end considerations and data collection.

In the back-end, developers should consider the most fundamental reason for an app crash: the number of requests an app must make with a server to complete a user request. The more requests, the more opportunity for an app to slow down and crash. The fewer requests — you get the point.

So, the easiest back-end fix for this is to reduce and limit the number of calls and requests an app needs to make with a server.

Keeping this in mind during the development stage and then collecting crash data from UX to monitor and avoid crashes — is a perfect example of the cause-and-effect relationship between the front- and back-ends of mobile app development.

2. Check device performance



Even though the front-end manifests the consequences, the problems lie in the back-end. Apps that are buggy, glitchy, consume memory and battery life — are clearly in need of urgent optimization, but another good place to start is evaluating the device itself.

Optimize screen rendering times

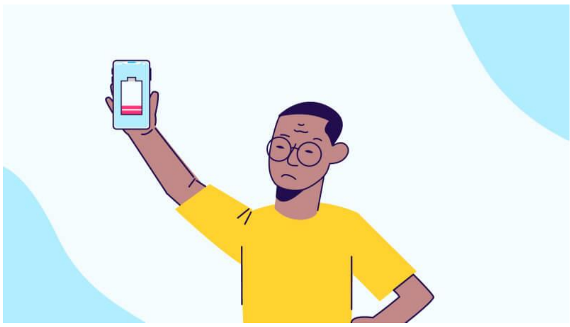
Yes, users want speed, but what happens if an app does not render well? How quickly can a user interact with and use the app or page? Even if the app is launched and the pages are loading, what's the point if the user cannot interact and engage?

Developers should consider the following:

1. Again, does the app render well on different operating systems (Apple/Android) or screen sizes? Remember that content created for a desktop screen will probably not render well on a mobile screen. Users do not want to zoom in or out, so when developing an app, images need to be scaled appropriately.

2. How consistent are image and font sizes? Consistency is one (easy) way to improve screen rendering times. Making font and image sizes uniform limits screen resizing while scrolling, allowing users to interact with the app immediately.

Limit energy and memory consumption



Memory and battery life are actually crucial to mobile users, as our phones have only so much to spare. Therefore, the less memory and the less battery drain an app can cause, the better your UX will be as a result.

Things to consider:

1. Memory leaks and push notifications are two examples of what can affect memory consumption.

2. Continual app usage can quickly drain battery life, so be sure to consider unnecessary energy drains in the development phase. Alternatively, consider alerting users to turn off features

like GPS and Bluetooth when they are not in use, which will help them preserve battery life and position your app as friendly and consumer-minded..

s systems load, and how it responds to user interactions.

11 Mobile Apps That Failed and How To Learn From Their Mistakes

It's fair to say that since then, the stores have undergone significant changes in design, functionality, and app databases. However, the aim of these stores remains the same: to provide a platform for mobile users to browse and download apps. Apps are so much a part of our lives that Apple and Google release annual roundups of the best apps and games.
In 2020, the App Store announced that Wakeout!, a family-friendly exercise and movement app, earned the iPhone App of the Year*. The app also won app of the year in 2017. For Google Play, there was a similar theme to its 2020 winner. Mood-altering app Loóna won best app of the year, and according to Google Play data, it's been downloaded 500,000 times.
For these app developers, it must feel great to receive this recognition from Apple, Google, and, of course, their users. However, for every success story, there are hundreds of apps that fail. According to a 2018 Gartner study, less than 0.01% of consumer mobile apps will be considered a financial success by their developers*.
With millions of apps now available in Google Play and the iOS store, how many have missed the mark with consumers? In this roundup, we highlight 11 apps that came up short, and crucial lessons that developers and marketers can learn. Check out our infographic below to learn how to win from mobile app failures.

Why Do Mobile Apps Fail? A Few Theories

With a 0.01% success rate, why do app developers even bother? While there are a number of theories as to why mobile apps fail, the main reasons fall under these three categories:

1. There is such a thing as a free app

According to Statista, as of March 2021, 93% of all iOS applications were available for free compared to 97% in Google Play*. With so many free apps available, it's no wonder that users are hesitant to pull out their wallets for apps that have a price tag. This makes it difficult to monetize apps and pump money back into app development.

2. There are *so many* apps

In the time it will take you to read this article, thousands of apps were added to the Google Play store. In fact, 3,739 apps are added to Google Play every day*. The demand for new content has driven the global mobile industry and widened the app pond. However, progress is a double-edged sword and this demand and growth means more competition. Bottom line: the definition and standards for the 'best' app are harder to match.

3. There is a hefty build cost for an app

According to industry insiders, it can cost as little as $1,000 for a simple iOS app and as much as $150,000 for a high-tech Android app*. It can be a tough place to start, and as a result, profitability and future investment can be difficult to maintain for a large number of apps.

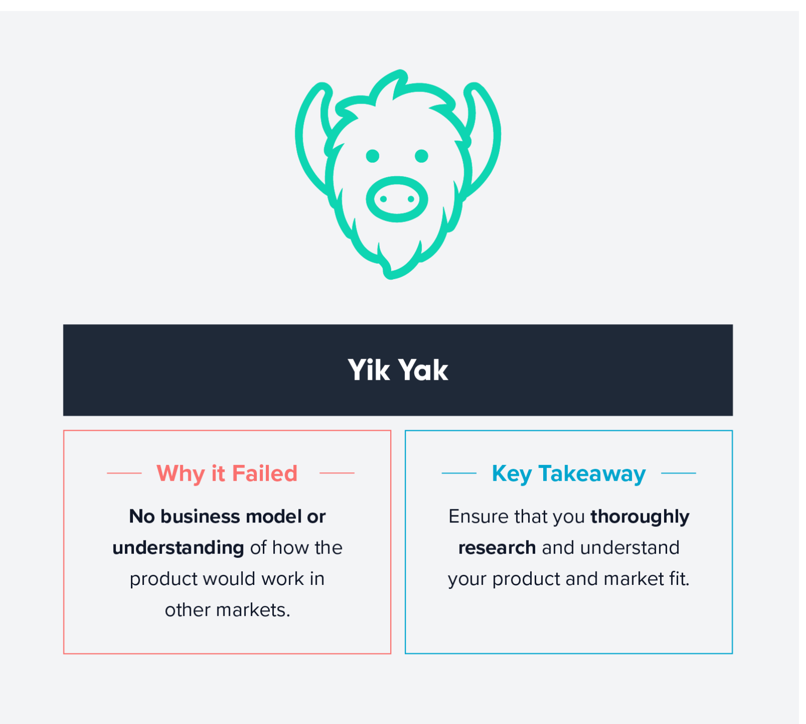
Failure to Launch: 11 Mobile Apps That Lost Their Value

With all of this in mind, we decided to look at 11 mobile apps across gaming, technology, media, transportation, social networking, ecommerce, and finance that failed to stay the course.

Yik Yak

- **App Life:** 2013 to 2017
- **Synopsis:** An anonymous, location-based messaging app for college students
- **Market:** Social networking
- **Claim to fame:** Investors once valued Yik Yak at $400 million

Why it failed: This was an initially popular app that didn't have a business model or firm understanding of its product or market fit. The anonymity that once attracted users to the app later resulted in serious complaints around cyberbullying, threats, and harassment. User growth slowed, employees sadly lost their jobs, and the app shut down after only four years.
Key takeaway: Don't just focus on your product. Before you launch an app, ensure that you thoroughly research and understand your product-market fit. How do you keep users happy with your app and how do you increase time spent on the app? How do these answers contribute to your ongoing success and product growth?



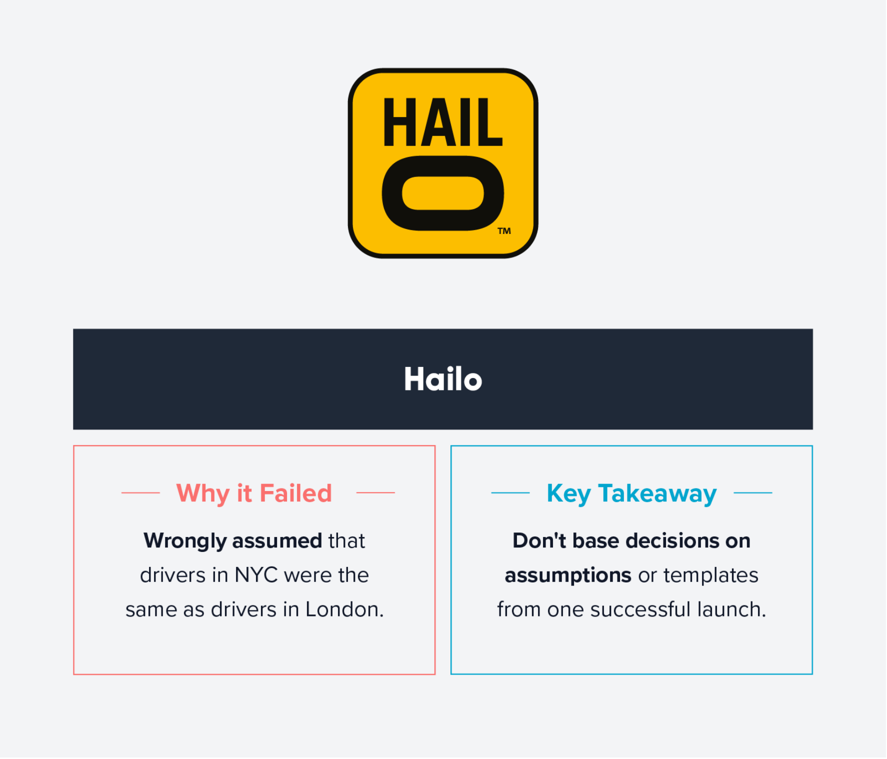
Hailo

- **App Life:** 2012 to 2014 (North America market)
- **Synopsis:** British-born taxi app that matched passengers with taxi drivers
- **Market:** Transportation
- **Claim to fame:** In 2013, the app had 30,000 drivers and 2.5 million users

Why it failed: While the app still operates in Europe, it failed in North America, particularly when it launched in New York City. Here's why: the app assumed that drivers in NYC were the same as drivers

in London. In NYC, cabs are not seen as a luxury like they are in London. Additionally, London has tricky routes while NYC is on an easy-to-use grid system. It also didn't help that Uber also launched in 2011 and was on track to dominate.

Key takeaway: It might sound like common sense, but no two users are the same. The same goes for markets. Don't base your decisions and assumptions on cookie cutter versions of one successful launch. You might have a great idea, but you have to put in the work to succeed.



Vine

- **App Life:** 2012 to 2017
- **Synopsis:** A popular app for making short-form videos
- **Market:** Social networking
- **Claim to fame:** Twitter bought the app in 2012 for a reported $30 million

Why it failed: The short answer: Instagram and its 15-second video capability. While Instagram wasn't the only contributor to Vine's demise, it played a big part in it. As downloads and users declined, so did advertisers and marketers that once considered Vine the platform to invest in. Other issues included an unsustainable business model and inability to adapt and compete with other apps.

Key takeaway: You'll always have competition, but there are a couple of important elements of success: recognizing and embracing change. As technology evolves, so will the needs of users. If you

stick with what you know and miss the boat, it will literally be sink or swim time for your app.

Quibi

- **App Life:** April to October 2020
- **Synopsis:** A short-form video streaming service
- **Market:** Media and entertainment
- **Claim to fame:** The app successfully raised $2 billion before launch

Why it failed: One big reason was the app did not have memorable, binge-worthy content. The app also only planned on people wanting to watch content on their phones. There was no compatible Fire TV app or ability to stream via Airplay or Chromecast. It wasn't a worthy contender for a standalone subscription-based service. Who would pay $5 a month ($8 without ads) for mediocre content and a single-device approach to streaming?

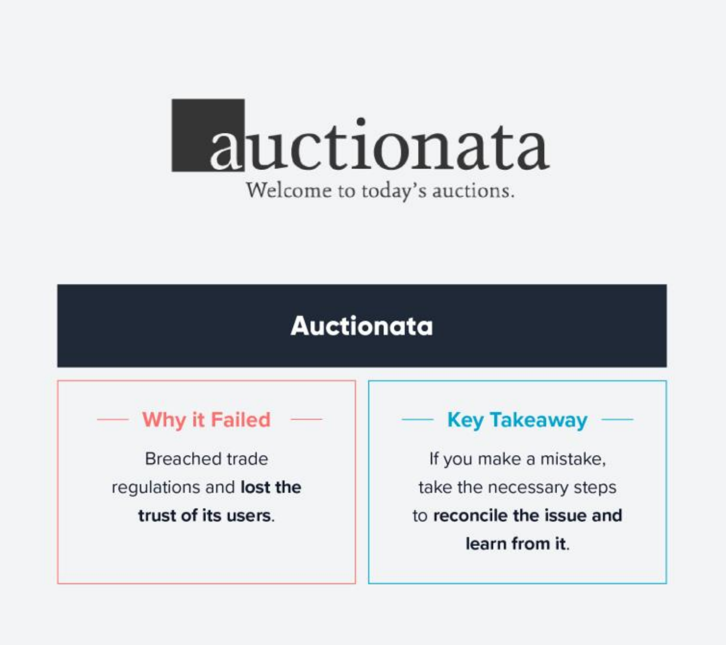
Key takeaway: Why does your app exist? If you can't answer this question, your users definitely won't be able to. Also, be sure that you can answer another important question: why would a user choose your app over Netflix? TikTok? Instagram? Competition is fierce and your app has to stand out for the right reasons.

Auctionata

- **App Life**: 2012 to 2017
- **Synopsis**: An online auction house that specialized in luxury goods and collectibles
- **Market**: Ecommerce
- **Claim to fame:** 250 art experts and $96+ million in funding raised

Why it failed: Here's a great idea: an app that allows live-streamed auctions for selling artifacts, collectibles, and fine art. Here's a bad idea: breaching trade regulations. In 2016, a report was released that accused Aucionata of shill bidding and unethical behavior*. After these issues came to light, the app lost the trust of users and had officially lost its value.

Key takeaway: It's simple: don't be irresponsible, unethical, or bid against your customers. In addition to adhering to policies, regulations, and codes within your industry, make sure that you are always transparent with your app's T&Cs and responsible with user data. If you do make a mistake and lose the trust of your users, take the necessary steps to reconcile the issue and learn lessons from the moment.



Blippy

- **App Life:** 2009 to 2011
- **Synopsis:** A social network that embraced payment sharing, which refers to sharing your credit card payments with friends and strangers
- **Market:** Social networking/finance
- **Claim to fame:** Received backing by Twitter co-founder Evan Williams

Why it failed: Blippy came on the scene BV, i.e., Before Venmo. The concept was just crazy enough to work until a 2010 data breach exposed credit card information to Google. Blippy launched before social and payment oversharing was the 'norm' and the data breach definitely didn't help.

Key takeaway: Apps like Venmo, Instagram, and Snapchat have definitely made oversharing popular, and when done correctly, this tactic and USP can yield the desired results. However, it goes back to the age old question: who cares? If you're trying to solve a problem that doesn't exist or make a product that no one wants, your app will become a blip on the radar.

Pay by Touch

- **App Life:** 2002 to 2008
- **Synopsis:** An app and system that offered biometric payment service
- **Market:** Finance
- **Claim to fame:** Convinced Oracle and Accenture executives to join the business and then raised $340 million in funding

Why it failed: To name a few, irresponsible leadership, lack of market research, and lack of marketing. It is a shame as there was real potential and talent behind its name to be a useful payment service. What's worse: when Pay by Touch filed for bankruptcy, clients were left in the dark.

Key takeaway: There's never just one reason an app fails. However, one thing is for certain: never underestimate the importance of effective leadership. Don't be afraid to invest in 'think outside the box' team members and leaders. It's good to be ahead of the curve but don't be the app that is too early to the party and asked to leave.

Pay by Touch

Why it Failed

Poor leadership, market research, and marketing efforts.

Key Takeaway

Invest in **'think outside the box'** team members and leaders.

Punch Quest

- **App Life**: 2012 to present
- **Synopsis**: An arcade-style fighting game about punching your enemies
- **Market:** Gaming
- **Claim to fame:** The app was downloaded more than 630,000 times in its first week

Why it failed: While Punch Quest is still available in the App Store and Google Play, in 2012, it made one giant mistake: the core game offered so much for free that players were understandably hesitant to pay for in-app purchases. The result: despite 600K+ downloads, the game made just over $10,000 in revenue.

Key takeaway: Offering free content, particularly as a teaser or at launch, can be a great marketing tactic. However, this is a great example of a gamble that literally did not pay off. Utilize your customer database and test fee ranges to find out which payment options work best. Test the free-to-play model but if you do offer in-app purchases, they need to be worth it.

PunchQuest

Why it Failed

Struggled to make a profit from a 'free-to-play' model.

Key Takeaway

Utilize customer database and test fee ranges. **Make any in-app purchases worth the extra cost**.

Zulily

- **App life:** 2010 to 2015 (under original owners)
- **Synopsis:** A retailer that sells discounted toys and clothing
- **Market:** Ecommerce
- **Claim to fame:** At one point, Zuilily was valued at $9 billion

Why it failed: The app didn't carry well-known brands, but the popularity of the flash sales was enough to generate a buzz and a decent number of subscribers. Yet it was a marketing mistake that contributed in a big way to Zulily's demise. The app demanded that users share their email addresses **before** they had the opportunity to properly check out the shop. Result: annoyed customers who shopped once and didn't return.

Key takeaway: Thanks to marketing channels like push notifications, apps can alert customers of surprise sales. It's a great marketing tactic to draw in dormant customers and showcase the value that your app can bring to a customer's life. However, don't get greedy with your marketing efforts. Apps should never forget that a user sharing their email address is a gift.

Shyp

- **App Life:** 2014 to 2018
- **Synopsis:** Take a picture of your package, upload to the app, and someone would pick it up for a total charge of $5
- **Market:** Transportation
- **Claim to fame:** Early coverage in *The New York Times* and $62 million in funding

Why it failed: Despite being compared to Uber, the app sadly grew too fast and couldn't keep up with demand. The app failed to adjust its strategy to compensate for this growth, and in the end, it didn't anticipate how many people would jump at the opportunity to offload a tedious task — mailing a package — for only $5.

Key takeaway: The growth of your app looks good on the surface but if there isn't enough substance, your app will be the victim of its own success. As the needs of your users evolve, so must your strategy. Don't just pay attention to industry trends. Anticipate and plan for them.

Google Wave

- **App Life:** 2009 to 2010
- **Synopsis:** A real-time messaging platform
- **Market:** Social networking
- **Claim to fame:** Generated tons of buzz and was deemed 'highly anticipated'

Why it failed: Although Google Wave never reached app users, this is an example of a platform launching too early. It also goes to show that a big name like Google isn't always enough to guarantee success. It was also never totally clear what Google Wave was supposed to be and sadly, after only a year, the wave crashed and burned for the search engine giant.

Key takeaway: Initial buzz for your app is definitely exciting but don't give in to internal or external pressure to launch your app before it's ready. It can be tempting to launch while the attention is on your app but if you don't manage expectations, make your product clear, or properly map out your rollout plan, you and your app will miss the mark entirely.

Like mobile devices, apps have a shelf life. It's important that your app makes the most of its time in the app store and on a user's phone. As we become even more dependent on our phones, and by extension apps, the competition will become fiercer and harder to beat. Failure is a part of life and as mentioned above, an app's failure is never just about one thing.

While this list of app failures isn't exhaustive, it demonstrates how mismanagement, lack of research, poor delivery, and timing and marketing mistakes can cost you. However, by mastering key processes such as app onboarding and useful notifications like in-app messaging, your app will have a bright future indeed.

Mobile app metrics are crucial for any application. They tell you how well your application performs from a technical standpoint, how engaged your users are, and how much money active users bring to your business. Apart from that, you should look into App Store optimization and Google Play Store efforts.

The most important KPI for application performance includes load time, crash reports, and device information, including screen resolution and operating systems. These metrics allow you to control your app's technical performance and improve the testing process. Identifying the devices your app runs on will give you a better understanding of your target audience and their sessions.

User engagement metrics influence revenue. They're by far the most important mobile app KPIs. Learn how long people spend on your app and how many screens they visit before churning or completing a target action (like making a purchase).

To keep track of these application KPIs, you'll need to **integrate analytics into your mobile app**. Mobile analytics tools come in all shapes and sizes. Do the research to select the right tool for your needs. A comprehensive analytics tool will include demographics, crash reports, heatmaps, and session recordings to give you a comprehensive understanding of how users interact with your app.

What Is Memory Optimization?

Memory optimization is a range of techniques related to improving computer memory, such as identifying memory leaks and corruption, to optimize memory usage and increase performance and application usability.

With memory optimization, memory resources are made more performant by resolving leaks and other difficult issues such as memory block overwrites and improper memory-API use.

What Is Memory Debugging?

While on the surface they may seem like synonyms, memory debugging is one of the techniques used towards the goal of memory optimization.

Memory debugging is a process of finding defects or issues in your code.

Memory debugging can help to identify why an application is crashing, or why it is producing data that is not correct. It can also be looked at as a form of dynamic analysis. It is a way of learning how an application is running so you can have a good, thorough understanding of it. You may need this information as you're refactoring code to extend it or build in a new functionality.

Why Is Memory Optimization Important?

In a word — performance.

If you use too much memory, or build up too much memory with a program running over time, performance is going to degrade. Using too much memory without releasing it to the heap (the virtual memory available as a resource to all programs) can result in running out of memory on the machine, requiring a reboot and potentially leading to many issues.

Ideally, in a well-written program, you use memory and then return it to the heap. That way there is always something on the heap to take from. Each program is allocated a piece of the heap. If a particular program (or programs) keep sucking up a disproportionate amount of the heap, program performance suffers until an eventual crash.

A prime example of a memory leak is a software that we all have on our smartphones and personal computers — web browsers. Web browsers utilize an enormous amount of memory, which grows with every new page, tab, or window opened, causing the device to run much slower while those functions remain open.

What Causes Memory Leaks?

When an application is running, it operates with allocated memory that is consumed from the heap. After that memory is used, that variable in the program needs to be deleted or released.

If you don't delete it, that variable is not returned to the heap and any reference to it is lost. From there, you are never able to access the memory again. This is what is referred to as a memory leak, and as these leaks pile up it is referred to as "memory bloat."

How to Avoid Memory Leaks

One way to minimize performance issues due to memory leaks to is to prevent memory leaks from occurring in the first place. While you can never be 100% resistant to memory leaks, there are best practices that can be followed to reduce their likelihood.

In good programming practice, you allocate memory for variables and data. When you are done using that variable or data, you then deallocate it to release it. This puts the memory back on the heap so you can draw from it in another instance.

How TotalView Addresses Memory Optimization

There are a number of techniques that can be used to optimize memory usage. MemoryScape, a tool within TotalView, provides the data to improve usage and identify memory leaks in C, C++, and Fortran. With the data output from MemoryScape, you can perform process improvement to your code to reduce memory leaks and ensure that there are no memory errors.

You may not always know that you have a memory leak, but you could have a suspicion if your programs begin running sluggish. Or, there are times when there may not be any apparent performance issues at all, but smaller, less noticeable issues can build up over time and manifest into larger issues. MemoryScape detects these issues to prevent their growth into systemic problems.

Within MemoryScape, the user can customize their memory debugging options to alert the user when any of a number of memory event actions occur. With each memory event, a follow-up action can be set such as halting execution, generating a core file, or saving memory data to a file.

Leak detection within MemoryScape has multiple forms. The below screenshot shows a sample leak detection report:

For visualization, leak detection can also be presented in a graphical report:

For a quick demo on how MemoryScape works, view this five-minute demonstration to see how it can analyze memory usage and detect memory errors.

Memory Management in Operating System

The term memory can be defined as a collection of data in a specific format. It is used to store instructions and process data. The memory comprises a large array or group of words or bytes, each with its own location. The primary purpose of a computer system is to execute programs. These programs, along with the information they access, should be in the main memory during execution. The CPU fetches instructions from memory according to the value of the program counter.

To achieve a degree of multiprogramming and proper utilization of memory, memory management is important. Many memory management methods exist, reflecting various approaches, and the effectiveness of each algorithm depends on the situation.

Here, we will cover the following memory management topics:

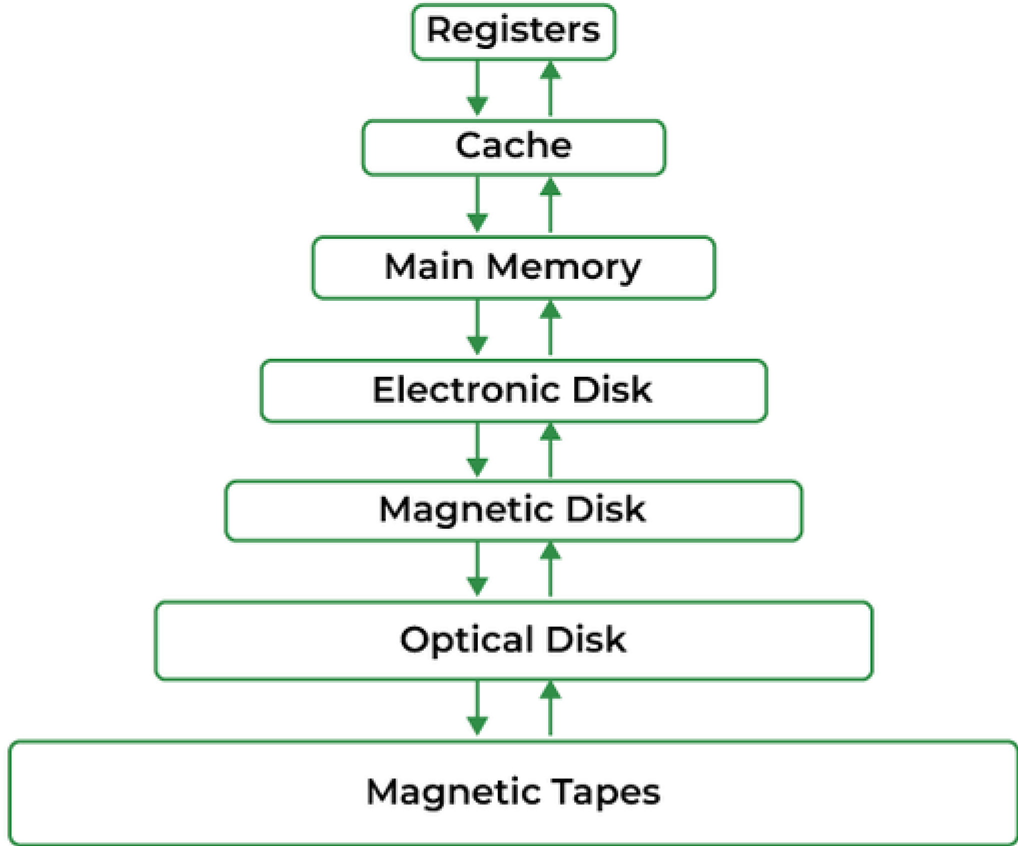
- *What is Main Memory?*
- *What is Memory Management?*
- *Why Memory Management is Required?*
- *Logical Address Space and Physical Address Space*
- *Static and Dynamic Loading*

- *Static and Dynamic Linking*
- *Swapping*
- *Contiguous Memory Allocation*
    - *Memory Allocation*
        - *First Fit*
        - *Best Fit*
        - *Worst Fit*
    - *Fragmentation*
        - *Internal Fragmentation*
        - *External Fragmentation*
    - *Paging*

Before we start Memory management, let us know what is main memory is.

What is Main Memory?

The main memory is central to the operation of a Modern Computer. Main Memory is a large array of words or bytes, ranging in size from hundreds of thousands to billions. Main memory is a repository of rapidly available information shared by the CPU and I/O devices. Main memory is the place where programs and information are kept when the processor is effectively utilizing them. Main memory is associated with the processor, so moving instructions and information into and out of the processor is extremely fast. Main memory is also known as RAM (Random Access Memory). This memory is volatile. RAM loses its data when a power interruption occurs.



Main Memory

What is Memory Management?

In a multiprogramming computer, the Operating System resides in a part of memory, and the rest is used by multiple processes. The task of subdividing the memory among different processes is called Memory Management. Memory management is a method in the operating system to manage operations between main memory and disk during process execution. The main aim of memory management is to achieve efficient utilization of memory.

Why Memory Management is Required?

- Allocate and de-allocate memory before and after process execution.
- To keep track of used memory space by processes.
- To minimize fragmentation issues.
- To proper utilization of main memory.
- To maintain data integrity while executing of process.

Now we are discussing the concept of Logical Address Space and Physical Address Space

Logical and Physical Address Space

- **Logical Address Space:** An address generated by the CPU is known as a "Logical Address". It is also known as a Virtual address. Logical address space can be defined as the size of the process. A logical address can be changed.

- **Physical Address Space:** An address seen by the memory unit (i.e the one loaded into the memory address register of the memory) is commonly known as a "Physical Address". A Physical address is also known as a Real address. The set of all physical addresses corresponding to these logical addresses is known as Physical address space. A physical address is computed by MMU. The run-time mapping from virtual to physical addresses is done by a hardware device Memory Management Unit(MMU). The physical address always remains constant.

Static and Dynamic Loading

Loading a process into the main memory is done by a loader. There are two different types of loading :

- **Static Loading:** Static Loading is basically loading the entire program into a fixed address. It requires more memory space.
- **Dynamic Loading:** The entire program and all data of a process must be in physical memory for the process to execute. So, the size of a process is limited to the size of physical memory. To gain proper memory utilization, dynamic loading is used. In dynamic loading, a routine is not loaded until it is called. All routines are residing on disk in a relocatable load format. One of the advantages of dynamic loading is that the unused routine is never loaded. This loading is useful when a large amount of code is needed to handle it efficiently.

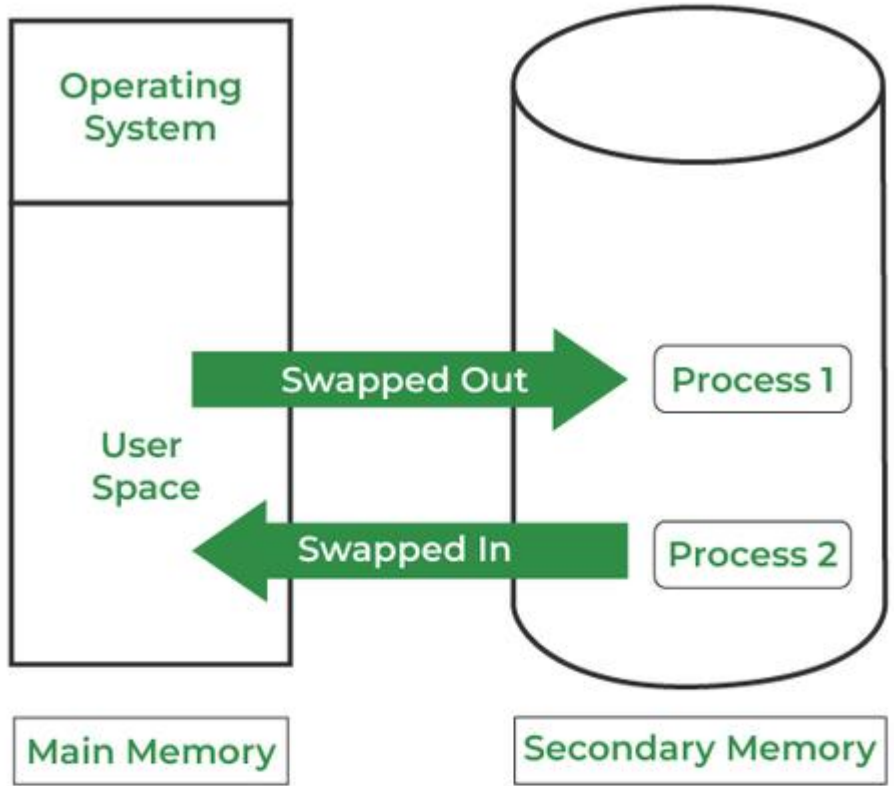
Static and Dynamic Linking

To perform a linking task a linker is used. A linker is a program that takes one or more object files generated by a compiler and combines them into a single executable file.

- **Static Linking:** In static linking, the linker combines all necessary program modules into a single executable program. So there is no runtime dependency. Some operating systems support only static linking, in which system language libraries are treated like any other object module.
- **Dynamic Linking:** The basic concept of dynamic linking is similar to dynamic loading. In dynamic linking, "Stub" is included for each appropriate library routine reference. A stub is a small piece of code. When the stub is executed, it checks whether the needed routine is already in memory or not. If not available then the program loads the routine into memory.

Swapping

When a process is executed it must have resided in memory. Swapping is a process of swapping a process temporarily into a secondary memory from the main memory, which is fast compared to secondary memory. A swapping allows more processes to be run and can be fit into memory at one time. The main part of swapping is transferred time and the total time is directly proportional to the amount of memory swapped. Swapping is also known as roll-out, or roll because if a higher priority process arrives and wants service, the memory manager can swap out the lower priority process and then load and execute the higher priority process. After finishing higher priority work, the lower priority process swapped back in memory and continued to the execution process.



swapping in memory management

Memory Management with Monoprogramming (Without Swapping)

This is the simplest memory management approach the memory is divided into two sections:

- One part of the operating system
- The second part of the user program

Fence Register	
operating system	user program

- In this approach, the operating system keeps track of the first and last location available for the allocation of the user program
- The operating system is loaded either at the bottom or at top
- Interrupt vectors are often loaded in low memory therefore, it makes sense to load the operating system in low memory
- Sharing of data and code does not make much sense in a single process environment
- The Operating system can be protected from user programs with the help of a fence register.

Advantages of Memory Management

- It is a simple management approach

Disadvantages of Memory Management

- It does not support multiprogramming
- Memory is wasted

Multiprogramming with Fixed Partitions (Without Swapping)

- A memory partition scheme with a fixed number of partitions was introduced to support multiprogramming. this scheme is based on contiguous allocation
- Each partition is a block of contiguous memory
- Memory is partitioned into a fixed number of partitions.
- Each partition is of fixed size

Example: As shown in fig. memory is partitioned into 5 regions the region is reserved for updating the system the remaining four partitions are for the user program.

Fixed Size Partitioning

Operating System
p1
p2
p3
p4

Partition Table

Once partitions are defined operating system keeps track of the status of memory partitions it is done through a data structure called a partition table.

Sample Partition Table

Starting Address of Partition	**Size of Partition**	**Status**
0k	200k	allocated
200k	100k	free

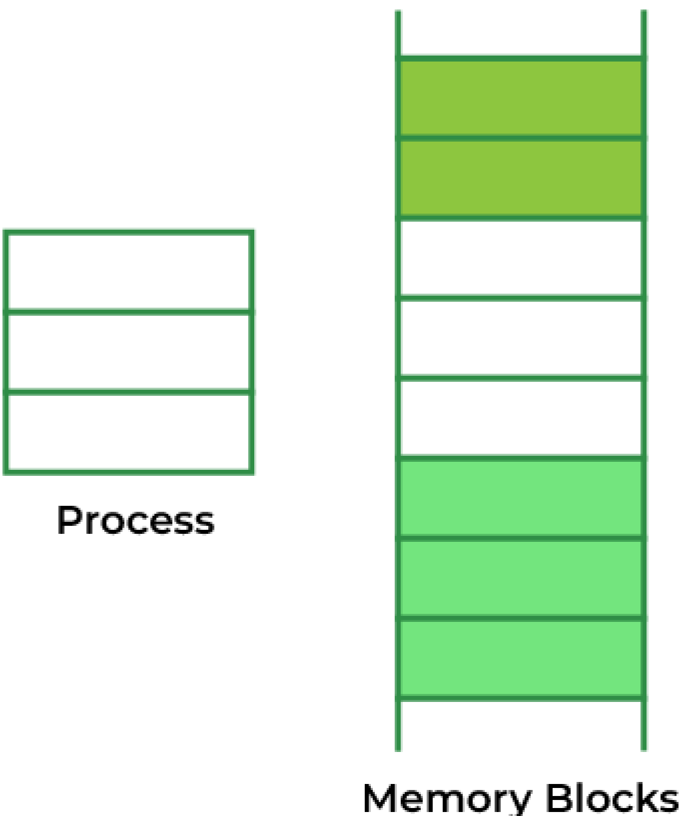
Starting Address of Partition	Size of Partition	Status
300k	150k	free
450k	250k	allocated

Logical vs Physical Address

An address generated by the CPU is commonly referred to as a logical address. the address seen by the memory unit is known as the physical address. The logical address can be mapped to a physical address by hardware with the help of a base register this is known as dynamic relocation of memory references.

Contiguous Memory Allocation

The main memory should accommodate both the operating system and the different client processes. Therefore, the allocation of memory becomes an important task in the operating system. The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. We normally need several user processes to reside in memory simultaneously. Therefore, we need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In adjacent memory allotment, each process is contained in a single contiguous segment of memory.



Contiguous Memory Allocation

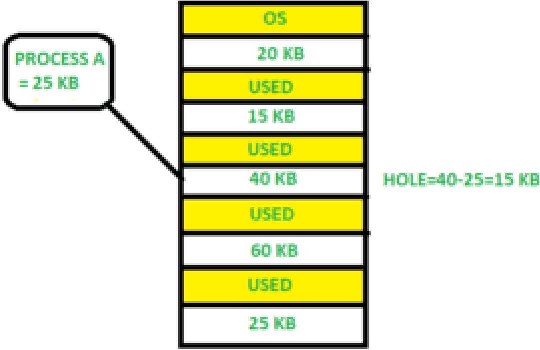
Memory Allocation

To gain proper memory utilization, memory allocation must be allocated efficient manner. One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions and each partition contains exactly one process. Thus, the degree of multiprogramming is obtained by the number of partitions.

- **Multiple partition allocation:** In this method, a process is selected from the input queue and loaded into the free partition. When the process terminates, the partition becomes available for other processes.
- **Fixed partition allocation:** In this method, the operating system maintains a table that indicates which parts of memory are available and which are occupied by processes. Initially, all memory is available for user processes and is considered one large block of available memory. This available memory is known as a "Hole". When the process arrives and needs memory, we search for a hole that is large enough to store this process. If the requirement is fulfilled then we allocate memory to process, otherwise keeping the rest available to satisfy future requests. While allocating a memory sometimes dynamic storage allocation problems occur, which concerns how to satisfy a request of size n from a list of free holes. There are some solutions to this problem:

First Fit

In the First Fit, the first available free hole fulfil the requirement of the process allocated.

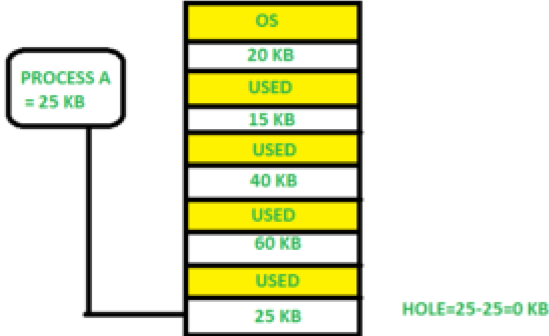


First Fit

Here, in this diagram, a 40 KB memory block is the first available free hole that can store process A (size of 25 KB), because the first two blocks did not have sufficient memory space.

Best Fit

In the Best Fit, allocate the smallest hole that is big enough to process requirements. For this, we search the entire list, unless the list is ordered by size.

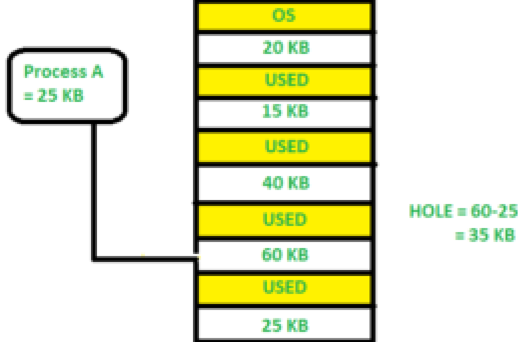


Best Fit

Here in this example, first, we traverse the complete list and find the last hole 25KB is the best suitable hole for Process A(size 25KB). In this method, memory utilization is maximum as compared to other memory allocation techniques.

Worst Fit

In the Worst Fit, allocate the largest available hole to process. This method produces the largest leftover hole.



Worst Fit

Here in this example, Process A (Size 25 KB) is allocated to the largest available memory block which is 60KB. Inefficient memory utilization is a major issue in the worst fit.

Fragmentation

Fragmentation is defined as when the process is loaded and removed after execution from memory, it creates a small free hole. These holes can not be assigned to new processes because holes are not combined or do not fulfill the memory requirement of the process. To achieve a degree of multiprogramming, we must reduce the waste of memory or fragmentation problems. In the operating systems two types of fragmentation:

1. **Internal fragmentation:** Internal fragmentation occurs when memory blocks are allocated to the process more than their requested size. Due to this some unused space is left over and creating an internal fragmentation problem.**Example:** Suppose there is a fixed partitioning used for memory allocation and the different sizes of blocks 3MB, 6MB, and 7MB space in memory. Now a new process p4 of size 2MB comes and demands a block of memory. It gets a memory block of 3MB but 1MB block of memory is a waste, and it can not be allocated to other processes too. This is called internal fragmentation.
2. **External fragmentation:** In External Fragmentation, we have a free memory block, but we can not assign it to a process because blocks are not contiguous. **Example:** Suppose (consider the above example) three processes p1, p2, and p3 come with sizes 2MB, 4MB, and 7MB respectively. Now they get memory blocks of size 3MB, 6MB, and 7MB allocated respectively. After allocating the process p1 process and the p2 process left 1MB and 2MB. Suppose a new process p4 comes and demands a 3MB block of memory, which is available, but we can not assign it because free memory space is not contiguous. This is called external fragmentation.

Both the first-fit and best-fit systems for memory allocation are affected by external fragmentation. To overcome the external fragmentation problem Compaction is used. In the compaction technique, all free memory space combines and makes one large block. So, this space can be used by other processes effectively.

Another possible solution to the external fragmentation is to allow the logical address space of the processes to be noncontiguous, thus permitting a process to be allocated physical memory wherever the latter is available.

Paging

Paging is a memory management scheme that eliminates the need for a contiguous allocation of physical memory. This scheme permits the physical address space of a process to be non-contiguous.

- **Logical Address or Virtual Address (represented in bits):** An address generated by the CPU.
- **Logical Address Space or Virtual Address Space (represented in words or bytes):** The set of all logical addresses generated by a program.
- **Physical Address (represented in bits):** An address actually available on a memory unit.
- **Physical Address Space (represented in words or bytes):** The set of all physical addresses corresponding to the logical addresses.

Example:

- If Logical Address = 31 bits, then Logical Address Space = 231 words = 2 G words (1 G = 230)
- If Logical Address Space = 128 M words = 27 * 220 words, then Logical Address = log2 227 = 27 bits
- If Physical Address = 22 bits, then Physical Address Space = 222 words = 4 M words (1 M = 220)
- If Physical Address Space = 16 M words = 24 * 220 words, then Physical Address = log2 224 = 24 bits

The mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device and this mapping is known as the paging technique.

- The Physical Address Space is conceptually divided into several fixed-size blocks, called **frames**.
- The Logical Address Space is also split into fixed-size blocks, called **pages**.
- Page Size = Frame Size

Let us consider an example:

- Physical Address = 12 bits, then Physical Address Space = 4 K words
- Logical Address = 13 bits, then Logical Address Space = 8 K words
- Page size = frame size = 1 K words (assumption)

Paging

The address generated by the CPU is divided into:

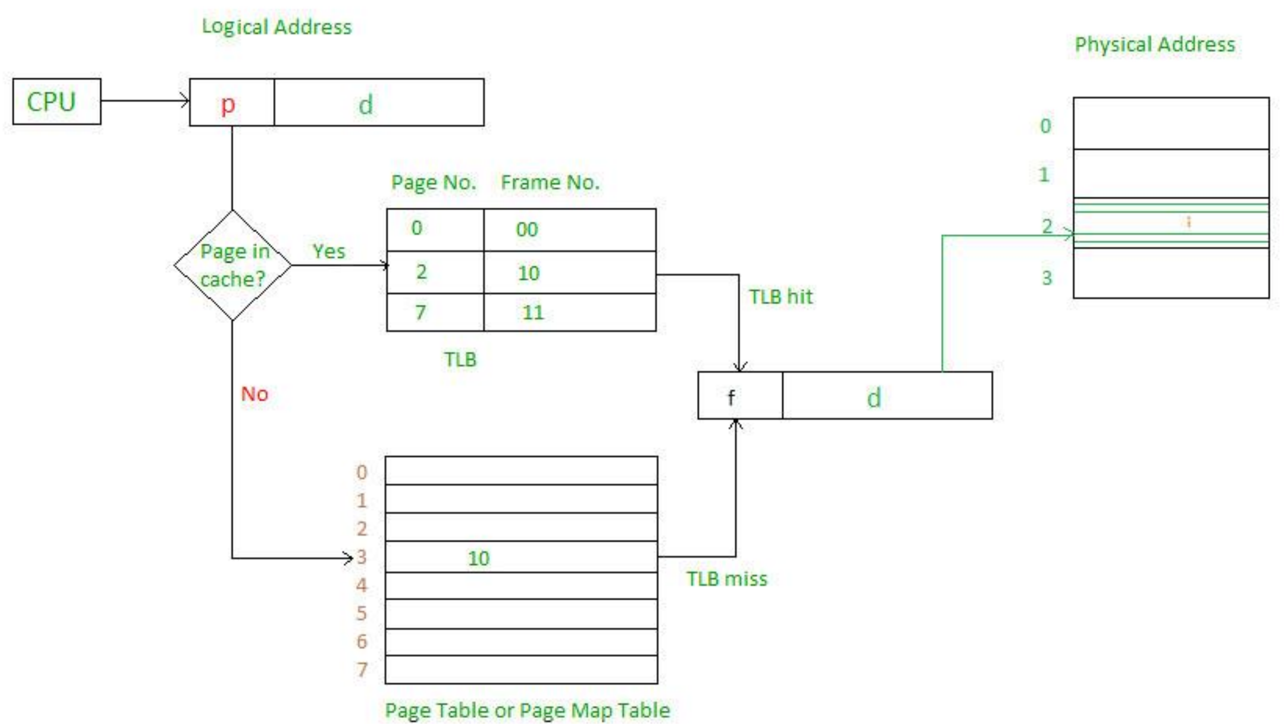
- **Page Number(p):** Number of bits required to represent the pages in Logical Address Space or Page number
- **Page Offset(d):** Number of bits required to represent a particular word in a page or page size of Logical Address Space or word number of a page or page offset.

Physical Address is divided into:

- **Frame Number(f):** Number of bits required to represent the frame of Physical Address Space or Frame number frame
- **Frame Offset(d):** Number of bits required to represent a particular word in a frame or frame size of Physical Address Space or word number of a frame or frame offset.

The hardware implementation of the page table can be done by using dedicated registers. But the usage of the register for the page table is satisfactory only if the page table is small. If the page table contains a large number of entries then we can use TLB(translation Look-aside buffer), a special, small, fast look-up hardware cache.

- The TLB is an associative, high-speed memory.
- Each entry in TLB consists of two parts: a tag and a value.
- When this memory is used, then an item is compared with all tags simultaneously. If the item is found, then the corresponding value is returned.



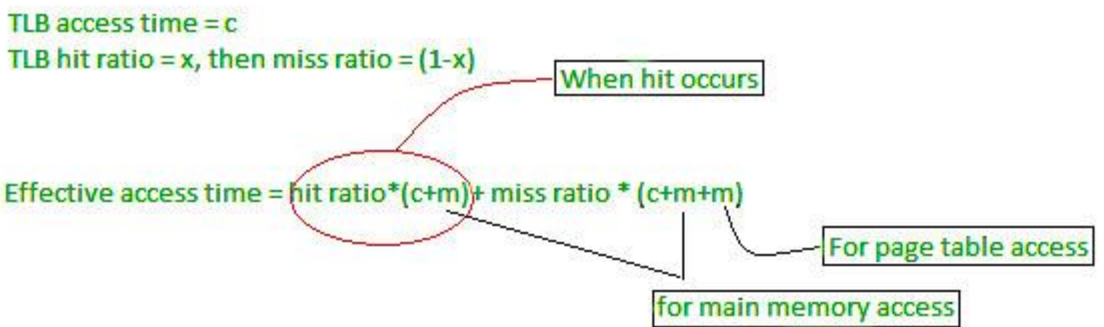
Page Map Table

Main memory access time = m

If page table are kept in main memory,

Effective access time = m(for page table)

+ m(for particular page in page table)



TLB Hit and Miss

For more details, must-read **Paging in Operating System**

Frequently Asked Questions

Q.1: What is a memory leak, and how does it affect system performance?

Answers:

A memory leak occurs when a program fails to release memory that it no longer needs, resulting in wasted memory resources. Over time, if memory leaks accumulate, the system's available memory diminishes, leading to reduced performance and possibly system crashes.

Q.2: Can memory fragmentation be prevented in an operating system?

Answers:

While it is challenging to completely eliminate memory fragmentation, certain techniques can help minimize its impact. One approach is to use memory allocation algorithms that focus on reducing external fragmentation, such as buddy systems or slab allocation. Additionally, techniques like compaction can be employed to reduce internal fragmentation.

Memory Allocation

Memory allocation is the process of setting aside sections of memory in a program to be used to store variables, and instances of structures and classes. There are two basic types of memory allocation:

Static Memory Allocation

Declaration of variables, structures and classes at the beginning of a class or function

```
simple  SimArray[50];
simple  sim1;
int     x;
double  d;
char    ch;
```

When you declare a variable or an instance of a structure or class. The memory for that object is allocated by the operating system. The name you declare for the object can then be used to access that block of memory.

Dynamic Memory Allocation

Memory allocated while the program is running using calls to *new* (C++) or *malloc()* (C)

```
simple    *sptr;
int       *iptr;
double    *dptr;
char      *cptr;
MyList    *list;
```

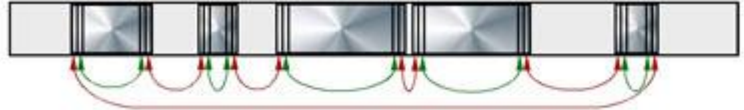
```
sptr = new simple();     or   sptr = (struct simple *)malloc(sizeof(struct simple));
iptr = new int;          or   iptr = (int *)malloc(sizeof(int));
dptr = new double;       or   dptr = (double *)malloc(sizeof(double));
cptr = new char;         or   cptr = (char *)malloc(sizeof(char));
list = new MyList();          malloc is not used when creating classes
```

When you use dynamic memory allocation you have the operating system designate a block of memory of the appropriate size while the program is running. This is done either with the **new** operator or with a call to the **malloc** function. The block of memory is allocated and a pointer to the block is returned. This is then stored in a pointer to the appropriate data type.

The Heap

The **Heap** is that portion of computer memory, allocated to a running application, where memory can be allocated for variables, class instances, etc. From a program's heap the OS allocates memory for dynamic use. Given a pointer to any one of the allocated blocks the OS can search in either direction to locate a block big enough to fill a dynamic memory allocation request.

Blocks of memory allocated on the heap are actually a special type of data structure consisting of (1) A pointer to the end of the previous block, (2) a pointer to the end of this block, (3) the allocated block of memory which can vary in size depending on its use, (4) a pointer to the beginning of this block, and (5) a pointer to the next block.



The Heap Layout.
Blank spaces between allocated blocks are where previously used blocks have been deallocated.

Depending on the type of operating system there are two possible algorithms that can be implemented in order to locate a block of memory in the heap to allocate:

- **First-Fit** - The needed amount of memory is allocated in the first block located in the heap that is big enough. This is faster but can result in greater fragmentation* of the heap.
- **Best-Fit** - All of the heap is searched and the needed amount of memory is allocated in the block where there is the least amount of memory left over. Slower but can result in less fragmentation*.

***Heap Fragmentation** - the condition that results when there is a lot of memory allocation and deallocation taking place in a program while it is running. With either memory allocation algorithm method a point may be reached where an attempt to allocate additional memory may fail. There may be enough total memory available in the heap, but it is just not all in one place. Some operating systems (Mac OS most noteably) have a special algorithm that reduces heap fragmentation through a process known as **compacting the heap**. That is allocated blocks in the heap are moved together and pointers to these blocks are changed to match the new locations. This can be done automatically without the programmer having to provide any special coding and completely invisible to the user.

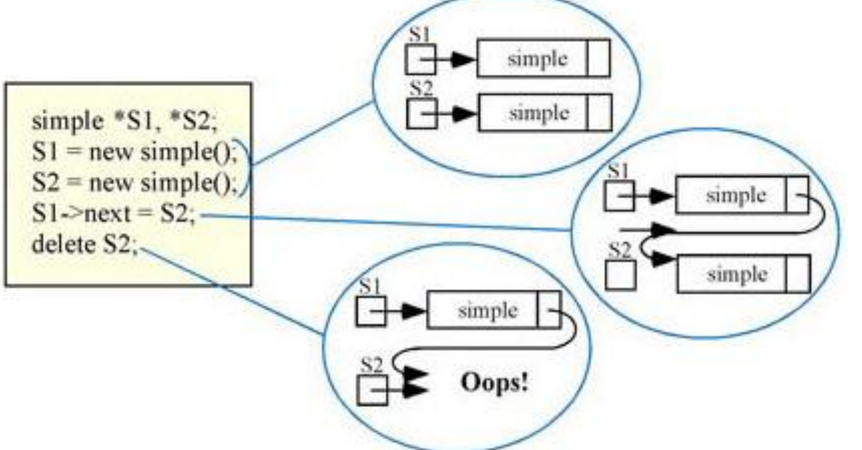
Foot Shot Warnings

These are some things to be careful of when allocating memory for used in linked data structures.

It is a standing joke among old time C programmers that you must be careful when writing code because C will happily let you shoot yourself in the foot. In keeping with this idea here are some things to be careful of when allocating memory for use in linked data structures.

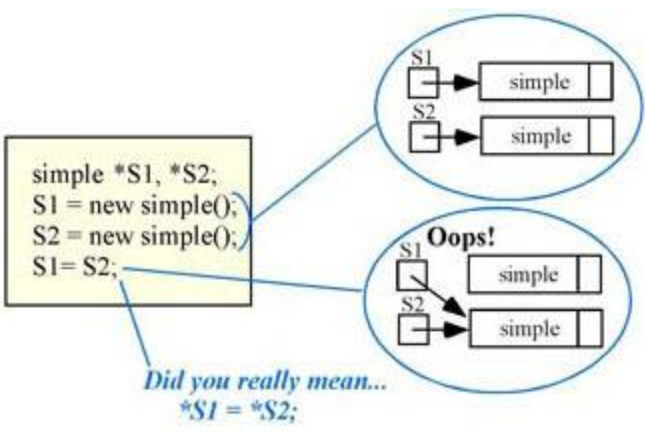
Creating dangling pointers.

In the example two instances of struct simple are created an linked by setting S1->next = S2. Unfortunately when S2 is deleted you are left with S1->next still pointing at the now unallocated memory location.



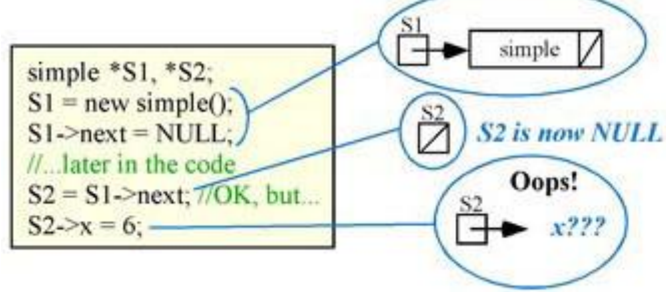
Creating garbage with no way to do garbage collection.

Garbage collection means deallocating dynamic memory when it is no longer needed using the **delete** operator. Failure to do so, for example, by moving a pointer to a block of memory without first deallocating it, leaves allocated memory with no reference to it to delete it, i.e. garbage.



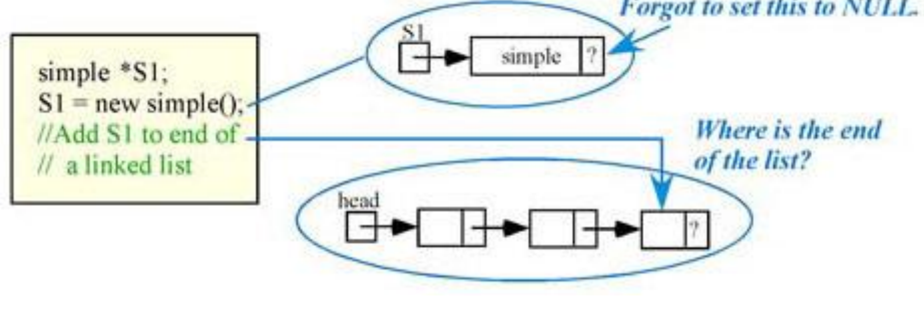
Dereferencing the NULL pointer.

In spite of the fact that this seems to be a no-brainer it is very easy to do. You must carefully study an algorithm to determine if it is possible for it to set a pointer to NULL and include code to handle this possibility.



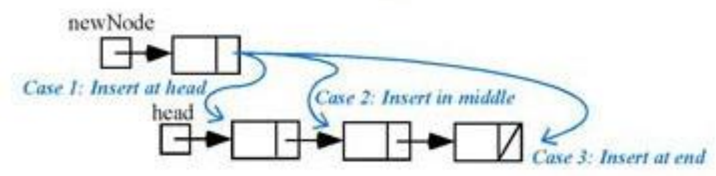
Forgetting to mark the end of a list.

Many linked data types use NULL in pointers to indicate that is the end of the line of linked structures. If you fail to set this pointer to NULL is it not done automatically. Any non-NULL value, even if it is garbage, may be misinterpreted as a valid memory address.



Failure to handle boundary cases.

When inserting and deleting items from a linked structure the algorithm must take into account all possible cases. For example, in the insert algorithm for an ordered linked list you may be inserting a new item into (1) an empty list, (2) at the head of the list, (3) somewhere in the middle of the list, or (4) at the end of the list. Failure to handle all of the possible cases can result in poionter problems.



What Is Resource Optimization?

Resource optimization is the process of allocating available resources in the most efficient manner to achieve desired goals. These resources can be in any form, including time, capital, electronics, and manpower. Both under-utilization and over-utilization of these resources can lead to higher costs, burnouts, and turnaround time.

The **primary goal of resource optimization** is to maximize the productivity of resources along with reducing costs of labor and other expenses. It can also help deliver quality products on pre-determined timelines.

Key Benefits Of Resource Optimization In Project Management

Optimizing resources brings more than one advantage to organizations. Some of the most prominent of these are

1. Demand Forecasting

Demand forecasting is the process of predicting future customer demands for a defined period based on past data analysis. Making these estimates is an important part of resource optimization and risk management. Demand forecasting is helpful for various business processes, including decision-making and business strategy formulation.

2. Resource Scheduling

Resource scheduling is the process of assigning the right resources to the right tasks. This is done based on the employee's skills, availability, and capacity. Resource scheduling helps ensure employees have adequate time to prepare for and complete the tasks assigned to them. It also helps save capital and properly utilize each available resource without overburdening anyone with tasks.

3. Improved Transparency And Trust

Things can become chaotic without the right resource optimization techniques in place. Project Managers might assign tasks to the same resources in a pool, and others might be left out.

This creates confusion and leads to poor project management. With appropriate optimization, all ongoing and upcoming tasks can be viewed and planned easily, leading to better transparency and trust among teams.

4. Reduced Overheads And Increased Revenue

Inefficient use of resources eventually lead to higher expenses in every aspect of running a business. Instead of utilizing the available resources, there might arise the need to hire more employees. However, efficient resource utilization can help reduce these costs along with producing desired results.

Resource Optimization Techniques To Achieve Cost-Efficiency

Resource optimization in IT projects are implemented to make necessary changes in project deliverables and manage allocated resources and resource accessibility. These techniques also bring about cost-efficiency, which might or might not be the ultimate goal, but is a very important part of the business strategy.

Here are two of the most used resource optimization techniques:

Resource leveling

Resource leveling is a process where the number of resources is fixed, but the commencement and completion dates of a project are altered as required to produce quality output. This resource optimization technique helps project managers to allocate resources in a balanced manner while considering the demand for a particular resource and its availability. It also helps reduce employee burnout and allocate the tasks that employees are qualified for and experience.

Resource smoothing

In the **resource smoothing technique**, the dates of project initiation and completion are fixed, but the project managers are free to allocate all the necessary resources. It is also known as **time-constrained scheduling** and helps ensure project requirements stay as per the pre-defined plan. Resource allocation is done based on the availability and priority of the tasks. If one task initiation is critical, the project manager can move resources from other less critical tasks and meet the timeline.

Difference Between Resource Smoothing And Resource Leveling

Both resource smoothing and leveling are widely used project management techniques aimed at achieving project goals while maintaining constraints. But they are fundamentally different in their approach. Some of the major differences include (and are not limited to):

Parameter	Resource Smoothing	Resource Leveling
Project Timeline	Fixed	Movable
Availability of Resources	Unlimited	Fixed or Limited
Order of Occurrence	Performed after resource leveling	Usually scheduled first
Change in Critical Path	Not allowed	Allowed
Required When	Resources are unevenly allocated	Resources are over-allocated

How To Effectively Implement Resource Optimization Techniques

To overcome all the challenges and ensure effective implementation of resource optimization techniques, it is first necessary to realize that there is no textbook solution that can work the same for all organizations. Optimum resource allocation is a requirement-based process that is also ongoing in nature. Continuous research can lead to ensuring improvement in the implemented resource optimization techniques.

At **GeekyAnts- App Design & Development Studio**, we undertake the planning of projects both as per resource and project requirements. We work by implementing techniques to ensure no resource is over-booked or under-booked. We also ensure all resources are allocated appropriately by implementing OKRs, upskilling programs for employees, analyzing project requirements, and determining the best suitable resources based on analyzing their interests and expertise.

This approach of combining multiple factors to analyze requirements allows us to ensure higher employee satisfaction and retention, enhance market presence, build a work culture that helps gain the trust of existing and potential clients and employees, and eventually makes running the organization easier. It is an agile way of working that requires constant monitoring of resources

Debugging and Profiling

A golden rule in programming is that code does not do what you expect it to do, but what you tell it to do. Bridging that gap can sometimes be a quite difficult feat. In this lecture we are going to cover useful techniques for dealing with buggy and resource hungry code: debugging and profiling.

Debugging

Printf debugging and Logging

"The most effective debugging tool is still careful thought, coupled with judiciously placed print statements" — Brian Kernighan, *Unix for Beginners*.

A first approach to debug a program is to add print statements around where you have detected the problem, and keep iterating until you have extracted enough information to understand what is responsible for the issue.

A second approach is to use logging in your program, instead of ad hoc print statements. Logging is better than regular print statements for several reasons:

- You can log to files, sockets or even remote servers instead of standard output.
- Logging supports severity levels (such as INFO, DEBUG, WARN, ERROR, &c), that allow you to filter the output accordingly.
- For new issues, there's a fair chance that your logs will contain enough information to detect what is going wrong.

Here is an example code that logs messages:

$ python logger.py

 # Raw output as with just prints

$ python logger.py log

 # Log formatted output

$ python logger.py log ERROR

 # Print only ERROR levels and above

$ python logger.py color

 # Color formatted output

One of my favorite tips for making logs more readable is to color code them. By now you probably have realized that your terminal uses colors to make things more readable. But how does it do it? Programs like ls or grep are using ANSI escape codes, which are special sequences of characters to indicate your shell to change the color of the output. For example, executing echo -e "\e[38;2;255;0;0mThis is red\e[0m" prints the message This is red in red on your terminal, as long as it supports true color. If your terminal doesn't support this (e.g. macOS's Terminal.app), you can use the more universally supported escape codes for 16 color choices, for example echo -e "\e[31;1mThis is red\e[0m".

The following script shows how to print many RGB colors into your terminal (again, as long as it supports true color).

```
#!/usr/bin/env bash
for R in $(seq 0 20 255); do
  for G in $(seq 0 20 255); do
    for B in $(seq 0 20 255); do
      printf "\e[38;2;${R};${G};${B}m█\e[0m";
    done
  done
done
```

Third party logs

As you start building larger software systems you will most probably run into dependencies that run as separate programs. Web servers, databases or message brokers are common examples of this kind of dependencies. When interacting with these systems it is often necessary to read their logs, since client side error messages might not suffice.

Luckily, most programs write their own logs somewhere in your system. In UNIX systems, it is commonplace for programs to write their logs under /var/log. For instance, the NGINX webserver places its logs under /var/log/nginx. More recently, systems have started using a **system log**, which is increasingly where all of your log messages go. Most (but not all) Linux systems use systemd, a system daemon that controls many things in your system such as which services are enabled and running. systemd places the logs under /var/log/journal in a specialized format and you can use the journalctl command to display the messages. Similarly, on macOS there is still /var/log/system.log but an increasing number of tools use the system log, that can be displayed with log show. On most UNIX systems you can also use the dmesg command to access the kernel log.

For logging under the system logs you can use the logger shell program. Here's an example of using logger and how to check that the entry made it to the system logs. Moreover, most programming languages have bindings logging to the system log.

logger "Hello Logs"

 # On macOS

log show --last 1m | grep Hello

 # On Linux

journalctl --since "1m ago" | grep Hello

As we saw in the data wrangling lecture, logs can be quite verbose and they require some level of processing and filtering to get the information you want. If you find yourself heavily filtering through journalctl and log show you can consider using their flags, which can perform a first pass of filtering of their output. There are also some tools like lnav, that provide an improved presentation and navigation for log files.

Debuggers

When printf debugging is not enough you should use a debugger. Debuggers are programs that let you interact with the execution of a program, allowing the following:

- Halt execution of the program when it reaches a certain line.
- Step through the program one instruction at a time.
- Inspect values of variables after the program crashed.
- Conditionally halt the execution when a given condition is met.
- And many more advanced features

Many programming languages come with some form of debugger. In Python this is the Python Debugger pdb.

Here is a brief description of some of the commands pdb supports:

- **l**(ist) - Displays 11 lines around the current line or continue the previous listing.
- **s**(tep) - Execute the current line, stop at the first possible occasion.
- **n**(ext) - Continue execution until the next line in the current function is reached or it returns.
- **b**(reak) - Set a breakpoint (depending on the argument provided).
- **p**(rint) - Evaluate the expression in the current context and print its value. There's also **pp** to display using pprint instead.
- **r**(eturn) - Continue execution until the current function returns.
- **q**(uit) - Quit the debugger.

Let's go through an example of using pdb to fix the following buggy python code. (See the lecture video).

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n):
            if arr[j] > arr[j+1]:
                arr[j] = arr[j+1]
                arr[j+1] = arr[j]
    return arr


print(bubble_sort([4, 2, 1, 8, 7, 6]))
```

Note that since Python is an interpreted language we can use the pdb shell to execute commands and to execute instructions. ipdb is an improved pdb that uses the IPython REPL enabling tab completion, syntax highlighting, better tracebacks, and better introspection while retaining the same interface as the pdb module.

For more low level programming you will probably want to look into gdb (and its quality of life modification pwndbg) and lldb. They are optimized for C-like language debugging but will let you probe pretty much any process and get its current machine state: registers, stack, program counter, &c.

Specialized Tools

Even if what you are trying to debug is a black box binary there are tools that can help you with that. Whenever programs need to perform actions that only the kernel can, they use System Calls. There are commands that let you trace the syscalls your program makes. In Linux there's strace and macOS and BSD have dtrace. dtrace can be tricky to use because it uses its own D language, but there is a wrapper called dtruss that provides an interface more similar to strace (more details here).

Below are some examples of using strace or dtruss to show stat syscall traces for an execution of ls. For a deeper dive into strace, this article and this zine are good reads.

```
# On Linux
sudo strace -e lstat ls -l > /dev/null

# On macOS
sudo dtruss -t lstat64_extended ls -l > /dev/null
```

Under some circumstances, you may need to look at the network packets to figure out the issue in your program. Tools like tcpdump and Wireshark are network packet analyzers that let you read the contents of network packets and filter them based on different criteria.

For web development, the Chrome/Firefox developer tools are quite handy. They feature a large number of tools, including:

- Source code - Inspect the HTML/CSS/JS source code of any website.

- Live HTML, CSS, JS modification - Change the website content, styles and behavior to test (you can see for yourself that website screenshots are not valid proofs).
- Javascript shell - Execute commands in the JS REPL.
- Network - Analyze the requests timeline.
- Storage - Look into the Cookies and local application storage.

Static Analysis

For some issues you do not need to run any code. For example, just by carefully looking at a piece of code you could realize that your loop variable is shadowing an already existing variable or function name; or that a program reads a variable before defining it. Here is where static analysis tools come into play. Static analysis programs take source code as input and analyze it using coding rules to reason about its correctness.

In the following Python snippet there are several mistakes. First, our loop variable foo shadows the previous definition of the function foo. We also wrote baz instead of bar in the last line, so the program will crash after completing the sleep call (which will take one minute).

```
import time


def foo():
    return 42


for foo in range(5):
    print(foo)
bar = 1
bar *= 0.2
time.sleep(60)
print(baz)
```

Static analysis tools can identify these kinds of issues. When we run pyflakes on the code we get the errors related to both bugs. mypy is another tool that can detect type checking issues.
Here, mypy will warn us that bar is initially an int and is then casted to a float. Again, note that all these issues were detected without having to run the code.

```
$ pyflakes foobar.py

foobar.py:6: redefinition of unused 'foo' from line 3

foobar.py:11: undefined name 'baz'


$ mypy foobar.py
```

foobar.py:6: error: Incompatible types **in** assignment **(**expression has type "int", variable has type "Callable[[], Any]"**)**

foobar.py:9: error: Incompatible types **in** assignment **(**expression has type "float", variable has type "int"**)**

foobar.py:11: error: Name 'baz' is not defined

Found 3 errors **in** 1 file **(**checked 1 source file**)**

In the shell tools lecture we covered shellcheck, which is a similar tool for shell scripts.

Most editors and IDEs support displaying the output of these tools within the editor itself, highlighting the locations of warnings and errors. This is often called **code linting** and it can also be used to display other types of issues such as stylistic violations or insecure constructs.

In vim, the plugins ale or syntastic will let you do that. For Python, pylint and pep8 are examples of stylistic linters and bandit is a tool designed to find common security issues. For other languages people have compiled comprehensive lists of useful static analysis tools, such as Awesome Static Analysis (you may want to take a look at the *Writing* section) and for linters there is Awesome Linters.

A complementary tool to stylistic linting are code formatters such as black for Python, gofmt for Go, rustfmt for Rust or prettier for JavaScript, HTML and CSS. These tools autoformat your code so that it's consistent with common stylistic patterns for the given programming language. Although you might be unwilling to give stylistic control about your code, standardizing code format will help other people read your code and will make you better at reading other people's (stylistically standardized) code.

Profiling

Even if your code functionally behaves as you would expect, that might not be good enough if it takes all your CPU or memory in the process. Algorithms classes often teach big *O* notation but not how to find hot spots in your programs. Since premature optimization is the root of all evil, you should learn about profilers and monitoring tools. They will help you understand which parts of your program are taking most of the time and/or resources so you can focus on optimizing those parts.

Timing

Similarly to the debugging case, in many scenarios it can be enough to just print the time it took your code between two points. Here is an example in Python using the time module.

```
import time, random
n = random.randint(1, 10) * 100

# Get current time
start = time.time()

# Do some work
```

```
print("Sleeping for {} ms".format(n))
time.sleep(n/1000)


# Compute time between start and now
print(time.time() - start)


# Output
# Sleeping for 500 ms
# 0.5713930130004883
```

However, wall clock time can be misleading since your computer might be running other processes at the same time or waiting for events to happen. It is common for tools to make a distinction between *Real*, *User* and *Sys* time. In general, *User* + *Sys* tells you how much time your process actually spent in the CPU (more detailed explanation here).

- *Real* - Wall clock elapsed time from start to finish of the program, including the time taken by other processes and time taken while blocked (e.g. waiting for I/O or network)
- *User* - Amount of time spent in the CPU running user code
- *Sys* - Amount of time spent in the CPU running kernel code

For example, try running a command that performs an HTTP request and prefixing it with time. Under a slow connection you might get an output like the one below. Here it took over 2 seconds for the request to complete but the process only took 15ms of CPU user time and 12ms of kernel CPU time.

```
$ time curl https://missing.csail.mit.edu &> /dev/null
real    0m2.561s
user    0m0.015s
sys     0m0.012s
```

Profilers

CPU

Most of the time when people refer to *profilers* they actually mean *CPU profilers*, which are the most common. There are two main types of CPU profilers: *tracing* and *sampling* profilers. Tracing profilers keep a record of every function call your program makes whereas sampling profilers probe your program periodically (commonly every millisecond) and record the program's stack. They use these records to present aggregate statistics of what your program spent the most time doing. Here is a good intro article if you want more detail on this topic.

Most programming languages have some sort of command line profiler that you can use to analyze your code. They often integrate with full fledged IDEs but for this lecture we are going to focus on the command line tools themselves.

In Python we can use the cProfile module to profile time per function call. Here is a simple example that implements a rudimentary grep in Python:

```
#!/usr/bin/env python


import sys, re


def grep(pattern, file):
    with open(file, 'r') as f:
        print(file)
        for i, line in enumerate(f.readlines()):
            pattern = re.compile(pattern)
            match = pattern.search(line)
            if match is not None:
                print("{}: {}".format(i, line), end="")


if __name__ == '__main__':
    times = int(sys.argv[1])
    pattern = sys.argv[2]
    for i in range(times):
        for file in sys.argv[3:]:
            grep(pattern, file)
```

We can profile this code using the following command. Analyzing the output we can see that IO is taking most of the time and that compiling the regex takes a fair amount of time as well. Since the regex only needs to be compiled once, we can factor it out of the for.

```
$ python -m cProfile -s tottime grep.py 1000 '^(import|\s*def)[^,]*$' *.py


[omitted program output]


 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   8000    0.266    0.000    0.292    0.000 {built-in method io.open}
   8000    0.153    0.000    0.894    0.000 grep.py:5(grep)
  17000    0.101    0.000    0.101    0.000 {built-in method builtins.print}
```

```
    8000    0.100    0.000    0.129    0.000 {method 'readlines' of '_io._IOBase' objects}
   93000    0.097    0.000    0.111    0.000 re.py:286(_compile)
   93000    0.069    0.000    0.069    0.000 {method 'search' of '_sre.SRE_Pattern' objects}
   93000    0.030    0.000    0.141    0.000 re.py:231(compile)
   17000    0.019    0.000    0.029    0.000 codecs.py:318(decode)
       1    0.017    0.017    0.911    0.911 grep.py:3(<module>)

[omitted lines]
```

A caveat of Python's cProfile profiler (and many profilers for that matter) is that they display time per function call. That can become unintuitive really fast, especially if you are using third party libraries in your code since internal function calls are also accounted for. A more intuitive way of displaying profiling information is to include the time taken per line of code, which is what *line profilers* do.

For instance, the following piece of Python code performs a request to the class website and parses the response to get all URLs in the page:

```
#!/usr/bin/env python
import requests
from bs4 import BeautifulSoup

# This is a decorator that tells line_profiler
# that we want to analyze this function
@profile
def get_urls():
    response = requests.get('https://missing.csail.mit.edu')
    s = BeautifulSoup(response.content, 'lxml')
    urls = []
    for url in s.find_all('a'):
        urls.append(url['href'])

if __name__ == '__main__':
    get_urls()
```

If we used Python's cProfile profiler we'd get over 2500 lines of output, and even with sorting it'd be hard to understand where the time is being spent. A quick run with line_profiler shows the time taken per line:

```
$ kernprof -l -v a.py
Wrote profile results to urls.py.lprof
Timer unit: 1e-06 s

Total time: 0.636188 s
File: a.py
Function: get_urls at line 5

Line #  Hits         Time  Per Hit   % Time  Line Contents
==============================================================
 5                                           @profile
 6                                           def get_urls():
 7         1     613909.0 613909.0     96.5      response = requests.get('https://missing.csail.mit.edu')
 8         1      21559.0  21559.0      3.4      s = BeautifulSoup(response.content, 'lxml')
 9         1          2.0      2.0      0.0      urls = []
10        25        685.0     27.4      0.1      for url in s.find_all('a'):
11        24         33.0      1.4      0.0          urls.append(url['href'])
```

Memory

In languages like C or C++ memory leaks can cause your program to never release memory that it doesn't need anymore. To help in the process of memory debugging you can use tools like Valgrind that will help you identify memory leaks.

In garbage collected languages like Python it is still useful to use a memory profiler because as long as you have pointers to objects in memory they won't be garbage collected. Here's an example program and its associated output when running it with memory-profiler (note the decorator like in line-profiler).

```
@profile
def my_func():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a


if __name__ == '__main__':
```

```
    my_func()

$ python -m memory_profiler example.py

Line #    Mem usage  Increment   Line Contents

================================================

     3                           @profile

     4      5.97 MB    0.00 MB   def my_func():

     5     13.61 MB    7.64 MB       a = [1] * (10 ** 6)

     6    166.20 MB  152.59 MB       b = [2] * (2 * 10 ** 7)

     7     13.61 MB -152.59 MB       del b

     8     13.61 MB    0.00 MB       return a
```

Event Profiling

As it was the case for strace for debugging, you might want to ignore the specifics of the code that you are running and treat it like a black box when profiling. The perf command abstracts CPU differences away and does not report time or memory, but instead it reports system events related to your programs. For example, perf can easily report poor cache locality, high amounts of page faults or livelocks. Here is an overview of the command:

- perf list - List the events that can be traced with perf
- perf stat COMMAND ARG1 ARG2 - Gets counts of different events related to a process or command
- perf record COMMAND ARG1 ARG2 - Records the run of a command and saves the statistical data into a file called perf.data
- perf report - Formats and prints the data collected in perf.data

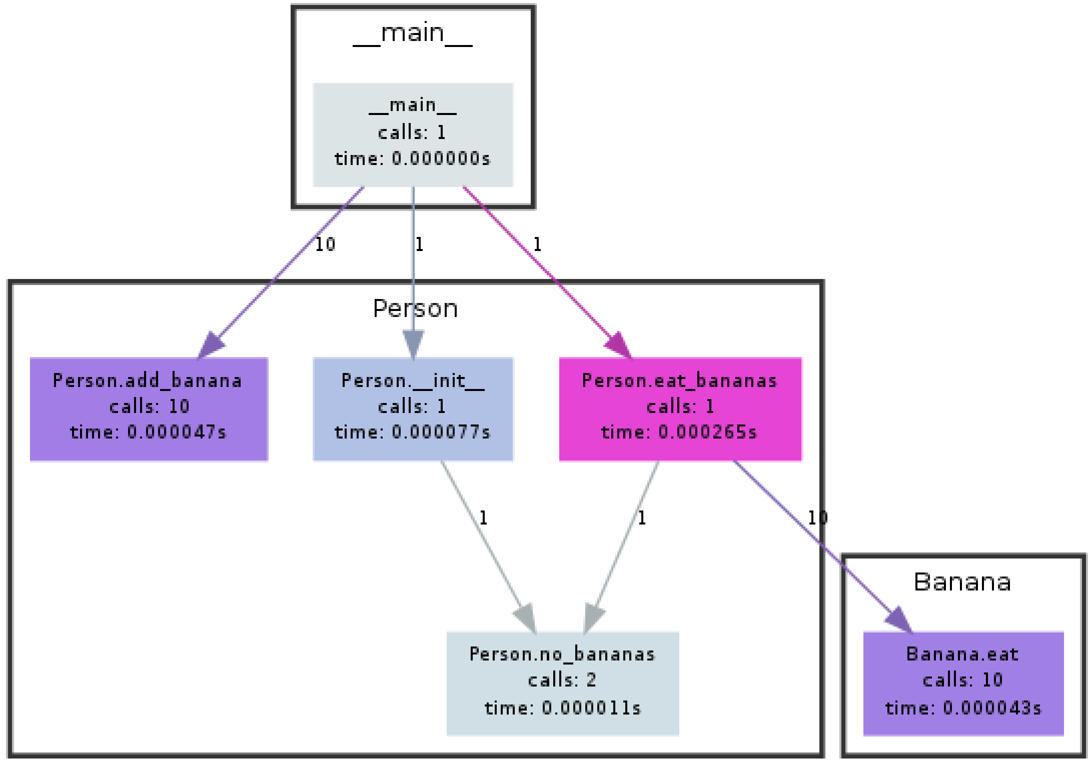
Visualization

Profiler output for real world programs will contain large amounts of information because of the inherent complexity of software projects. Humans are visual creatures and are quite terrible at reading large amounts of numbers and making sense of them. Thus there are many tools for displaying profiler's output in an easier to parse way.

One common way to display CPU profiling information for sampling profilers is to use a Flame Graph, which will display a hierarchy of function calls across the Y axis and time taken proportional to the X axis. They are also interactive, letting you zoom into specific parts of the program and get their stack traces (try clicking in the image below).

Call graphs or control flow graphs display the relationships between subroutines within a program by including functions as nodes and functions calls between them as directed edges. When coupled with profiling information such as the number of calls and time taken, call graphs can be quite useful

for interpreting the flow of a program. In Python you can use the [pycallgraph](#) library to generate them.



Resource Monitoring

Sometimes, the first step towards analyzing the performance of your program is to understand what its actual resource consumption is. Programs often run slowly when they are resource constrained, e.g. without enough memory or on a slow network connection. There are a myriad of command line tools for probing and displaying different system resources like CPU usage, memory usage, network, disk usage and so on.

- **General Monitoring** - Probably the most popular is htop, which is an improved version of top. htop presents various statistics for the currently running processes on the system. htop has a myriad of options and keybinds, some useful ones are: <F6> to sort processes, t to show tree hierarchy and h to toggle threads. See also glances for similar implementation with a great UI. For getting aggregate measures across all processes, dstat is another nifty tool that computes real-time resource metrics for lots of different subsystems like I/O, networking, CPU utilization, context switches, &c.
- **I/O operations** - iotop displays live I/O usage information and is handy to check if a process is doing heavy I/O disk operations
- **Disk Usage** - df displays metrics per partitions and du displays **d**isk **u**sage per file for the current directory. In these tools the -h flag tells the program to print with **h**uman readable format. A more interactive version of du is ncdu which lets you navigate folders and delete files and folders as you navigate.

- **Memory Usage** - free displays the total amount of free and used memory in the system. Memory is also displayed in tools like htop.
- **Open Files** - lsof lists file information about files opened by processes. It can be quite useful for checking which process has opened a specific file.
- **Network Connections and Config** - ss lets you monitor incoming and outgoing network packets statistics as well as interface statistics. A common use case of ss is figuring out what process is using a given port in a machine. For displaying routing, network devices and interfaces you can use ip. Note that netstat and ifconfig have been deprecated in favor of the former tools respectively.
- **Network Usage** - nethogs and iftop are good interactive CLI tools for monitoring network usage.

If you want to test these tools you can also artificially impose loads on the machine using the stress command.

Specialized tools

Sometimes, black box benchmarking is all you need to determine what software to use. Tools like hyperfine let you quickly benchmark command line programs. For instance, in the shell tools and scripting lecture we recommended fd over find. We can use hyperfine to compare them in tasks we run often. E.g. in the example below fd was 20x faster than find in my machine.

```
$ hyperfine --warmup 3 'fd -e jpg' 'find . -iname "*.jpg"'

Benchmark #1: fd -e jpg

  Time (mean ± σ):      51.4 ms ±   2.9 ms    [User: 121.0 ms, System: 160.5 ms]

  Range (min … max):    44.2 ms …  60.1 ms    56 runs


Benchmark #2: find . -iname "*.jpg"

  Time (mean ± σ):      1.126 s ±  0.101 s    [User: 141.1 ms, System: 956.1 ms]

  Range (min … max):    0.975 s …  1.287 s    10 runs


Summary

  'fd -e jpg' ran

   21.89 ± 2.33 times faster than 'find . -iname "*.jpg"'
```

As it was the case for debugging, browsers also come with a fantastic set of tools for profiling webpage loading, letting you figure out where time is being spent (loading, rendering, scripting, &c). More info for Firefox and Chrome.

Exercises

Debugging

1. Use journalctl on Linux or log show on macOS to get the super user accesses and commands in the last day. If there aren't any you can execute some harmless commands such as sudo ls and check again.
2. Do this hands on pdb tutorial to familiarize yourself with the commands. For a more in depth tutorial read this.
3. Install shellcheck and try checking the following script. What is wrong with the code? Fix it. Install a linter plugin in your editor so you can get your warnings automatically.
4. *#!/bin/sh*
5. *## Example: a typical script with several problems*
6. **for** f **in** $(ls *.m3u)
7. **do**
8. grep -qi hq.*mp3 $f \
9. **&&** echo -e 'Playlist $f contains a HQ file in mp3 format'
10. **done**
11. (Advanced) Read about reversible debugging and get a simple example working using rr or RevPDB.

Profiling

12. Here are some sorting algorithm implementations. Use cProfile and line_profiler to compare the runtime of insertion sort and quicksort. What is the bottleneck of each algorithm? Use then memory_profiler to check the memory consumption, why is insertion sort better? Check now the inplace version of quicksort. Challenge: Use perf to look at the cycle counts and cache hits and misses of each algorithm.
13. Here's some (arguably convoluted) Python code for computing Fibonacci numbers using a function for each number.
14. *#!/usr/bin/env python*
15. **def fib0**(): **return** 0
16.
17. **def fib1**(): **return** 1
18.
19. s = """def fib{}(): return fib{}() + fib{}()"""
20.
21. **if** __name__ == '__main__':
22.
23. **for** n **in** range(2, 10):

24. exec(s.format(n, n-1, n-2))

25. # from functools import lru_cache

26. # for n in range(10):

27. #   exec("fib{} = lru_cache(1)(fib{})".format(n, n))

28. print(eval("fib9()"))

Put the code into a file and make it executable. Install prerequisites: pycallgraph and graphviz. (If you can run dot, you already have GraphViz.) Run the code as is with pycallgraph graphviz -- ./fib.py and check the pycallgraph.png file. How many times is fib0 called?. We can do better than that by memoizing the functions. Uncomment the commented lines and regenerate the images. How many times are we calling each fibN function now?

29. A common issue is that a port you want to listen on is already taken by another process. Let's learn how to discover that process pid. First execute python -m http.server 4444 to start a minimal web server listening on port 4444. On a separate terminal run lsof | grep LISTEN to print all listening processes and ports. Find that process pid and terminate it by running kill <PID>.

30. Limiting a process's resources can be another handy tool in your toolbox. Try running stress -c 3 and visualize the CPU consumption with htop. Now, execute taskset --cpu-list 0,2 stress -c 3 and visualize it. Is stress taking three CPUs? Why not? Read man taskset. Challenge: achieve the same using cgroups. Try limiting the memory consumption of stress -m.

31. (Advanced) The command curl ipinfo.io performs a HTTP request and fetches information about your public IP. Open Wireshark and try to sniff the request and reply packets that curl sent and received. (Hint: Use the http filter to just watch HTTP packets).

Debugging, Profiling, and Validation

by Dave Levermore
AMSC 664 lecture, 31 January 2003

Once code is written you must begin three tasks that should continue so long as the code is in use:

- **debugging** --- getting the code to work as you intended;
- **profiling** --- assessing how the code carries out a given scientific task on a given platform and how its performance might be improved;
- **validation** --- assessing how accurately the code carries out a given scientific task.

These tasks should be carried out on each component (module) of your code in "isolation". You should be aware of and use all tools at your disposal. You should keep a log of both your profiling and validation efforts. There should be a section addressing each in your final project report.

Debugging has two stages:

- to get the code to compile and run,
- to see that it runs as intended.

The first stage must be completed before doing anything else. It must be revisited every time new coding is introduced. The second stage begins during the profiling and validation processes, but should continue throughout the useful life of the code. Of course, most compilers come with debugging tools that help you through the first stage. There are also some platform-dependent debugging tools as well as many general strategies that can be applied to the second stage. Examples will be given in subsequent lectures.

Profiling involves the insertion of diagnostics into your code to assess its performance on a given platform. You want to find out:

- What is the sensitivity to round-off error?
- In which subroutines and loops is the code spending its time?
- On parallel platforms, is the load balanced?
  Is there a synchronization bottleneck?
  Are there communication delays?

One can use such information to tune your code's performance. Many of the tools with which you make such assessments are platform dependent. For example, one can check round-off sensitivity by masking the two least significant bits of the calculation. In some cases this can be done at the level of the compiler while in others one has to insert some lines into your code. Other examples will be given in subsequent lectures.

Validation of both your model and your algorithms is critical to the development of any piece of scientific software. How one might do this is the focus of this lecture. We will mention several approaches that are geared toward traditional simulation codes, but most of them have analogs that can be applied to any scientific computation project.

- **Hand Calculation Checks.** These are among the most basic algorithm checks. Simply set up small scale calculations that you can check by "hand". For example, check a linear system solver on a two-by-two matrix. Similarly, check a two dimensional simulator on a two-by-two grid. For such small scale problems you can look at every number produced by your code and compare it with your hand calculation (usually carried out on a programmable calculator or some other computer). It does not matter that such under-resolved calculations are not physically relevant. Such checks are used primarily for debugging. Because they are time consuming, they should not be the first thing you try. However, if a possible bug is suspect at any stage of your validation process you should consider using one.
- **Basic Stability Studies.** These are also among the most basic tests of your algorithms. You must check that simple physically stable states are indeed stable when simulated by your code. For example, in a gas dynamics setting, does your codes maintian a constant solution with uniform density, velocity, and temperature? Of course, you should check that such a solution is in fact a stable solution of your model (or a reduction thereof) before carrying out its simulation. If your simulation is then unstable then the problem can be either a bug in the code (see above), the fact you have chosen an unstable algorithm, or both.
- **Symmetry Invariance Studies.** These simply compute two or more problems that are computationally identical up to a rotation, reflection, or translation to see if the simulations behave as they should.

For example, a one-dimensional problem with slab symmetry can be set up on a grid first from left to right and then from right to left. If these simulations were to be carried out with a perfect algorithm on a perfect platform they would produce the same numbers up to a reflection. However, in a realistic setting one should see differences that should be understood, especially if they are significantly larger than the expected round-off error. Such differences might be caused by a bug like a subtle index mistake in a loop or a boundary value specification. They might also arise due to an asymmetry in your algorithm. For example, if your code includes a Gaussian elimination subroutine it may always back-solve from right to left.

Another example in the same spirit is to compute a problem with periodic boundary conditions and its shift by half a period. Any funny looking behavior at a computational boundary that does not then shift to the middle of the problem is almost certainly an indication of a bug, most likely in your imposition of the periodic boundary condition.

- **Symmetry Preservation Studies.** These simulate solutions of the model (or a reduction thereof) that have more symmetries than can be preserved by the simulations. For example, how well does your code compute solutions with spherical symmetry? (There is always symmetry breaking in the simulation of such solutions when they are computed by a multi-dimensional code except in some trivial cases.) How much symmetry breaking one sees will depend the stability of the sphereically symmetric solution. Similarly, in a multi-dimensional code one can simulate a solution with slab symmetry on a grid aligned with the plane of symmetry and on a grid oblique to the plane of symmetry. In general one will see differences in the speed at which the waves propagate.
- **Comparisons with Special Solutions.** Many models have reductions for which analytic solutions can be found. You should use such solutions to validate your algorithms. These can include spatially homogeneous solutions, self-similar solutions, traveling-wave and other steady-state solutions. Almost as good as an analytical solution is a special solution that can be computed by numerically integrating an ordinary differential equation. Such a reduction can be found through symmetries.
- **Convergence Studies.** These can be applied on the level of a given subroutine to the level of a full calculation. For example, many efficient (linear or nonlinear) system solvers use iterative algorithms. Both the convergence rate and the stopping criterion should be validated on a few systems representive of the systems your code will face in a typical simulation. If a convergence rate is known theoretically for your algorithm, your code should converge at that rate (allowing for round-off). For a code that simulates the solution of a system of differential equations the convergence rate should be validated as the spatial grid is refined and as the time-step is reduced. To compute error one must study either a special solution, a well-resolved benchmark calculation, or both.
- **Comparisons with Other Code.** One of the most useful methods to validate both your algoritms and your model is through code comparisons. This can be done either by installing various options in your own code or by using another code.

When validating an algorithm, comparisons should be done on the same model. For example, give yourself the capability to choose from among several algorithms in your code. Algorithms that are known to be accurate can be used to validate faster algorithms. For example, the direct solution of a linear system can be used to validated an iterative solver, even if the direct solver is far too slow to be used in a full simulation. Similarly, one iterative solver can be used to validate another. Validation

on the level of a full simulation can be studied by comparing with a simulation by a mature code that uses completely different algorithms. For example, you can compare a Monte Carlo simulation with a finite element simulation of the same model.

When validating a model, comparisons should be done either with the same algorithm or in such a way as to minimize the effect of any algorithm differences. For example, when validating a model of chemical reactions, one should compare it to a more complete model solved by the same algorithm. On the other hand, when comparing a diffusive model of particle transport with a kinetic one the algorithms will be very different.

- **Parametric Sensitivity Studies.** One should have a sense of how sensitive your simulations are to uncertainties in your model. You should know if a small change in the value of some parameter in your model will lead to a large change in a critical predicted value. There are three basic approaches to sensitivity studies: ensembles, linearization, and adjoints. These will be discussed later in the term.
- **Comparisons with Benchmark Calculations.** In any mature area of scientific computation there is usually a collection of so-called benchmark calculations in the literature to which you can compare your simulations. A benchmark calculation is usually carried out using a well-validated "full physics" model on a state-of-the-art platform over weeks or months. While such comparisions should never replace comparisons with experimental data, their value is that they usually offer more data than an experiment can provide.
- **Comparisons with Experimental Data.** These are the ultimate tests for any scientific code. Before making any such comparison you should check that the experiment is being carried out in a regime where your model is valid. If it is, then you had better reproduce the data to within its error bars. This is particularly impressive if your simulations were done before you saw the experimental data, and even more so if your code predicted an unanticipated phenomenon.

Profile your app performance

bookmark_border

An app is considered to have poor performance if it responds slowly, shows choppy animations, freezes, or consumes too much power. Fixing performance problems involves identifying areas in which your app makes inefficient use of resources such as the CPU, memory, graphics, network, or the device battery.

To find and fix these problems, use the profiling and benchmarking tools and techniques described in this topic. To learn techniques for measuring performance and examples of how to use these techniques to resolve specific problems, see Measuring performance.

Android Studio offers several profiling tools to help find and visualize potential problems:

- **CPU profiler** helps track down runtime performance issues.
- **Memory profiler** helps track memory allocations.
- **Energy profiler** tracks energy usage, which can contribute to battery drain.

These tools are compatible with Android 5.0 (API level 21) and higher. For more information about the tools, see the other pages in this section of the user guide.

The Jetpack Benchmark libraries allow your application to measure various important operations:

- **Macrobenchmark**: Measure important performance use cases, including application startup and redrawing that is triggered by actions such as UI animations or scrolling.
- **Microbenchmark**: Measure CPU cost of specific functions.

To learn more about these libraries, see the Benchmark your app page.

Profileable applications

Profileable is a manifest configuration introduced in Android Q. It can specify whether the user of the device can profile this application through tools such as Android Studio, Simpleperf, and Perfetto.

Prior to profileable, most developers could only profile debuggable apps on Android, which added significant performance costs as a side effect. These performance costs could invalidate profiling results, especially if they were related to timing. Table 1 summarizes the differences between debuggable and profileable apps.

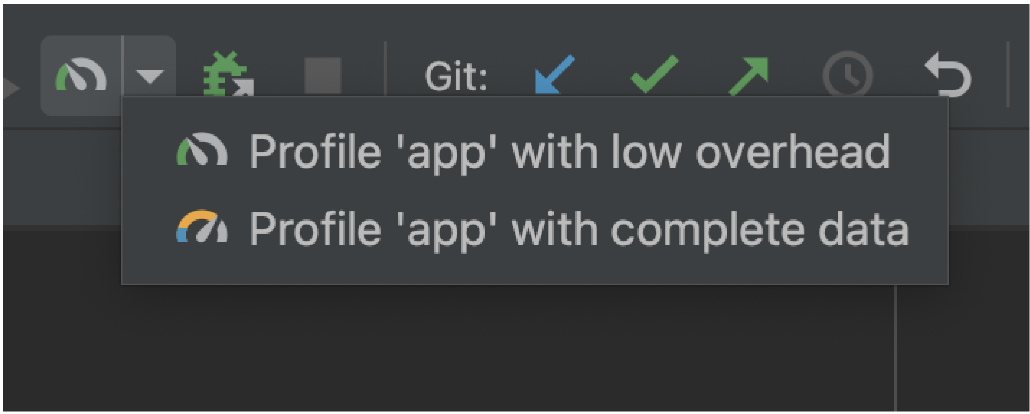
Table 1. Summary of key differences between debuggable and profileable apps.

Feature	**Debuggable**	**Profileable**
Memory Profiler	Full	Yes: • Default view • Native Memory Profiler No: • Event timeline • Heap dump • Live allocation recording
CPU Profiler	Full	Yes: • Default view • UI-initiated recording No: • Event timeline • Display status of API-initiated recording
Network Profiler	Yes	No
Energy Profiler	Yes	No
Power Profiler	Yes	Yes
Event Monitor	Yes	No

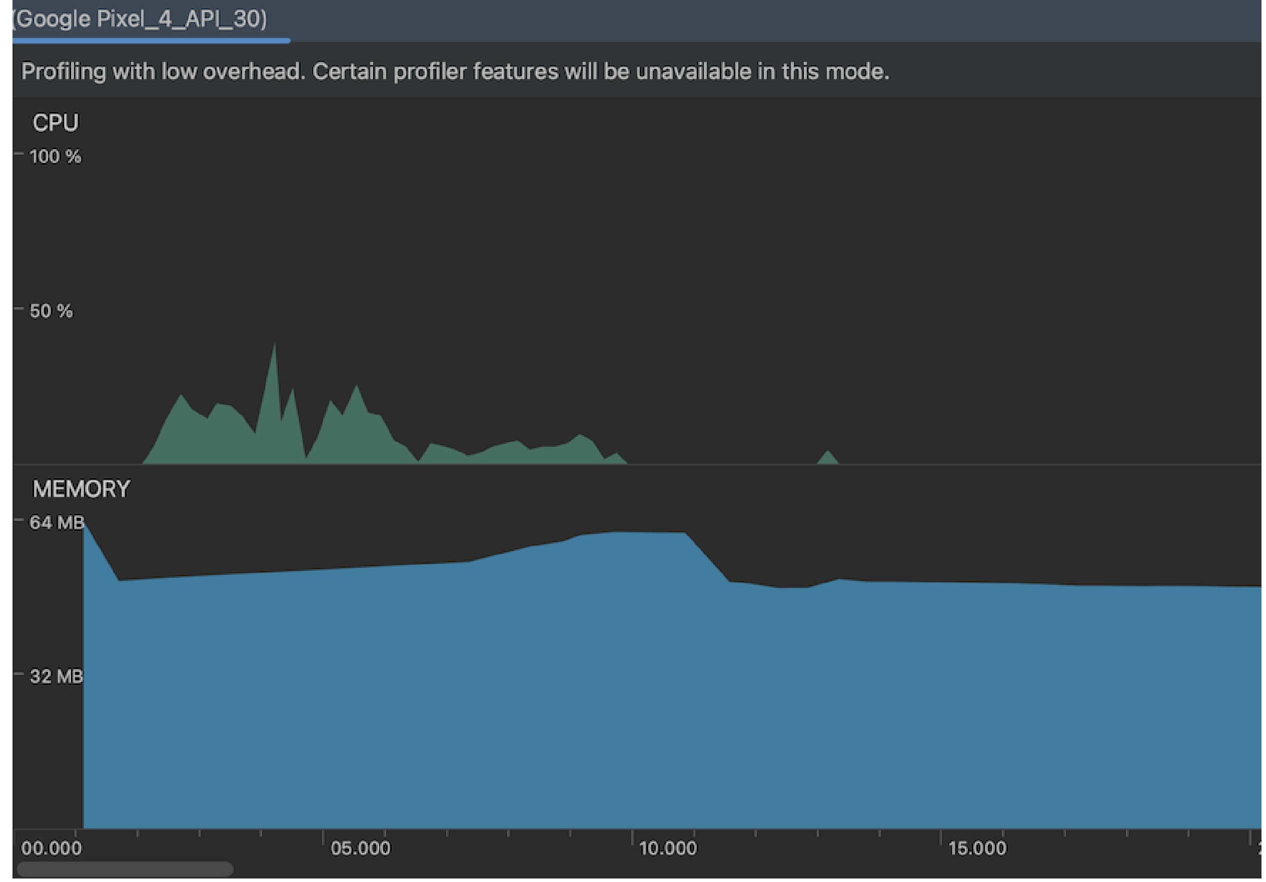
Profileable has been introduced so that developers can choose to allow their apps to expose information to profiling tools, while incurring very little performance costs. A profileable APK is essentially a release APK with a line of <profileable android:shell="true"/> added within the <application> section of the manifest file.

Build and run a profileable app automatically

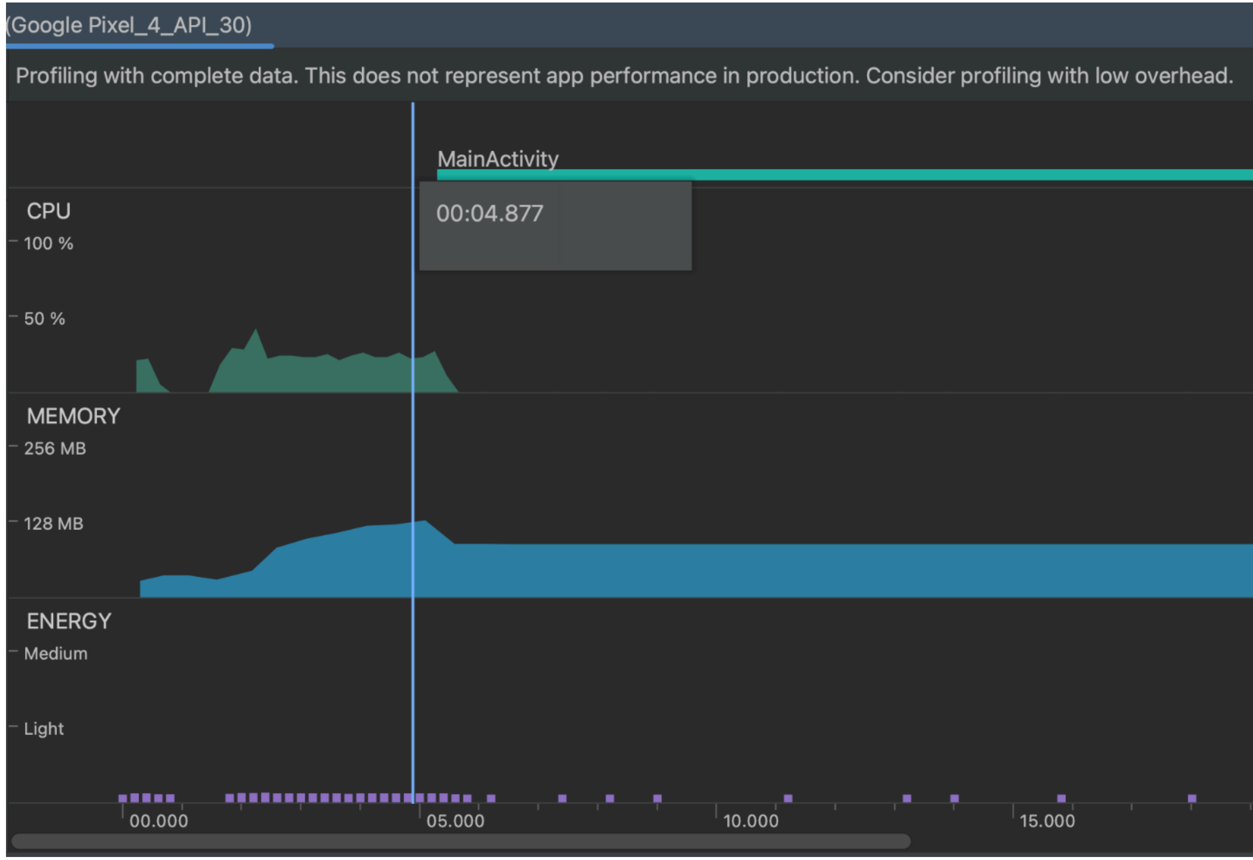
You can configure, build, and run a profileable app with one click. This feature requires a virtual or physical test device that runs API level 29 or higher and has Google Play. To use the feature, click the arrow next to the **Profile app** icon and choose between two options:



- **Profile 'app' with low overhead** starts the CPU and Memory profilers. In the Memory profiler, only [Record Native Allocations] is enabled.



- **Profile 'app' with complete data** starts the CPU, Memory, and Energy profilers.



Build and run a profileable app manually

To build a profileable application manually, you need to first build a release application and then update its manifest file, which turns the release application into a profileable application. After you configure the profileable application, launch the profiler and select a profileable process to analyze.

Build a release app

To build a release application for profiling purposes, do the following:

1. Sign your application with the debug key by adding the following lines to your application's build.gradle file. If you already have a working release build variant, you can skip to the next step.
2. buildTypes {
3. release {
4. signingConfig signingConfigs.debug
5. }
6. }

7. In Android Studio, select **Build** > **Select Build Variant...** and choose the release variant.

Change release to profileable

1. Convert your release application from above into a profileable application by opening the AndroidManifest.xml file and adding the following within <application>. For more details, see Building your application for release.

<profileable android:shell="true"/>

2. Depending on the SDK version, you may need to add the following lines to the application's build.gradle file.

3. aaptOptions {

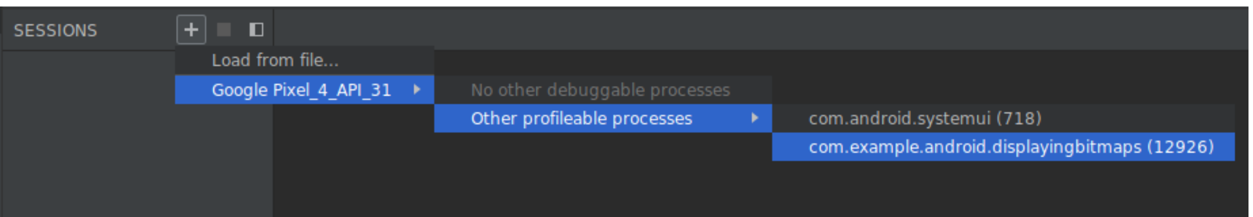
4. additionalParameters =["--warn-manifest-validation"]

5. }

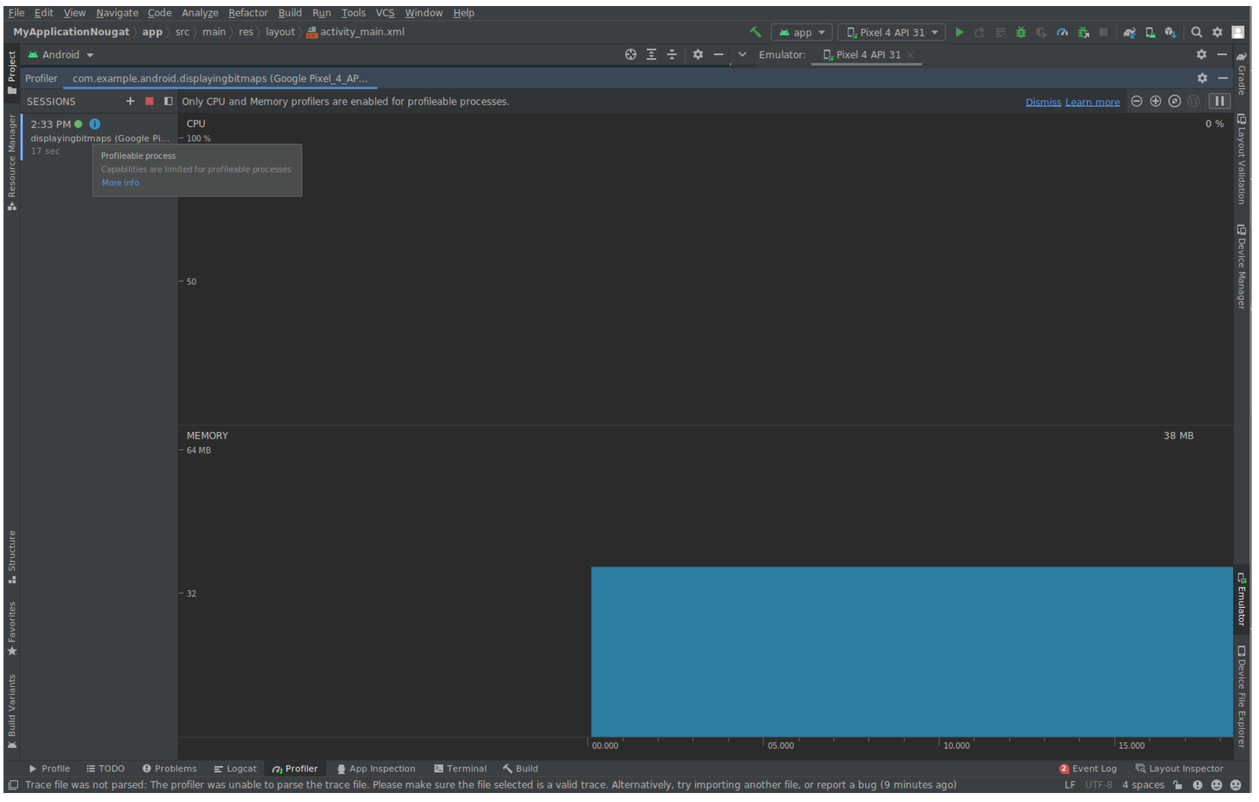
Profile a profileable app

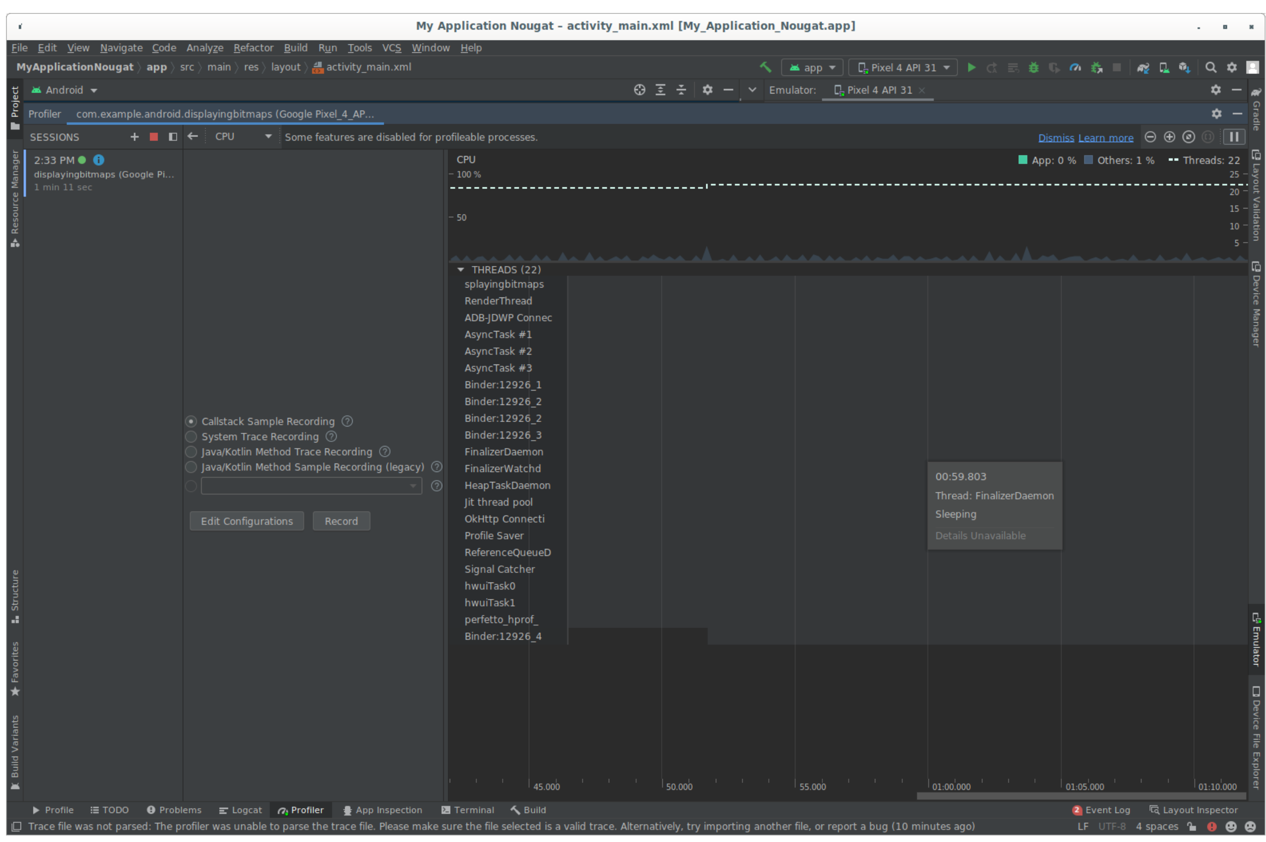
To profile a profileable app, do the following:

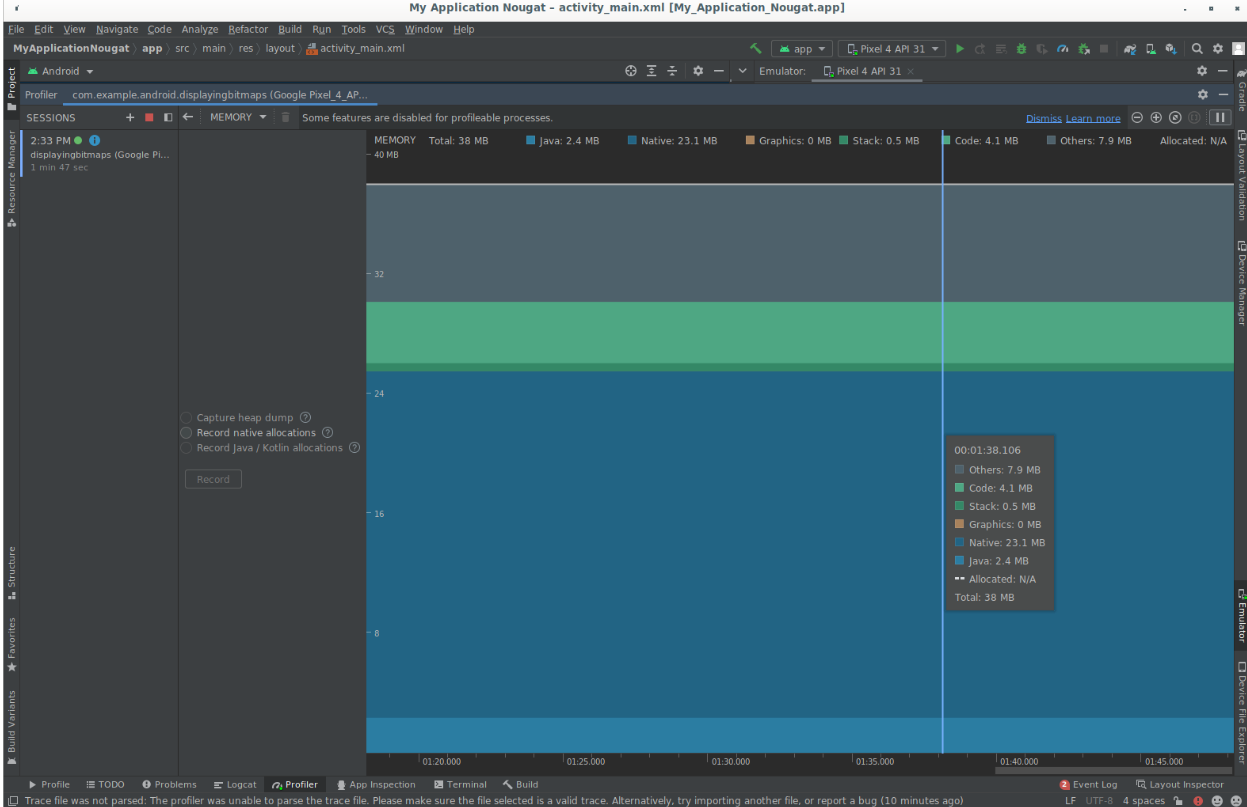
1. From the development emulator or device, start the app.
2. In Android Studio, launch the profiler by selecting **View** > **Tool Windows** > **Profiler**.
3. After the application has launched, click the + button in the profiler to see the dropdown menu. Select your device, then select the application's entry under **Other profileable processes**.



4. The profiler should attach to the application. Only the CPU and Memory Profilers are available, with limited capabilites for the Memory Profiler.


File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
MyApplicationNougat app src main res layout activity_main.xml
Profiler com.example.android.displayingbitmaps (Google Pixel_4_AP...
SESSIONS
Only CPU and Memory profilers are enabled for profileable processes.
Dismiss Learn more
2:33 PM
displayingbitmaps (Google Pi...
17 sec
Profileable process
Capabilities are limited for profileable processes
More info
CPU
100 %
0 %
50
MEMORY
64 MB
38 MB
32
00.000
05.000
10.000
15.000
Trace file was not parsed: The profiler was unable to parse the trace file. Please make sure the file selected is a valid trace. Alternatively, try importing another file, or report a bug (9 minutes ago)


My Application Nougat - activity_main.xml [My_Application_Nougat.app]
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
MyApplicationNougat app src main res layout activity_main.xml
Profiler com.example.android.displayingbitmaps (Google Pixel_4_AP...
SESSIONS
CPU
Some features are disabled for profileable processes.
Dismiss Learn more
2:33 PM
displayingbitmaps (Google Pi...
1 min 11 sec
App: 0 % Others: 1 % Threads: 22
THREADS (22)
splayingbitmaps
RenderThread
ADB-JDWP Connec
AsyncTask #1
AsyncTask #2
AsyncTask #3
Binder:12926_1
Binder:12926_2
Binder:12926_2
Binder:12926_3
FinalizerDaemon
FinalizerWatchd
HeapTaskDaemon
Jit thread pool
OkHttp Connecti
Profile Saver
ReferenceQueueD
Signal Catcher
hwuiTask0
hwuiTask1
perfetto_hprof_
Binder:12926_4
Callstack Sample Recording
System Trace Recording
Java/Kotlin Method Trace Recording
Java/Kotlin Method Sample Recording (legacy)
Edit Configurations
Record
00:59.803
Thread: FinalizerDaemon
Sleeping
Details Unavailable
45.000
50.000
55.000
01:00.000
01:05.000
01:10.000
Trace file was not parsed: The profiler was unable to parse the trace file. Please make sure the file selected is a valid trace. Alternatively, try importing another file, or report a bug (10 minutes ago)



Sessions

You can save Profiler data as *sessions*, which are retained until you quit Android Studio. By recording profiling information in multiple sessions and switching between them, you can compare resource usage in various scenarios.

- To start a new session, click the **Start a new profiling session** button and select an app process from the dropdown menu that appears.
- When you record a trace or capture a heap dump, Android Studio adds that data (along with your app's network activity) as a separate entry to the current session.
- To stop adding data to the current session, click **Stop the current profiling session**.
- To import a trace exported from a previous run of Android Studio, click **Start new profiler session** and choose **Load from file**.

Enable additional support for older devices (API level < 26)

To show you additional profiling data when running a device with Android 7.1 or lower, Android Studio must inject monitoring logic into your compiled app. These additional profiling data include the following:

- The event timeline on all profiler windows
- The number of allocated objects in Memory Profiler
- Garbage collection events in Memory Profiler

- Details about all transmitted files in Network Profiler

Note: These features are available by default if your device is running Android 8.0 or higher.

To enable additional support for older devices, follow these steps:

1. Select **Run > Edit Configurations**.
2. Select your app module in the left pane.
3. Click the **Profiling** tab, and then check **Enable additional support for older devices (API level < 26)**.
4. Build and run your app again.

Enabling additional support for older devices makes the build process slower, so you should enable it only when you want to start profiling your app.

Note: Additional support for older devices is not available for native code on devices using Android 9 or lower. If your app is a pure native app targeting Android 9 or lower (that is, it doesn't have a Java **Activity** class), the additional profiling data are not available. If your app uses JNI, some profiling data are available—such as the event timeline, garbage collection events, Java-allocated objects, and Java-based network activity—but it cannot detect native-based allocations and network activity.

Run standalone profilers

The standalone Android Studio Profilers let you profile your app without running the full Android Studio IDE.

To run the standalone profilers, do the following:

1. Make sure the profiler is not currently running inside of Android Studio.
2. Go to the installation directory and navigate to the bin directory:

Windows/Linux: *studio-installation-folder*/bin

macOS: The use of standalone profilers is not supported on macOS.

3. Depending on your OS, run profiler.exe or profiler.sh. The Android Studio splash screen appears. After the splash screen disappears, a profiler window opens.

Overview of measuring app performance

bookmark_border

This document helps you to identify and fix key performance issues in your app.

Key performance issues

There are many problems that can contribute to bad performance in an app, but the following are some common issues to look for in your app:

Scroll jank

Jank is the term that describes the visual hiccup that occurs when the system isn't able to build and provide frames in time to draw them to the screen at the requested cadence of 60hz or higher. Jank is most apparent when scrolling, when instead of smoothly animated flow, there are hiccups. Jank

appears when the movement pauses along the way for one or more frames, as the app takes longer to render content than the duration of a frame on the system.

Apps must target 90Hz refresh rates. Conventional rendering rates are 60Hz, but many newer devices operate in 90Hz mode during user interactions, such as scrolling. Some devices support even higher rates of up to 120Hz.

To see what refresh rate a device is using at a given time, enable an overlay using **Developer Options > Show refresh rate** in the **Debugging** section.

Startup latency

Startup latency is the amount of time it takes between tapping on the app icon, notification, or other entry point, and the user's data showing on the screen.

Aim for the following startup goals in your apps:

- Cold start in less than 500ms. A *cold start* happens when the app being launched isn't present in the system's memory. This happens when it is the app's first launch since reboot or since the app process is stopped by either the user or the system.

In contrast, a *warm start* occurs when the app is already running in the background. A cold start requires the most work from the system, as it has to load everything from storage and initialize the app. Try to make cold starts take 500ms or less.

- P95 and P99 latencies very close to the median latency. When the app takes a long time to start, it makes a poor user experience. Interprocess communications (IPCs) and unnecessary I/O during the critical path of app startup can experience lock contention and introduce inconsistencies.

Transitions that aren't smooth

This is apparent during interactions such as switching between tabs or loading a new activity. These types of transitions must be smooth animations and not include delays or visual flicker.

Power inefficiencies

Doing work reduces battery charge, and doing unnecessary work reduces battery life.

Memory allocations, which come from creating new objects in code, can be the cause of significant work in the system. This is because not only do the allocations themselves require effort from the Android Runtime (ART), but freeing these objects later (*garbage collection*) also requires time and effort. Both allocation and collection are much faster and more efficient, especially for temporary objects. Although it used to be best practice to avoid allocating objects whenever possible, we recommend you do what makes the most sense for your app and architecture. Saving on allocations at the risk of unmaintainable code isn't the best practice, given what ART is capable of.

However, it requires effort, so keep in mind that it can contribute to performance problems if you are allocating many objects in your inner loop.

Identify issues

We recommended the following workflow to identify and remedy performance issues:

1. Identify and inspect the following critical user journeys:

    - Common startup flows, including from launcher and notification.
    - Screens where the user scrolls through data.
    - Transitions between screens.
    - Long-running flows, like navigation or music playback.
2. Inspect what is happening during the preceding flows using the following debugging tools:
    - Perfetto: lets you see what is happening across the entire device with precise timing data.
    - Memory Profiler: lets you see what memory allocations are happening on the heap.
    - Simpleperf: shows a flamegraph of what function calls are using the most CPU during a certain period of time. When you identify something that's taking a long time in Systrace, but you don't know why, Simpleperf can provide additional information.

To understand and debug these performance issues, it's critical to manually debug individual test runs. You can't replace the preceding steps by analyzing aggregated data. However, to understand what users are actually seeing and identify when regressions might occur, it's important to set up metrics collection in automated testing and in the field:

- Startup flows
    - Field metrics: Play Console startup time
    - Lab tests: test startup with Macrobenchmark
- Jank
    - Field metrics
        - Play Console frame vitals: within the Play Console, you can't narrow down metrics to a specific user journey. It only reports overall jank throughout the app.
        - Custom measurement with FrameMetricsAggregator: you can use FrameMetricsAggregator to record jank metrics during a particular workflow.
    - Lab tests
        - Scrolling with Macrobenchmark.
        - Macrobenchmark collects frame timing using dumpsys gfxinfo commands that bracket a single user journey. This is a way to understand variation in jank over a specific user journey. The RenderTime metrics, which highlight how long frames are taking to draw, are more important than the count of janky frames for identifying regressions or improvements.

Set up your app for performance analysis

It's essential to properly set up to get accurate, repeatable, actionable benchmarks from an app. Test on a system that is as close to production as possible, while suppressing sources of noise. The

following sections show a number of APK- and system-specific steps you can take to prepare a test setup, some of which are use-case specific.

Tracepoints

Apps can instrument their code with custom trace events.

While traces are being captured, tracing does incur a small overhead of roughly 5µs per section, so don't put it around every method. Tracing larger chunks of work of >0.1ms can give significant insights into bottlenecks.

APK considerations

Caution: Don't measure performance on a debug build.

Debug variants can be helpful for troubleshooting and symbolizing stack samples, but they have severe non-linear impacts on performance. Devices running Android 10 (API Level 29) and higher can use profileable android:shell="true" in their manifest to enable profiling in release builds.

Use your production-grade code shrinking configuration. Depending on the resources your app uses, this can have a substantial impact on performance. Some ProGuard configurations remove tracepoints, so consider removing those rules for the configuration you're running tests on.

Compilation

Compile your app on-device to a known state—generally speed or speed-profile. Background just-in-time (JIT) activity can have significant performance overhead, and you reach it often if you are reinstalling the APK between test runs. The following is a command to do this is:

```
adb shell cmd package compile -m speed -f com.google.packagename
```

The speed compilation mode compiles the app completely. The speed-profile mode compiles the app according to a profile of the utilized code paths that is collected during app usage. It can be difficult to collect profiles consistently and correctly, so if you decide to use them, confirm they are collecting what you expect. The profiles are located in the following location:

```
/data/misc/profiles/ref/[package-name]/primary.prof
```

Macrobenchmark lets you directly specify compilation mode.

System considerations

For low-level and high fidelity measurements, calibrate your devices. Run A/B comparisons across the same device and same OS version. There can be significant variations in performance, even across the same device type.

On rooted devices, consider using a lockClocks script for Microbenchmarks. Among other things, these scripts do the following:

- Place CPUs at a fixed frequency.
- Disable small cores and configure the GPU.
- Disable thermal throttling.

We don't recommend using a lockClocks script for user-experience focused tests such as app launch, DoU testing, and jank testing, but it can be essential for reducing noise in Microbenchmark tests.

When possible, consider using a testing framework like [Macrobenchmark](), which can reduce noise in your measurements and prevent measurement inaccuracy.

Slow app startup: unnecessary trampoline activity

A trampoline activity can extend app startup time unnecessarily, and it's important to be aware if your app is doing it. As shown in the following example trace, one activityStart is immediately followed by another activityStart without any frames being drawn by the first activity.

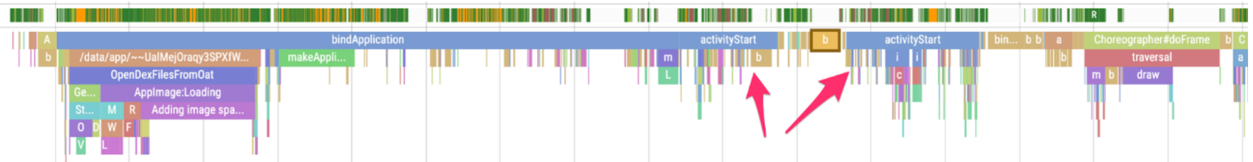


Figure 1. A trace showing trampoline activity.

This can happen both in a notification entrypoint and a regular app startup entrypoint, and you can often address it by refactoring. For example, if you're using this activity to perform setup before another activity runs, factor this code out into a reusable component or library.

Note: To improve app performance and UX, apps that target Android 12 or higher can't start activities from services or broadcast receivers that are used as trampolines into an activity.

Unnecessary allocations triggering frequent GCs

You might see garbage collections (GCs) are happening more frequently than you expect in a Systrace.

In the following example, every 10 seconds during a long-running operation is an indicator that the app might be allocating unnecessarily but consistently over time:

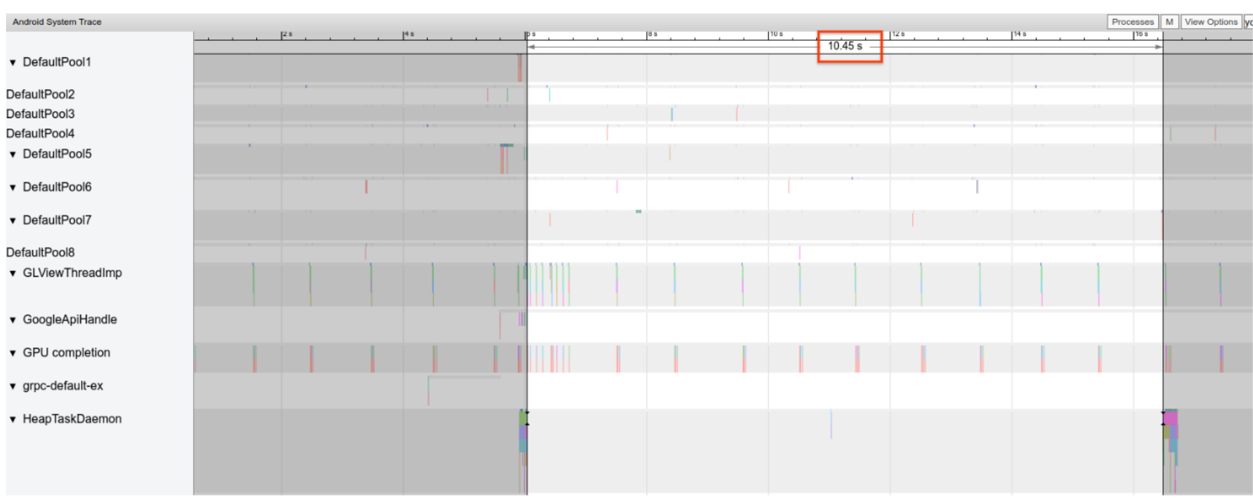


Figure 2. A trace showing space between GC events.

You might also notice that a specific call stack is making the vast majority of the allocations when using the Memory Profiler. You don't need to eliminate all allocations aggressively, as this can make code harder to maintain. Instead, start by working on hotspots of allocations.

Janky frames

The graphics pipeline is relatively complicated, and there can be some nuance in determining whether a user ultimately might see a dropped frame. In some cases, the platform can "rescue" a frame using buffering. However, you can ignore most of that nuance to identify problematic frames from your app's perspective.

When frames are being drawn with little work required from the app, the Choreographer.doFrame() tracepoints occur on a 16.7ms cadence on a 60 FPS device:

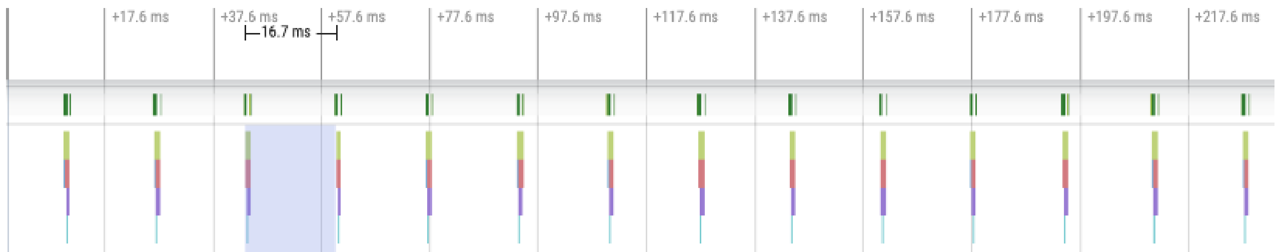


Figure 3. A trace showing frequent fast frames.

If you zoom out and navigate through the trace, you sometimes see frames take a little longer to complete, but it's still okay because they're not taking more than their allotted 16.7ms time:

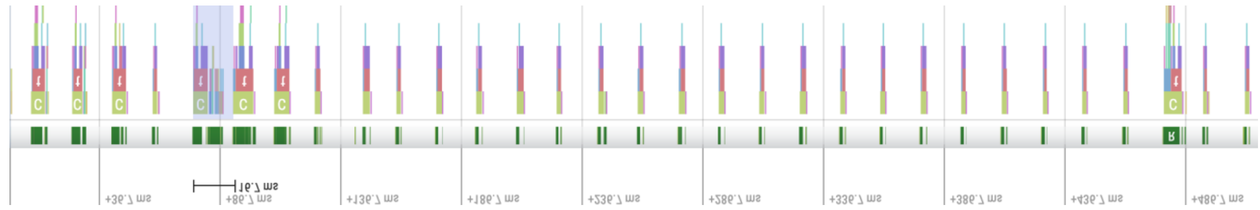


Figure 4. A trace showing frequent fast frames with periodic bursts of work.

When you see a disruption to this regular cadence, it is a janky frame, as shown in figure 5:

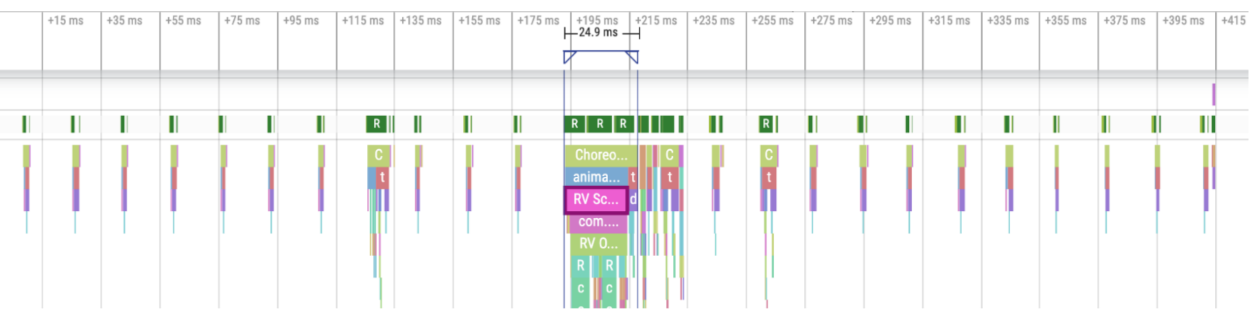


Figure 5. A trace showing a janky frame.

You can practice identifying them.

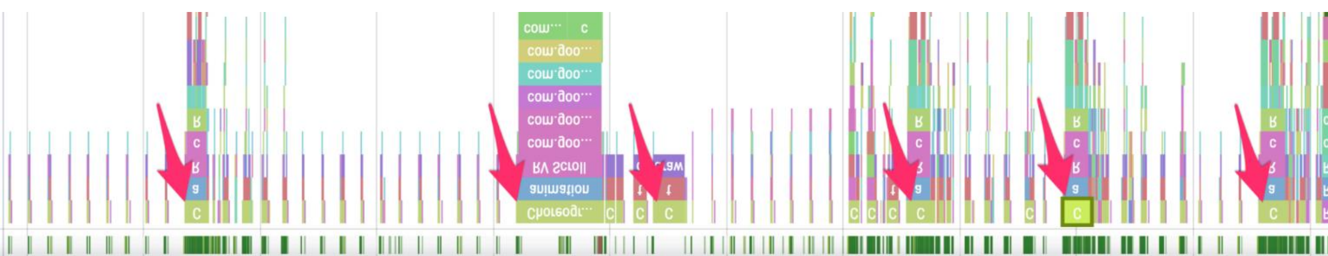


Figure 6. A trace showing more janky frames.

In some cases, you need to zoom into a tracepoint for more information about which views are being inflated or what RecyclerView is doing. In other cases, you might have to inspect further.

For more information about identifying janky frames and debugging their causes, see Slow rendering.

Common RecyclerView mistakes

Invalidating the entire backing data of RecyclerView unnecessarily can lead to long frame rendering times and jank. Instead, to minimize the number of views that need to update, invalidate only the data that changes.

See Present dynamic data for ways to avoid costly notifyDatasetChanged() calls, which cause content to update rather than replacing it entirely.

If you don't support every nested RecyclerView properly, it can cause the internal RecyclerView to be completely recreated every time. Every nested, inner RecyclerView must have a RecycledViewPool set to help ensure views can be recycled between every inner RecyclerView.

Not prefetching enough data, or not prefetching in a timely manner, can make reaching the bottom of a scrolling list jarring when a user needs to wait for more data from the server. Although this isn't technically jank, as no frame deadlines are being missed, you can significantly improve UX by modifying the timing and quantity of prefetching so that the user doesn't have to wait for data.

8 Debugging Techniques

What Is Debugging?

Debugging is the process of identifying and removing defects you find in your program or product. While it's certainly a fundamental part of QA testing, QA is a more comprehensive phase that involves ensuring other aspects of quality, such as performance and usability, not just finding bugs.

It's impossible to completely guarantee that a product is error-free, but debugging goes a long way to minimize the occurrence of these defects. By locating and addressing the problems in the code as early as possible, you can prevent issues from interfering with the program later on, when it's in the consumer's hands.

1. Ask Yourself the Right Questions

Start the debugging process by defining the problem and asking yourself questions about it. For example:

- What do you need the program to do?
- What does it actually do?
- What are the issues you've found?
- Have you encountered these types of problems before?
- What did you do to fix them?
- Where and why do you think the bugs occurred?

Often, asking these questions will lead to your forming some kind of hypothesis about the nature of the errors, which will allow you to find the root cause and resolve them.

2. Pay Attention to Error Messages

Error messages aren't just there as an annoyance — they can actually tell you exactly what the problem is with your software. So, when an error message pops up, make sure you read it, because it can give you a lot of insight into what's going on with the product.

If you're not sure what the error message means, try searching for it online. Chances are someone else has encountered the same problem in the past and could know precisely how to fix it.

3. Leverage a Debugger

A debugger, also known as a debugging tool or debugging mode, can be used to easily identify and correct bugs. To effectively leverage the tool, you'll need to run your program within the debugger, which allows you to monitor it in real-time and see the error when it occurs. You can pause the program while it's running to pinpoint and investigate any issues that are occurring and review your code line by line.

You'll typically use a debugger after you encounter an exception. With the tool, you can identify the problem that's taking place.

4. Log Everything

Make sure you're logging every issue you encounter, as well as steps you take to address them and ensure your program is running correctly. Once you've documented the error, you can start mapping out potential scenarios and solutions. You should keep track of all possible steps to take and the information you need to make a decision regarding your errors. This will also allow you to navigate different potential solutions.

5. Localize the Problem

The method of problem localization entails removing pieces of code line by line until you find the issue that is interfering with your program. While this is a somewhat painstaking and involved way of identifying the error that's taking place, it can be highly effective in determining what, exactly, is going wrong with your product. Of course, you'll need to keep repeating the process until you've tracked down the bugs.

6. Try to Replicate the Problem

By replicating the problem, you'll find out what the nature of the problem is precisely. In fact, this can lead to you creating better, cleaner code in general since you're exercising the critical thinking skills required to find the cause of an issue.

This, of course, demands a thorough investigation of the ins and outs of the product. But once you've successfully reproduced the error that's interfering with your product's performance, usability, or functionality, fixing the problem should require far less time. In fact, most of the time, replicating the issue is the hard work, while resolving it takes only minutes.

You might turn to these solutions either because a client comes to you with a problem, and in order to fix it, you'll need to reproduce it, or as a debugging technique to have in your arsenal when you're developing products from the beginning.

7. Turn to the Community

It's highly likely that any error you encounter is one others have encountered before you. It can be very helpful to turn to a community associated with the language, framework, or another development tool you're using to find a solution for addressing the bug you've encountered. Many development tools, such as languages like Python and frameworks like Ruby on Rails, have huge, thriving communities, offering an abundance of support to developers within them.

8. Test, Test, and Test Again

The best way to spot bugs — and successfully resolve them before they derail your app — is by repeatedly testing the product. While the QA team will more thoroughly vet the product, developers themselves can script simple tests during the development phase, such as unit testing, which involves individually testing different pieces of the code — units.

These are some of the many simple debugging techniques you should have in your toolkit for building and refining software and ultimately making better products. Different methods will make more sense depending on the precise problems and scenarios you encounter, but it's a good idea to become well-versed in all of them to have more tools at your disposal and improve your development skills.

What is data profiling?

Data profiling, or data archeology, is the process of reviewing and cleansing data to better understand how it's structured and maintain data quality standards within an organization. The main purpose is to gain insight into the quality of the data by using methods to review and summarize it, and then evaluating its condition. The work is typically performed by data engineers who will use a range of business rules and analytical algorithms.

Data profiling evaluates data based on factors such as accuracy, consistency, and timeliness to show if the data is lacking consistency or accuracy or has null values. A result could be something as simple as statistics, such as numbers or values in the form of a column, depending on the data set. Data profiling can be used for projects that involve data warehousing or business intelligence and is even more beneficial for big data. Data profiling can be an important precursor to data processing and data analytics.

Now available: watsonx.data

Scale AI workloads, for all your data, anywhere

Try watsonx.data

How does data profiling work?

Companies integrate software or applications to ensure data sets are prepared appropriately and can be used to the best of their advantage to remove bad data. Specifically, you can determine what sources have or are creating data quality issues, which ultimately affects your overall business operational and financial success. This process will also perform a necessary data quality assessment.

The first step of data profiling is gathering data sources and associated metadata for analysis, which can often lead to the discovery of foreign key relationships. The next steps that follow are meant to clean the data to ensure a unified structure and to eliminate duplication, among other things. Once the data has been cleaned, the data profiling software will return statistics to describe the data set and can include things such as the mean, minimum/maximum value, and frequency. Below, we will outline for you proper data profiling techniques.

Data profiling vs. data mining

While there is overlap with data mining, data profiling has a different goal in mind. What is the difference?

- Data profiling helps in the understanding of data and its characteristics, whereas data mining is the process of discovering patterns or trends by analyzing the data.
- Data profiling focuses on the collection of metadata and then using methods to analyze it to support data management.
- Data profiling, unlikely data mining, produces a summary of the data's characteristics and enables use of the data.

In other words, data profiling is the first of the tools you use to ensure the data is accurate and there are no inaccuracies.

Types of data profiling

Data profiling should be an essential part of how an organization handles its data and companies should look at it as a key component of data cleaning. It not only can help you understand your data, it can also verify that your data is up to standard statistical measure. A team of analysts can approach data profiling in many different ways, but typically falls into three major categories with the same goal in mind which is to improve the quality of your data and gain a better understanding.

Here are the approaches analysts may use to profile your data:

- **Structure discovery**: This approach focuses on the format of the data and ensuring it is consistent all throughout the database. There are a number of different processes analysts might use for this type when examining the database. One is pattern matching, which can help you to understand format-specific information. An example of this is if you're lining up phone numbers and one has a missing value. This is something that could be caught in structure discovery.
- **Content discovery:** This type is when you analyze data rows for errors or systemic issues. This process is a closer look at the individual elements of the database and can help you find incorrect values.
- **Relationship discovery**: This type entails finding out what data is in use and trying to find the connection between each set. In order to do this, analysts will begin with metadata analysis to figure out what the relationships are between data and then narrow down the connections between specific fields.

Benefits and challenges of data profiling

Generally speaking, there are little to no downfalls when profiling your data. It is one thing when you have a good amount of data, but the quality matters and that's when data profiling comes into play. When you have standardized data that is precisely formatted it leaves little to no chance for there to be unhappy clients or miscommunication.

The challenges are mostly systemic in nature because if, for instance, your data is not all in one place it makes it very difficult to locate. But with the installment of certain data tools and applications it shouldn't be an issue and can only benefit a company when it comes to decision-making. Let's take a closer look at other key benefits and challenges.

Benefits

Data profiling can offer a high-level overview of data unlike any other tool. More specifically, you can expect:

- **More Accurate Analytics**: A complete data profiling will ensure better quality and more credible data. Properly profiling your data can help make better sense of the relationship between different data sets and sources, and help support data governance procedures.
- **Keeps Information Centralized:** By examining and analyzing your data through data profiling you can expect your data quality to be much higher and well-organized. The review of source data will eliminate errors and highlight the areas with the most issues. It will then produce insight and organization that centralizes your data in the best way possible.

Challenges

Data profiling challenges typically stem from the complexity of the work involved. More specifically, you can expect:

- **Expensive and time-consuming:** Data profiling can become very complex when trying to implement a successful program due in part because of the sheer volume of data being collected by a typical organization. This can become very expensive and a time-consuming task to hire trained experts to analyze the results and then make decisions without the correct tools.
- **Inadequate resources:** In order to start the data profiling process a company needs its data all in one place, which is often not the case. If the data lives across different departments and there isn't a trained data professional in place it can become very difficult to data profile a company as a whole.

Data profiling tools and best practices

No matter what the approach may be, the following data profiling tools and best practices optimize data profiling accuracy and efficiency:

Colum profiling: This method scans tables and counts the number of times each value shows up within each column. Column profiling can be useful in finding frequency distribution and patterns within a column.

Cross-column profiling: This technique is made up of two processes: key analysis and dependency analysis. The key analysis process looks at the array of attribute values by scouting for a possible primary key. While the dependency analysis process works to identify what relationships or patterns are embedded within the data set.

Cross-table profiling: This technique uses key analysis to identify stray data. The foreign key analysis identifies orphaned records or general differences to examine the relationship between column sets in different tables.

Data rule validation: This method assesses data sets against established rules and standards to verify that they're in fact following those predefined rules.

Key Integrity: Ensuring keys are always present in the data and identifies orphan keys, which can be problematic.

Cardinality: This technique checks relationships such as one-to-one and one-to-many, between data sets.

Pattern and frequency distribution: This technique ensures data fields are formatted correctly.

Data profiling use cases

While data profiling can enhance accuracy, quality and usability in multiple contexts across industries, its more prominent use cases include:

Data transformation: Before data can be processed it needs to be transformed into a useable and organized set. This is an important step before creating a prediction model and examining the data, therefore data profiling must be done prior to any of these steps. In fact, the IBM Db2 Warehouse on Cloudis an elastic cloud data warehouse built for high-performance analytics and AI. This data warehouse allows you to aggregate data from across your business.

Additionally, ELT (extra, load, transform) and ETL (extract, transform, load) are data integration processes that move raw data from a source system to a target database. IBM offers data integration

services and solutions to support a business-ready data pipeline and give your enterprise the tools it needs to scale efficiently.

Data Integration: In order to properly integrate multiple datasets, you have to first understand the relationships between each dataset. This is a vital step when trying to understand the metrics of the data and determining how to link them.

Query Optimization: If you want to have the most accurate and optimized information about your company, data profiling is key. Data profiling takes into account information on the characteristics of a database and creates statistics about each database. IBM i 7.2 software provides database performance and query optimization for just that purpose. The goal of database turning is to minimize the response time of your queries by making the best use of your system resources.

In-memory Cache for Mobile Data Management

For more than a decade, mobile data has been a focus of research for data and IT specialists due to the inherent challenges that come with its management. Mobile computing is an excellent innovation, and a measure of a company's digital maturity 'but it comes with a price. This innovation brings with it challenges related to intermittent network connections, low bandwidth, and scarcity of resources or infrastructure. To provide the best possible experience, businesses should find a way to ensure quality of both data and service is maintained. This means that consistency and coherence of data should be the goal, while also providing high availability and quick response times.

Mobility has varying implications on how data is managed, and these implications often decide what methods or strategies to use in mobile data management. It presents challenges at the networking layer but also provides opportunities at the higher layers through leveraging location information to provide personalization and better performance through data prefetching. Mobile data management falls under two distinct infrastructures: single-hop, where each mobile device communicates via a stationary host that routes the data, and multi-hop, where an ad-hoc wireless network is formed between the data source and requester that forms a dissemination tree where in-network processing can be achieved.

Caching provides benefits that improve system performance and availability for both infrastructures because it eliminates the need for performing I/O or remote data retrieval. By keeping data requests in main memory, service time is reduced and network bottlenecks are avoided. In mobile computing, however, improving performance goes beyond increasing speed and mitigating delays in data retrieval; data consistency and efficient data dissemination should also be a focus. Ultimately, the goal should be ensuring the accuracy and correctness of the operations performed on cached data.

Maintaining Data Consistency and Integrity

The challenges related to speed and data consistency in mobile communication are doubled due to the inherent limitations of mobile devices. The limited bandwidth of wireless communication increases response times since data has to be accessed remotely from a mobile host; the high-energy consumption of transmitting and receiving data is also taxing on the relatively small batteries of mobile devices. This poses a challenge because data and the operations performed on the mobile device needs to be synchronized with the mobile host and other sites used to route data.

The complexity of this synchronization is reduced if updates are allowed on the mobile device so that data can be cached. By caching data locally, being disconnected from the network won't affect performance. Essentially, there are two ways to propagate updates and ensure correctness. First, all copies of an item within a single transaction can be synchronized, which is referred to as eager replication; second, transactions that help keep coherent replicas are run independently as database transactions after the original transaction is done, which is referred to as lazy replication. Serializability, which is one of the main correctness criteria, can be achieved by either of these two methods.

7 Best Practices for Successful Data Management

1. Build strong file naming and cataloging conventions

If you are going to utilize data, you have to be able to find it. You can't measure it if you can't manage it. Create a reporting or file system that is user- and future-friendly—descriptive, standardized file names that will be easy to find and file formats that allow users to search and discover data sets with long-term access in mind.

- To list dates, a standard format is YYYY-MM-DD or YYYYMMDD.
- To list times, it is best to use either a Unix timestamp or a standardized 24-hour notation, such as HH:MM:SS. If your company is national or even global, users can take note of where the information they are looking for is from and find it by time zone.

2. Carefully consider metadata for data sets

Essentially, metadata is descriptive information about the data you are using. It should contain information about the data's content, structure, and permissions so it is discoverable for future use. If you don't have this specific information that is searchable and allows for discoverability, you cannot depend on being able to use your data years down the line.

Catalog items such as:

- Data author
- What data this set contains
- Descriptions of fields
- When/Where the data was created
- Why this data was created and how

This information will then help you create and understand a data lineage as the data flows to tracking it from its origin to its destination. This is also helpful when mapping relevant data and documenting data relationships. Metadata that informs a secure data lineage is the first step to building a robust data governance process.

3. Data Storage

If you ever intend to be able to access the data you are creating, storage plans are an essential piece of your process. Find a plan that works for your business for all data backups and preservation methods. A solution that works for a huge enterprise might not be appropriate for a small project's needs, so think critically about your requirements.

A variety of storage locations to consider:

- Desktops/laptops
- Networked drives
- External hard drives
- Optical storage
- Cloud storage
- Flash drives (while a simple method, remember that they do degrade over time and are easily lost or broken)

Week-10 App Performance Optimization

Day 1: Introduction to App Performance Optimization

Why Performance Matters:

Understanding the impact of performance on user experience and app success.

Real-world examples of poorly performing apps.

Key Performance Metrics:

Identifying important metrics like response time, CPU usage, memory consumption, etc.

Day 2: Memory Management and Resource Optimization

Memory Management Techniques:

Understanding memory allocation and deallocation.

Implementing best practices for efficient memory usage.

Resource Optimization:

Techniques for optimizing resource-intensive operations (e.g., image loading, network requests).

Day 3: Performance Profiling and Debugging

Profiling Tools:

Introduction to performance profiling tools (e.g., Instruments for iOS, Android Profiler for Android).

Using these tools to identify performance bottlenecks.

Debugging Techniques:

Troubleshooting common performance issues.

Using profiling data to pinpoint areas for improvement.

Day 4: UI/UX Impact on Performance

Optimizing UI Rendering:

Strategies for efficient rendering of UI components.

Avoiding unnecessary layout recalculations.

Reducing Network Latency:

Techniques for minimizing network requests and optimizing data retrieval.

Day 5: Caching and Data Management

Implementing Caching:

Utilizing caching mechanisms to store and retrieve data.

Choosing the right caching strategy for your app.

Data Management Best Practices:

Strategies for handling large datasets efficiently.

Week-11 Mobile App Security and Privacy

What Is Mobile App Security?

Mobile application security refers to the technologies and security procedures that protect mobile applications against cyberattacks and data theft. An all-in-one mobile app security framework automates mobile application security testing on platforms like iOS, Android, and others.

Mobile device usage has been steadily increasing in recent years. Recent statistics note that about 90% of the global internet population uses a mobile device to go online. For hackers, this means more people to victimize, making endpoint security for mobile devices increasingly vital.

The Need for Mobile App Security

Mobile app security can guard against a variety of harmful consequences, including:

Personal and Login Data Theft

Losing sensitive data, such as client information and login passwords, typically stem from inadequate mobile app security, which hackers leverage to obtain access to sensitive information.

Stolen Financial Data

Mobile banking applications may contain customer financial information, including credit and debit card details. If a hacker successfully hijacks a banking app, they may also take control of the user's phone and perform a transaction without the victim's knowledge.

Intellectual Property Theft

Without adequate mobile app security, copyrights, patents, and other forms of intellectual property can fall into malicious hands. For example, every mobile application is built on a foundational piece of code. To develop copies of popular apps, which are intended to deceive users into downloading a fake version of the real software, hackers will attempt to steal the source codes. On mobile devices, these fake apps can be used to spread malware.

Reputational Damage

Security flaws in a mobile application can put a company's reputation at risk. User data being made public will destroy customers' faith in the app developer and damage the brand's reputation.

5 Reasons For Increased Security Threats to Mobile Apps

Mobile applications constantly face a barrage of threats because of the following:

1. Hackers Taking Advantage of App Platforms

Applications are downloaded via a mobile app platform, such as the Apple Store and Google Play Store. These platforms provide rules for secure application development, such as keychains and platform permissions. Hackers can take advantage of these platforms' communication systems to intercept information being transferred from the platform to a mobile application.

2. Insecure Data Storage

Data stored without the right safeguards poses a significant risk. An attack on the mobile device's operating system, jailbroken devices, and vulnerabilities in the application’s data maintenance

framework present critical security issues. As a result, apps can be hacked, enabling thieves to steal the data they contain.

3. Communication Vulnerabilities

Mobile applications transfer data using the standard client-server approach, which involves the device's carrier network, such as AT&T, and the internet. Hackers use communication security weaknesses to obtain access to private data. For example, an unprotected Wi-Fi network can be exploited via routers or proxy servers.

4. Poor Authentication Procedures

A skilled hacker can bypass standard identification processes and access information using a bogus identity. Online authentication procedures are not often required for mobile apps, making them more vulnerable than standard web applications.

5. Inadequate Data Encryption

Data encryption and decryption are necessary to send and receive data securely. But security can be jeopardized by subpar data encryption technology, which hackers can leverage to manipulate, steal, or alter the original data.

Most Common Vulnerabilities in Mobile Applications

While there are several ways to hack mobile applications, here are some of the most common vulnerabilities:

Server-side Vulnerabilities

Because the server stores and processes all the data necessary for the application to function—such as authentication data, business data, financial or transactional data, and personal data—most communication between an application and a user takes place via the server. Therefore, vulnerabilities in the server will put the security of the application in danger.

Storing Data Insecurely

A mobile application can store different kinds of data—such as cookies, text files, and device settings—using various storage media, including a Structured Query Language (SQL) database, information property list (.plist) file, data warehouse, Secure Digital (SD) card, or Extensible Markup Language (XML) file. To ensure the privacy of the sensitive data the application uses, encryption should be effective.

The Data Exchange Process and Man-in-the-Middle Attacks

As mentioned above, many mobile applications rely on communication with servers to function. An application delivers or receives many kinds of data, such as user session data, login credentials, financial data, and personal data, depending on the needs of the business.

Client-server communication uses Hypertext Transfer Protocol (HTTP), but because this protocol lacks internal security measures, communications can be intercepted, altered, or diverted.

Impact of Fragile Mobile App Security on Enterprises

Insufficient app security entails both immediate and long-term consequences. Immediately, the resulting reputational damage can lead to financial repercussions and lost customers. This is why application security is a key component of mobile device management.

Some consequences are more significant in the long run than in the short term. An attacker can take advantage of the holes in your app's security in several ways. For example, they can use ports for illicit communication or execute data theft and man-in-the-middle (MITM) attacks.

Mobile Apps Hacking Statistics

Figures related to mobile app hacking are sobering, to say the least. Here are a few:

1. The Slack mobile app hack exposed the credentials of more than 12 million users.
2. Thirteen different Android apps ended up leaking the data of up to 100 million users.
3. The breach of ParkMobile, a parking app, impacted up to 21 million users.
4. The hack on Portpass, a COVID passport app, exposed the personal data of 650,000 users.

7 Steps to Boost Mobile App Security

Using the following seven mobile app security best practices can significantly boost the security of mobile apps:

1. Increase User Authentication Security

Stronger mobile app access controls must incorporate additional ways of verifying users' identities. Look for an authentication server solution that supports different ways of deploying two-factor authentication (2FA) and password protection. Your authentication procedures can be based on:

1. How sensitive the application's data is
2. The extent of the reputational damage a breach can expose your company to

2. Ensure the Software Supply Chain Is Secure

The software supply chain for mobile applications includes components provided by third parties. When choosing libraries and frameworks for mobile apps, developers have to be careful. You want respected, well-maintained, open-source projects.

3. Secure Data

Data security includes making sure data cannot be read by anyone who intercepts it. Encryption transforms data into an unreadable format that threat actors cannot exploit, so make it a core component of any mobile apps security system.

4. Ensure Safely Managed Sessions

Ineffective session management can seriously compromise security in applications that hold sensitive information, such as online banking apps. As such, set session timeouts to one hour for low-security applications and 15 minutes for high-risk ones. Also, use industry-standard technologies for issuing security tokens and ensuring sessions are terminated when a different user logs in, for example.

5. Use the Concept of Least Privilege

Sensitive user data is unnecessarily exposed when an app demands more permissions than needed, significantly increasing the mobile application's attack surface. Developers should approach permissions more carefully, making sure only those needing access to perform their jobs get authorization.

6. Modify Your Testing Strategy

One way to modify your testing strategy is by switching from periodic tests to a continuous testing methodology. This means developers will conduct tests on an ongoing basis instead of at specific intervals. To do this, use automated testing and threat modeling to constantly scan for flaws that can put your app's users at risk of a cyberattack.

7. Use App Shielding

App shielding is designed to safeguard Android and iOS mobile apps from tampering, reverse-engineering, and other types of attacks. It protects the data inside apps by separating the application's data from the runtime environment, making it a valuable tool during a mobile app security test, either before or after an app has been deployed.

A common method of app shielding is runtime application self-protection (RASP). RASP keeps an eye on the application's internal state, inputs, and outputs, enabling developers to identify vulnerabilities in their apps during mobile application security testing. RASP technology can also thwart attempts to exploit vulnerabilities in applications that are already deployed.

7 Mobile App Security Risks and How to Mitigate Them

July 10, 2020 By Cypress Data Defense In Technical

Mobile app security is a moving target. The need for better functionalities and features along with rapid deployment of software updates often comes at the expense of mobile security.

One of the major concerns for mobile app development is the rising mobile app security risks, particularly to prevent data breaches.

According to a study, over 10,573 malicious mobile apps were blocked per day in 2018.

As technology advances, it has not only become easier to build and deploy apps, but also easy to crack a mobile application's security as developers are still writing insecure code.

Some attackers might try to crack a mobile app to find out more about the special features and other information about your mobile application. Others might do it to breach backend services.

But how do you avoid such mobile security threats?

Let's find out.

Top 7 Mobile App Security Risks and Ways to Mitigate Them

Here are the top mobile app security risks and ways to mitigate them:

1. Insecure Communication

In a common mobile app, data is typically exchanged in a client-server fashion. When the application transmits data, it traverses through the internet and the mobile device's carrier network.

Attackers might exploit mobile security vulnerabilities to intercept sensitive information or user data while it is traversing across the network.

What are the threat agents that exist in insecure communication?

- Malware on your mobile device
- A malicious actor who shares your local network (monitored or compromised wifi)
- Carrier or network devices (proxies, cell towers, routers, etc.)

Mobile developers often use SSl/TLS only during authentication but not elsewhere. This leads to an inconsistent security layer which increases the risk of exposing sensitive data such as credentials, personal information, session IDs, and more to interception by attackers.

Having a SSL/TLS does not imply that the mobile application is entirely secure. You need to implement strong security protocols throughout the mobile application and its network.

How Can You Prevent Insecure Communication?

Only establish a secure connection after authenticating the identity of the endpoint server. While applying SSL/TLS to your mobile application, make sure you implement it on the transport channels that the mobile app will use to transverse sensitive data such as session tokens, credentials, etc.

Use strong, industry standard cipher suites with appropriate key lengths. Apart from this, also consider using certificates signed by a trusted CA provider and refrain from allowing self-signed certificates. You should also consider certificate pinning for sensitive applications.

Remember to account for third-parties like social networks as well by using their TLS versions when a mobile application runs a routine using webkit/browser.

Consider applying an additional layer of encryption to any sensitive data before it is even given to the SSL channel. If security vulnerabilities are found in the SSL implementation, the encryption layer will act as a secondary defense against attacks.

2. Lack of Input Validation

Input validation is the process of assessing input data to ensure that it is properly formed, preventing malformed data that might consist of harmful code or may trigger malfunction in the mobile app.

What is the impact of poor input validation in mobile apps?

Why is it a mobile security threat? Here's why:

When the mobile application does not validate input properly, it puts the application at risk of exposure to attackers who might be able to inject malicious data input and gain access to sensitive data in the app or breach backend data stores.

Ideally, input validation should occur instantly after the data is received from an external system. This includes data from third-party vendors, partners, regulators, or suppliers, each of which could be compromised to deliver malformed data.

While input validation is not sufficient to be used as a primary defense against preventing mobile app security risks, it is a significant way to filter out malicious data if implemented properly.

How Can You Prevent Weak Input Validation?

You can implement input validation by using programming techniques that facilitate the effective enforcement of data correctness such as:

- Minimum and maximum value range check for dates and numerical parameters along with length check of strings
- Input validation against XML Schema and JSON Scheme
- Minimum and maximum value range check for strings, minimum and maximum length check for dates and numerical parameters.
- Regular expressions for any other structured data covering the entire input string (^...$) and avoiding using "any character" wildcard (e.g. as . or \S)
- Array of permitted values for small sets of string parameters (e.g. hours of days)

Alternatively, a more efficient way to prevent attacks caused by poor input validation is to only allow known good rather than only rejecting known bad. This can set up much more stringent controls if done properly.

If the input data is structured like social security numbers, dates, email addresses, zip codes, etc, then the mobile app developer should be able to build and implement a strong input data validation pattern on the basis of regular expressions.

However, if the input data comes in a fixed set of options, such as radio buttons or drop down list, then the input data should match exactly as one of the options available to the user from the mobile application.

3. Insecure Data Storage

Insecure data storage can occur in many different places within your mobile app such as binary data stores, SQL databases, cookies stores, and more. The vulnerability in using an insecure data storage is if you use one, it could be compromised due to issues with jailbroken devices, frameworks, or other attacks.

Attackers can easily circumvent the security protocols of a mobile app if not implemented correctly, such as poor encryption libraries that can be bypassed by jailbreaking or rooting the mobile device.

If an attacker gains access to a database or device, they can modify the legitimate app to extract information to their systems.

What is the impact of insecure data storage?

- Insecure data storage may result in the following:
- Intellectual property (IP) loss
- Identity theft
- Fraud
- Privacy violations
- Reputation damage

Many times, insecure data storage also occurs due to a lack of processes to handle the cache of key presses, images, and data.

How Can You Prevent Insecure Data Storage?

Avoid the "MODE WORLD READABLE" or "MODE WORLD WRITABLE" modes for IPC files as they do not offer the ability to control data format or limit data access to specific applications.

However, if you want to share data with other app processes, consider using a content provider which provides specific read and write permissions to other apps with dynamic permission access on a case-by-case basis.

Also, consider encrypting local files that contain sensitive data using the security library. Further, reduce the number of permissions that your app requests. By limiting access to sensitive data permissions, you can significantly reduce the risk of exploitation of those permissions, making your mobile app much less vulnerable to attackers.

When it comes to iOS, it provides secure storage APIs which enables mobile app developers to use cryptographic hardware available on every iOS device. Developers can also utilize the iOS data security APIs to work with fine grained access control for user data stored in flash memory.

4. Client Code Security

Code security issues are quite common in mobile apps.

Many of these issues can take extensive time to detect using manual code reviews, you can leverage automated, third-party tools to perform fuzzing or static analysis. These tools can identify injection issues, insecure data storage, weak encryption, and other security issues.

However, automated tools are not sufficient on their own, you still need manual review to find security threats where automation fails.

How Can You Prevent Poor Code Quality Issues?

Maintain consistent secure coding practices that do not lead to vulnerable code. When using buffers, make sure you validate that the length of the incoming buffer data does not exceed the length of the target buffer.

Use automation to detect memory leaks and buffer overflows via third-party static analysis tools. Also ensure that you prioritize solving issues like memory leaks and buffer overflows over other code quality issues as they tend to give rise to more mobile security risks and can be easily exploited.

Use a security company that specializes in static analysis to review your code and identify these security threats and vulnerabilities.

5. Insufficient Authentication and Authorization Controls

Missing or poor authentication schemes allow attackers to anonymously execute functionalities within the mobile application or the backend server used by the app.

Authentication requirements in mobile apps can be different from traditional web applications, in the terms that in mobile apps, users are not needed to be online at all times during their session.

It's possible that mobile apps may have uptime requirements that need offline authentication. This method of offline authenticating a user's identity can pose security risks that developers should consider while implementing authentication schemes.

Similarly, poor authorization can also impact the security of a mobile app depending on the nature of high-privileges breached to attack a mobile user. For instance, if an attacker is able to execute high-privilege actions, such as those of administrators, it may result in data theft, modification, or complete compromise of backend services.

How Can You Prevent Poor Authentication and Authorization?

There are several ways you can implement proper authentication and authorization for increased mobile security:

- Ensure that authentication requests are performed on the server side. Upon successful authentication, the data should be loaded into the mobile device. This will ensure that data is only loaded after successful authentication.
- If client-side data storage is required, use encryption to protect your data and securely derive from the user's credentials.
- To implement strong authorization schemes, verify the roles and permissions of authenticated users using only data contained in backend systems.
- Use multi-factor authentication to validate a user's identity. You can use one-time passwords, security questions, etc.

6. Poor Encryption

Encryption is the process of converting data into an encrypted form that is only readable after it has been translated back using a secret decryption key. If devices and data are not encrypted properly, then attackers can much more readily access the data.

What is the impact of poor encryption?

Simply, poor encryption can lead to data loss and all of the repercussions that follow from that loss of information.

Where do developers screw up encryption?

Many times, developers implement strong encryption, however if the keys are not properly handled, even the best encryption algorithms can fail. For instance, including the keys in insecure databases or files that are easily readable by other users.

This is one of the most common failures we see. Attackers don't try to break the encryption algorithm, that's too hard; they go after the keys. Unfortunately, insecure key management is a huge issue.

Another way mobile developers mishandle encryption is by creating and using custom encryption algorithms or protocols. Often these encryption algorithms are not as secure as other modern algorithms available in the security community. Additionally, using weak or insecure encryption algorithms such as RC2, MD5, MD4, and SHA1 can also lead to attacks.

How Can You Prevent Poor Encryption Algorithms?

Make sure you implement modern encryption algorithms that are accepted as strong by the security community. Use the encryption APIs available within your mobile platform.

Consider implementing encryption in layers so that even if the attacker gets the decryption key to decrypt one layer, there's another two layers of encryption they need to break into. Also, make sure you store encryption keys securely. This is critical.

7. Reverse Engineering

If an attacker can read your code, they can find better ways to attack your application.

Reverse engineering can be used to determine how the app functions on the back end, modify the source code, expose encryption algorithms in place, and more. So the code you developed for your mobile app can be used against you and pose severe security risks.

How Can You Prevent Reverse Engineering?

An effective way of preventing mobile apps from reverse engineering is to limit the capabilities client side and expose more functionality through web services server side. Once functionality is limited to the bare minimum needed, then you obfuscate that code base using commercial obfuscators.

Also, avoid storing API keys in shared resource folders, assets, or anywhere else that's easily accessible by an outsider. Use either public/private key exchange or NDK to protect your mobile app's API key.

What are the main types of cybersecurity threats?

The main types of information security threats are:

- Malware attack
- Social engineering attacks
- Software supply chain attacks
- Advanced persistent threats (APT)
- Distributed denial of service (DDoS)
- Man-in-the-middle attack (MitM)
- Password attacks

We cover each of these threats in more detail below.

Related content: Read our explainer to cyber crime.

1. Malware attack

Attacks use many methods to get malware into a user's device, most often social engineering. Users may be asked to take an action, such as clicking a link or opening an attachment. In other cases, malware uses vulnerabilities in browsers or operating systems to install themselves without the user's knowledge or consent.

Once malware is installed, it can monitor user activities, send confidential data to the attacker, assist the attacker in penetrating other targets within the network, and even cause the user's device to participate in a botnet leveraged by the attacker for malicious intent.

Malware attacks include:

- **Trojan virus** — tricks a user into thinking it is a harmless file. A Trojan can launch an attack on a system and can establish a backdoor, which attackers can use.
- **Ransomware** — prevents access to the data of the victim and threatens to delete or publish it unless a ransom is paid. Learn more in our guide to ransomware prevention.
- **Wiper malware** — intends to destroy data or systems, by overwriting targeted files or destroying an entire file system. Wipers are usually intended to send a political message, or hide hacker activities after data exfiltration.
- **Worms** — this malware is designed to exploit backdoors and vulnerabilities to gain unauthorized access to operating systems. After installation, the worm can perform various attacks, including Distributed Denial of Service (DDoS).
- **Spyware** — this malware enables malicious actors to gain unauthorized access to data, including sensitive information like payment details and credentials. Spyware can affect mobile phones, desktop applications, and desktop browsers.
- **Fileless malware** — this type of malware does not require installing software on the operating system. It makes native files such as PowerShell and WMI editable to enable malicious functions, making them recognized as legitimate and difficult to detect.
- **Application or website manipulation** — OWASP outlines the top 10 application security risks, ranging from broken access controls and security misconfiguration through injection attacks and cryptographic failures. Once the vector is established through service account acquisition, more malware, credential, or APT attacks are launched.

2. Social engineering attacks

Social engineering attacks work by psychologically manipulating users into performing actions desirable to an attacker, or divulging sensitive information.

Social engineering attacks include:

- **Phishing** — attackers send fraudulent correspondence that seems to come from legitimate sources, usually via email. The email may urge the user to perform an important action or click on a link to a malicious website, leading them to hand over sensitive information to the attacker, or expose themselves to malicious downloads. Phishing emails may include an email attachment infected with malware.
- **Spear phishing** — a variant of phishing in which attackers specifically target individuals with security privileges or influence, such as system administrators or senior executives.
- **Malvertising** — online advertising controlled by hackers, which contains malicious code that infects a user's computer when they click, or even just view the ad. Malvertising has been found on many leading online publications.
- **Drive-by downloads** — attackers can hack websites and insert malicious scripts into PHP or HTTP code on a page. When users visit the page, malware is directly installed on their computer; or, the attacker's script redirects users to a malicious site, which performs the download. Drive-by downloads rely on vulnerabilities in browsers or operating systems. Learn more in the guide to drive-by downloads.

- **Scareware security software** — pretends to scan for malware and then regularly shows the user fake warnings and detections. Attackers may ask the user to pay to remove the fake threats from their computer or to register the software. Users who comply transfer their financial details to an attacker.
- **Baiting** — occurs when a threat actor tricks a target into using a malicious device, placing a malware-infected physical device, like a USB, where the target can find it. Once the target inserts the device into their computer, they unintentionally install the malware.
- **Vishing** — voice phishing (vishing) attacks use social engineering techniques to get targets to divulge financial or personal information over the phone.
- **Whaling** — this phishing attack targets high-profile employees (whales), such as the chief executive officer (CEO) or chief financial officer (CFO). The threat actor attempts to trick the target into disclosing confidential information.
- **Pretexting** — occurs when a threat actor lies to the target to gain access to privileged data. A pretexting scam may involve a threat actor pretending to confirm the target's identity by asking for financial or personal data.
- **Scareware** — a threat actor tricks the victim into thinking they inadvertently downloaded illegal content or that their computer is infected with malware. Next, the threat actor offers the victim a solution to fix the fake problem, tricking the victim into downloading and installing malware.
- **Diversion theft** — threat actors use social engineers to trick a courier or delivery company into going to a wrong drop-off or pickup location, intercepting the transaction.
- **Honey trap** — a social engineer assumes a fake identity as an attractive person to interact with a target online. The social engineer fakes an online relationship and gathers sensitive information through this relationship.
- **Tailgating or piggybacking** — occurs when a threat actor enters a secured building by following authorized personnel. Typically, the staff with legitimate access assumes the person behind is allowed entrance, holding the door open for them.
- **Pharming** — an online fraud scheme during which a cybercriminal installs malicious code on a server or computer. The code automatically directs users to a fake website, where users are tricked into providing personal data.

Related content: Read detailed explainer on social engineering techniques.

3. Software supply chain attacks

A software supply chain attack is a cyber attack against an organization that targets weak links in its trusted software update and supply chain. A supply chain is the network of all individuals, organizations, resources, activities, and technologies involved in the creation and sale of a product. A software supply chain attack exploits the trust that organizations have in their third-party vendors, particularly in updates and patching.

This is especially true for network monitoring tools, industrial control systems, "smart" machines, and other network-enabled systems with service accounts. An attack can be made in many places against the vendor continuous integration and continuous delivery (CI/CD) software lifecycle, or even against third-party libraries and components as seen via Apache and Spring.

Types of software supply chain attacks:

- Compromise of software build tools or dev/test infrastructure
- Compromise of devices or accounts owned by privileged third-party vendors
- Malicious apps signed with stolen code signing certificates or developer IDs
- Malicious code deployed on hardware or firmware components
- Malware pre-installed on devices such as cameras, USBs, and mobile phones

4. Advanced persistent threats (APT)

When an individual or group gains unauthorized access to a network and remains undiscovered for an extended period of time, attackers may exfiltrate sensitive data, deliberately avoiding detection by the organization's security staff. APTs require sophisticated attackers and involve major efforts, so they are typically launched against nation states, large corporations, or other highly valuable targets.

Common indicators of an APT presence include:

- **New account creation —** the P in Persistent comes from an attacker creating an identity or credential on the network with elevated privileges.
- **Abnormal activity** — legitimate user accounts typically perform in patterns. Abnormal activity on these accounts can indicate an APT is occurring, including noting a stale account which was created then left unused for a time suddenly being active.
- **Backdoor/trojan horse malware** — extensive use of this method enables APTs to maintain long-term access.
- **Odd database activity** — for example, a sudden increase in database operations with massive amounts of data.
- **Unusual data files** — the presence of these files can indicate data has been bundled into files to assist in an exfiltration process.

Related content: Read detailed explainer on privilege escalation detection.

5. Distributed denial of service (DDoS)

The objective of a denial of service (DoS) attack is to overwhelm the resources of a target system and cause it to stop functioning, denying access to its users. Distributed denial of service (DDoS) is a variant of DoS in which attackers compromise a large number of computers or other devices, and use them in a coordinated attack against the target system.

DDoS attacks are often used in combination with other cyberthreats. These attacks may launch a denial of service to capture the attention of security staff and create confusion, while they carry out more subtle attacks aimed at stealing data or causing other damage.

Methods of DDoS attacks include:

- **Botnets** — systems under hacker control that have been infected with malware. Attackers use these bots to carry out DDoS attacks. Large botnets can include millions of devices and can launch attacks at devastating scale.

- **Smurf attack** — sends Internet Control Message Protocol (ICMP) echo requests to the victim's IP address. The ICMP requests are generated from 'spoofed' IP addresses. Attackers automate this process and perform it at scale to overwhelm a target system.
- **TCP SYN flood attack** — attacks flood the target system with connection requests. When the target system attempts to complete the connection, the attacker's device does not respond, forcing the target system to time out. This quickly fills the connection queue, preventing legitimate users from connecting.

6. Man-in-the-middle attack (MitM)

When users or devices access a remote system over the internet, they assume they are communicating directly with the server of the target system. In a MitM attack, attackers break this assumption, placing themselves in between the user and the target server.

Once the attacker has intercepted communications, they may be able to compromise a user's credentials, steal sensitive data, and return different responses to the user.

MitM attacks include:

- **Session hijacking** — an attacker hijacks a session between a network server and a client. The attacking computer substitutes its IP address for the IP address of the client. The server believes it is corresponding with the client and continues the session.
- **Replay attack** — a cybercriminal eavesdrops on network communication and replays messages at a later time, pretending to be the user. Replay attacks have been largely mitigated by adding timestamps to network communications.
- **IP spoofing** — an attacker convinces a system that it is corresponding with a trusted, known entity. The system thus provides the attacker with access. The attacker forges its packet with the IP source address of a trusted host, rather than its own IP address.
- **Eavesdropping attack** — attackers leverage insecure network communication to access information transmitted between the client and server. These attacks are difficult to detect because network transmissions appear to act normally.
- **Bluetooth attacks** — Because Bluetooth is often open in promiscuous mode, there are many attacks, particularly against phones, that drop contact cards and other malware through open and receiving Bluetooth connections. Usually this compromise of an endpoint is a means to an end, from harvesting credentials to personal information.

7. Password attacks

A hacker can gain access to the password information of an individual by 'sniffing' the connection to the network, using social engineering, guessing, or gaining access to a password database. An attacker can 'guess' a password in a random or systematic way.

Password attacks include:

- **Brute-force password guessing** — an attacker uses software to try many different passwords, in hopes of guessing the correct one. The software can use some logic to trying passwords related to the name of the individual, their job, their family, etc.

- **Dictionary attack** — a dictionary of common passwords is used to gain access to the computer and network of the victim. One method is to copy an encrypted file that has the passwords, apply the same encryption to a dictionary of regularly used passwords, and contrast the findings.
- **Pass-the-hash attack** — an attacker exploits the authentication protocol in a session and captures a password hash (as opposed to the password characters directly) and then passes it through for authentication and lateral access to other networked systems. In these attack types, the threat actor doesn't need to decrypt the hash to obtain a plain text password.
- **Golden ticket attack** — a golden ticket attack starts in the same way as a pass-the-hash attack, where on a Kerberos (Windows AD) system the attacker uses the stolen password hash to access the key distribution center to forge a ticket-granting-ticket (TGT) hash. Mimikatz attacks frequently use this attack vector.

Cyberthreat actors

When you identify a cyberthreat, it's important to understand who the threat actor is, as well as their tactics, techniques, and procedures (TTP). Common sources of cyberthreats include:

- **State-sponsored** — cyberattacks by countries can disrupt communications, military activities, or other services that citizens use daily.
- **Terrorists** — terrorists may attack government or military targets, but at times may also target civilian websites to disrupt and cause lasting damage.
- **Industrial spies** — organized crime and international corporate spies carry out industrial espionage and monetary theft. Their primary motive is financial.
- **Organized crime groups** — criminal groups infiltrate systems for monetary gain. Organized crime groups use phishing, spam, and malware to carry out identity theft and online fraud. There are organized crime groups who exist to sell hacking services to others as well, maintaining even support and services for profiteers and industrial spies alike.
- **Hackers** — there is a large global population of hackers, ranging from beginner "script kiddies" or those leveraging ready-made threat toolkits, to sophisticated operators who can develop new types of threats and avoid organizational defenses.
- **Hacktivists** — hacktivists are hackers who penetrate or disrupt systems for political or ideological reasons rather than financial gain.
- **Malicious insider** — insiders represent a very serious threat, as they have existing access to corporate systems and knowledge of target systems and sensitive data. Insider threats can be devastating and very difficult to detect.
- **Cyber espionage** — is a form of cyberattack that steals classified, or sensitive intellectual data to gain an advantage over a competitive company or government entity.

Related content: Read detailed explainer on security incidents.

Emerging information security threats and challenges in 2023

As technology evolves, so do the threats and issues that security teams face. Below are a few of the top trends and concerns in cybersecurity today.

Use of artificial intelligence (AI) by attackers

AI is a double-edged sword; it is improving security solutions but at the same time is leveraged by attackers to bypass those solutions. Part of the reason for this is the growing accessibility to AI. In the past, developing machine learning models was only possible if you had access to significant budgets and resources. Now, however, models can be developed on personal laptops.

This accessibility makes AI a tool that has expanded from major digital arms races to everyday attacks. While security teams are using AI to try to detect suspicious behavior, criminals are using it to make bots that pass for human users and to dynamically change the characteristics and behaviors of malware.

Cybersecurity skills gap

There is a constant concern over the cybersecurity skills gap. There are simply not enough cybersecurity experts to fill all of the positions needed. As more companies are created and others update their existing security strategies, this number increases.

Modern threats, from cloned identities to deep fake campaigns, are getting harder to detect and stop. The security skills required to combat these threats go far beyond just understanding how to implement tools or configure encryptions. These threats require diverse knowledge of a wide variety of technologies, configurations, and environments. To obtain these skills, organizations must recruit high-level experts or dedicate the resources to training their own.

Vehicle hacking and Internet of Things (IoT) threats

The amount of data contained in a modern vehicle is huge. Even cars that are not autonomous are loaded with a variety of smart sensors. This includes GPS devices, built-in communications platforms, cameras, and AI controllers. Many people's homes, workplaces, and communities are full of similar smart devices. For example, personal assistants embedded in speakers are smart devices.

The data on these devices can provide sensitive information to criminals. This information includes private conversations, sensitive images, tracking information, and access to any accounts used with devices. These devices can be easily leveraged by attackers for blackmail or personal gain. For example, abusing financial information or selling information on the black market.

With vehicles in particular, the threat of personal harm is also very real. When vehicles are partially or entirely controlled by computers, attackers have the opportunity to hack vehicles just like any other device. This could enable them to use vehicles as weapons against others or as a means to harm the driver or passengers.

Threats facing mobile devices

Even if people haven't fully embraced smart technologies, nearly everyone has a mobile device of some sort. Smartphones, laptops, and tablets are common. These devices are often multipurpose, used for both work and personal activities, and users may connect devices to multiple networks throughout the day.

This abundance and widespread use make mobile devices an appealing target for attackers. Targeting is not new but the real challenge comes from security teams not having full control over devices. Bring your own device (BYOD) policies are common but these policies often do not include internal control or management.

Often, security teams are only able to control what happens with these devices within the network perimeter. Devices may be out of date, already infected with malware, or have insufficient protections. The only way security teams may have to block these threats is to refuse connectivity, which isn't practical.

Cloud security threats

With businesses moving to cloud resources daily, many environments are growing more complex. This is particularly true in the case of hybrid and multi-cloud environments, which require extensive monitoring and integration.

With every cloud service and resource that is included in an environment, the number of endpoints and the chances for misconfiguration increase. Additionally, since resources are in the cloud, most, if not all endpoints are Internet-facing, granting access to attackers on a global scale.

To secure these environments, cybersecurity teams need advanced, centralized tooling and often more resources. This includes resources for 24/7 protection and monitoring since resources are running and potentially vulnerable even when the workday is over.

State-sponsored attacks

The Russia-Ukraine war and the new geopolitical situation has raised the stakes of state-sponsored attacks against Western nations and organizations. As more of the world moves to the digital realm, the number of large-scale and state-sponsored attacks are increasing. Networks of hackers can now be leveraged and bought by opposing nation-states and interest groups to cripple governmental and organizational systems.

For some of these attacks, the results are readily apparent. For example, numerous attacks have been identified that involved tampering with elections. Others, however, may go unnoticed, silently gathering sensitive information, such as military strategies or business intelligence. In either case, the resources funding these attacks enables criminals to use advanced and distributed strategies that are difficult to detect and prevent.

Using threat intelligence for threat prevention

Threat intelligence is organized, pre-analyzed information about attacks that may threaten an organization. Threat intelligence helps organizations understand potential or current cyberthreats. The more information security staff have about threat actors, their capabilities, infrastructure, and motives, the better they can defend their organization.

Threat intelligence systems are commonly used in combination with other security tools. When a security system identifies a threat, it can be cross-referenced with threat intelligence data to immediately understand the nature of the threat, its severity, and known methods for mitigating or containing the threat. In many cases, threat intelligence can help automatically block threats — for example, known bad IP addresses can be fed to a firewall, to automatically block traffic from compromised servers.

Threat intelligence is typically provided in the form of feeds. There are free threat intelligence feeds, and others provided by commercial security research bodies. Several vendors provide threat intelligence platforms that come with numerous threat intelligence feeds and help manage threat data and integrate it with other security systems.

Using UEBA and SOAR to mitigate information security threats

User and Entity Behavior Analytics (UEBA) and Security Orchestration, Automation, and Response (SOAR) are technologies that aggregate threat activity data and automate processes related to its identification and analysis, increasing the effectiveness and efficiency of security teams.

UEBA

UEBA uses machine learning to construct a baseline of normal behavior for users or devices/entities within a network, which helps to detect deviations from the baseline behavior. Behavior models and machine learning assign various levels of risk depending on the type of behavior. The risk score of the user or device for an event is determined and is stitched with related events into a timeline to assess if these events pose a threat to an organization. By tying together the behaviors identified as anomalous, analysts can trace all the steps an attacker has taken and thus pin down the threat quickly.

Unlike SIEM, UEBA solutions can detect threat activity over an extended period across multiple organizational systems. UEBA allows security teams to work more efficiently by narrowing down the number of threats they need to investigate, generating alerts, and providing information on breaches that occur.

UEBA can help identify a variety of insider threats, data exfiltration, and lateral movement:

- **Malicious insiders** — by determining a baseline of behavior for users, UEBA can detect abnormal activity and assist in interpreting intent. For example, a user might have genuine access privileges but not need to access sensitive data at a given time or place.
- **Compromised insiders** — users with access privileges can become compromised through malware or phishing attempts, allowing their credentials to be used to initiate an attack. Attackers often change credentials, IP addresses, or devices once in the system. By comparing device and user behavior to baselines, UEBA can identify these attacks in a way that traditional security tools like firewalls and antivirus cannot.
- **Data exfiltration** — tools like data loss prevention (DLP) that use machine learning, dictionary models, and behavior models to gather all evidence related to sensitive data exfiltration can quickly investigate and alert on anomalous activity. This includes data uploads, remote logins, database activities, cloud access, and file share access.
- **Lateral movement** — attackers often traverse a network using a variety of IP addresses, credentials, and machines in search of key assets and data. UEBA tools detect this movement by enriching data with context which allows them to distinguish between servers, users, service accounts, HR personnel, finance staff, and executives and determine if they are behaving suspiciously.

UEBA can also prioritize high-risk events and monitor large numbers of devices:

- **Incident prioritization** — can help determine which incidents are particularly suspicious or dangerous by evaluating them in the context of organizational structure and potential for damage.
- **Monitoring large numbers of devices** — can be used even when a baseline for normal behavior has not yet been developed, using heuristic methods like supervised machine learning, Bayesian networks, unsupervised learning, reinforced machine learning, and deep learning.

SOAR

SOAR tools collect data for security investigations from multiple sources, facilitate incident analysis and triage with machine assistance, define and direct threat response workflow, and enable automated incident response.

Security teams can integrate SOAR tools with other security solutions to respond to incidents more effectively. They can use these solutions through a generic interface, eliminating the need for expert analysts specializing in each system. SOAR allows security teams to automate enforcement and status tracking or auditing tasks based on decision-making workflows as assigned.

SOAR tools simplify incident management and collaboration by automatically generating incidents based on guidelines and including relevant contextual information. They provide a timeline of events for analysis and allow for the addition of evidence as it is found as well as assisting case management by accepting documentation of threats, responses, and outcomes. A comprehensive UEBA solution goes hand-in-hand with SOAR as an effective investigation tool, where the ultimate goal of SOC analysts is to reduce the time needed to detect threats and respond to incidents.

Finally, SOAR tools aid security teams in effectively responding to security incidents by proactively enforcing processes to gather comprehensive evidence, seamlessly integrating with various third-party services and security vendors, and associating a timeline of events to pinpoint anomalous behavior.

What is a Data Breach? Definition, Consequences & Best Practices

A data breach is akin to a home invasion. During a home invasion, a burglar would break into your home during the silence of the night and steal your valuables without you knowing it until it is too late. This is exactly what a data breach feels like to organizations.

Data breaches are a common occurrence across the globe. In fact, hundreds of thousands of businesses experience a certain level of data breach every year. To put things into perspective, IBM's Cost of Data Breach 2022 report revealed that 83% of the surveyed organizations had experienced more than one data breach. No organization that collects personal or sensitive data is safe against the threat of unauthorized or illegal access to or loss or destruction of data. However, what they can do is take appropriate steps to prevent data breaches to some extent or minimize their impact.

But what exactly is a data breach? How does it occur? What are the consequences that organizations have to shoulder due to a breach? And, more importantly, what organizations can do to prevent or mitigate data breaches. If you wish to find answers to all these questions, we suggest you continue reading.

What Is a Data Breach?

Data breaches are security incidents that lead to loss, alteration, illegal or unauthorized destruction or unauthorized disclosure of, or unauthorized access to personal data that is processed, stored, or transmitted by an organization.

A cyber threat actor, an individual or a group, uses various tools and methods to execute a data breach. For instance, a threat actor may breach a corporate network through malware, also called malicious software. Or, they could disguise themselves as a corporate employee and send phishing emails containing malicious links to existing employees.

Often, the inherent vulnerabilities in the system or misconfigured settings give cyber attackers a way into the corporate network, such as a misconfigured cloud service or application that may have a default password or an unprotected publicly accessible storage bucket.

Data breaches have wide-reaching consequences that can greatly impact an organization's financial and reputational position. Therefore, preventing and responding to such cyber threats has become ever more critical.

For starters, we’ve witnessed the non-stop proliferation of data due to the increased number of devices, systems, and applications. In fact, we are leveraging data to generate more data. The abundance of personal data across different systems and devices creates more opportunities for attackers to gain unauthorized access to personal data. Therefore, it is important for organizations to primarily curb the occurrence of such incidents and mitigate their effects where necessary.

Secondly, and most importantly, due to the growing instances of data breaches and other threats, international regulatory authorities have enacted data protection and privacy laws. These laws give more control to individuals over their data and place greater responsibilities upon businesses in relation to data protection, integrity, accountability, and privacy. Hence, in the current era, a data breach means not only heavy loss of data but also huge regulatory fines.

Types of Data That Threat Actors Breach

There are different types of data that cyber attackers attempt to access or steal during a data breach, such as:

Personally Identifiable Information

Personally identifiable information (PII) is any information that can be used (often with another PII) to identify or distinguish between two individuals. This type of information includes an individual’s name, email address, phone number, date of birth, etc. Apart from that, PII also has a sub-category, i.e., sensitive personal information (SPI). As the name suggests, it includes any piece of information whose exposure to unauthorized persons can potentially harm the concerned individual. This type of data includes social security numbers, driver’s license numbers, fingerprint data, and data relating to one’s ethnic origin, religious affiliation, sexual orientation, etc.

Learn More About Personally Identifiable Information (PII)

Health Information

Health information usually includes any category of medical data of an individual, such as medical records, imaging data (CTR or MRI), mental health data, etc. The Health Insurance Portability and Accountability Act (HIPAA) in the United States defines different types of healthcare data as Personal Health Information (PHI).

Financial Information

As the name suggests, financial information includes data related to financial accounts, transactions, or assets of an individual or an organization.

Payment Card Industry (PCI) Information

Payment card data differs from financial information in that it is specific to payment cards, such as credit card data or debit card data. This type of data includes the card number, PIN, or CVV of an

individual's payment card. The Payment Card Industry Data Security Standard (PCI DSS) generally regulates payment card data.

Learn More About the Payment Card Industry Data Security Standard (PCI DSS)

Military Information

This type of data includes sensitive or confidential data that is related to a government or its military bodies. This type of information includes military intelligence data, weapon patent data, etc. In the United States, military data is regulated by the International Traffic in Arms Regulations (ITAR).

Learn More About ITAR Compliance

Destructive Fallouts of Data Breaches

A data breach can happen even due to minor negligence - however, it can certainly result in a great deal of chaos. Equifax's 2017 data breach is the primary example of a huge-scale data breach that occurred due to a system vulnerability that the organization could not fix in time. The resulting breach gave threat actors clear access to the data of over 130 million Americans, 15 million Britishers, and 19,000 Canadians.

When a breach occurs, it is not just the organization that suffers the consequences but also the affected individuals who are exposed to harm.

Following are some of the consequences of data breaches.

Financial Loss

Data breaches have serious implications, starting with heavy financial losses. According to the IBM Cost of Data Breach 2022 report, the average global cost of a data breach in 2022 was $4.35 million, while the average cost of a breach in the US alone in the same year was $9.44 million. Different factors impact the total cost of a breach, such as the cost of detection and escalation, breach notifications to the impacted individuals and relevant regulatory authorities, the post-breach responses and mitigation measures, and lost business opportunities.

Reputational Damage

A monetary loss is easier to recover than a loss of trust. Data breaches can negatively impact an organization's reputation, which takes years to build. In fact, it can have a long-lasting impact on an organization's ability to re-establish itself, gain positive reviews, or earn the trust of consumers or the general public. Moreover, negative media coverage also adds more fuel to the fire, making it more challenging to retain customers or even business partners.

Regulatory Fines

Data protection laws are very strict when it comes to security breaches. Almost every data protection law requires businesses to have optimal administrative and technical security controls in place for protecting data against unauthorized access, leak, destruction, loss of data, etc. Apart from that, data privacy laws also provide notification requirements in the event of a breach.

For instance, articles 33 and 34 of the European Union's General Data Protection Regulation (GDPR) outline that a personal data breach, which would likely put the rights and freedom of data subjects at risk, must be notified. In this regard, businesses must notify the relevant supervisory

authority and the impacted individuals where the breach will likely result in a 'high' risk to their rights and freedoms.

Failure to notify the breach to the concerned authorities and individuals in a timely manner can result in huge fines and penalties.

Lost Business Opportunities

A data breach may make an organization lose its ability to seek new business opportunities or bid on new contracts, as any potential business partners would only seek businesses with a good market reputation and are more secure.

Top Threat Vectors That Lead to Data Breaches

There is a myriad of tactics in cybercriminals' arsenal that they are not afraid to use to make their data breach attempt successful. Let's take a quick look at some of the most common yet effective ways in which threat actors execute data breaches.

Malware

Malware includes trojans, keyloggers, ransomware, and other types of malicious software that cybercriminals may use to steal data. For instance, a cybercriminal might disguise a malicious URL as a lottery or giveaway coupon to bait unsuspecting users.

Insider Threats

According to a recent insider threats report, 74% of organizations believe that insider attacks have become more frequent over the years. The report goes on to cite that 60% of organizations have experienced at least one insider attack, while 25% have suffered multiple attacks. An insider attack is any data breach that occurs when a person within an organization, intentionally or unintentionally, gains unauthorized or illegal access to sensitive, confidential, or proprietary information.

Learn About Six Different Insider Threats

Security Misconfigurations

It is a pretty common type of cybersecurity vulnerability where security settings or configurations are not properly implemented, especially in cloud offerings. In a multi-cloud environment, businesses may have multiple cloud service providers. Each service has a distinct set of configurations. Due to the complex infrastructure of a multi-cloud environment and often due to a lack of understanding of different settings, some key security misconfigurations may be overlooked. This ultimately leads to a security breach. A misconfiguration may include publicly accessible cloud storage, default passwords, opened internal or external ports, etc.

Social Engineering Attacks

Since humans are the weakest link in cybersecurity defenses, data breaches constituting social engineering attacks are often successful. These attacks are geared towards humans and are meant to manipulate them into taking certain actions, such as clicking a malicious link with malware or sharing sensitive information. There are many ways to conduct a social engineering attack, such as phishing, tailgating, spear phishing, etc.

Understanding Data Breach Cycle

Every cyber attacker uses a distinct tool or method to steal data into a target's network or system. However, on a broader level, the process of the attack remains the same.

Reconnaissance

The first is the research or observation phase. In this phase, the cyber attacker carefully and methodically picks the target, making sure that it is easier to breach or reach the target. The perpetrator tries to find the target's weaknesses to determine what method would best fit the breach attempt. This phase involves hours and days of observation, and it often brings forth expected results.

Execution

The next is the intrusion phase, where the perpetrator tries to make the initial contact. Since the attacker has the requisite understanding of the target, it is easier for them to execute the breach attempt. If it is a system or a network, the cyber attacker may look for vulnerabilities, open ports, or any misconfigured system. If it is an individual or an employee, the perpetrator would first stalk the individual on their social media profiles to learn more about them in order to be able to conduct a targeted social engineering attack.

Exfiltration

Once the attacker is successful in the breach, they will try to extract and transfer the sensitive data outside the corporate network. In this phase, the attacker can do a number of things with the breached data or the targeted system. For instance, the attacker may sell data on the dark web or use it to cause damage to concerned individuals, such as through blackmail or harassment, or the attacker may use a compromised system for distributed denial of service (DDoS) attacks.

Best Practices to Prevent & Mitigate Data Breaches

Here are some of the best practices you can consider to prevent or mitigate data breaches.

Data Risk Assessment

As we have learned so far, data breaches can be costly and chaotic for a company's reputation. Therefore, it is crucial for businesses to reinforce cyber defense mechanisms around their sensitive networks, data systems, and the sensitive data itself.

To kick it off, assess the current state of your organization's sensitive data assets and security. Review your organization's data landscape and see what sensitive data you have and what regulatory security guidelines apply to it. Moreover, assess the current security status of the sensitive data to pinpoint security gaps and reasonably foreseeable threats that may exploit the business mechanisms and vulnerabilities of the systems.

Security Configuration Management

One of the most common reasons for cloud security breaches is a misconfiguration. These issues or security lapses arise when the cloud services or applications' security settings are not implemented or configured with errors. For instance, the cloud service may have inadequate user access controls, or its password is set by default, or the sensitive storage bucket is left public by default.

Conduct a thorough analysis to discover and identify security misconfigurations across your cloud data assets. Remediate the configuration errors automatically or manually, as feasible. You would

require an automated multi-cloud data asset discovery mechanism to save time and reduce human errors.

User Access Controls

Businesses cannot fight off data breaches effectively if they do not have proper insights into who is accessing sensitive data, from where, how often, and what privileged access rights they have. It is challenging to address data breaches or unauthorized access instances without having those insights or implementing a least privilege principle. Therefore, gathering these much-needed sensitive data access insights is essential to implement appropriate access policies.

Sensitive Data Protection Mechanisms

Businesses should review the existing security mechanisms and policies around their sensitive data and systems regularly while keeping regulatory compliance in view. This way, businesses can better determine which data needs to be encrypted, made publicly available, truncated, tokenized, or masked.

Timely Security Patching

Most data breaches also occur due to security vulnerabilities in data systems. Outdated systems tend to have vulnerabilities that, if not patched in time, can definitely attract a cyber attack, such as in the case of Equifax. Therefore, create a regular review policy to periodically analyze the system's security status and update any security patches in a timely manner.

Employee Training

Even if an organization reinforces its entire corporate infrastructure against security incidents to the best of its ability, it can still face data breaches due to human error or negligence. Human negligence has wide-ranging implications on an organization's cyber security and, ultimately, its reputation. An employee can easily fall victim to a phishing or social engineering attack with just a click on a link.

The only way to prevent cyber mishaps is by training your employees and giving them data security and privacy awareness. Create training programs for employees and make it a part of the initial orientation so they maintain proper cyber hygiene. Moreover, at the end of employment, appropriate measures should be taken, such as the return of physical assets or removal of the organization's data, access rights, and privileges from the employee's personal device, to ensure the security and integrity of the organization's data and systems.

Data storage security involves protecting storage resources and the data stored on them – both on-premises and in external data centers and the cloud – from accidental or deliberate damage or destruction and from unauthorized users and uses. It's an area that is of critical importance to enterprises because the majority of data breaches are ultimately caused by a failure in data storage security.

Secure Data Storage:

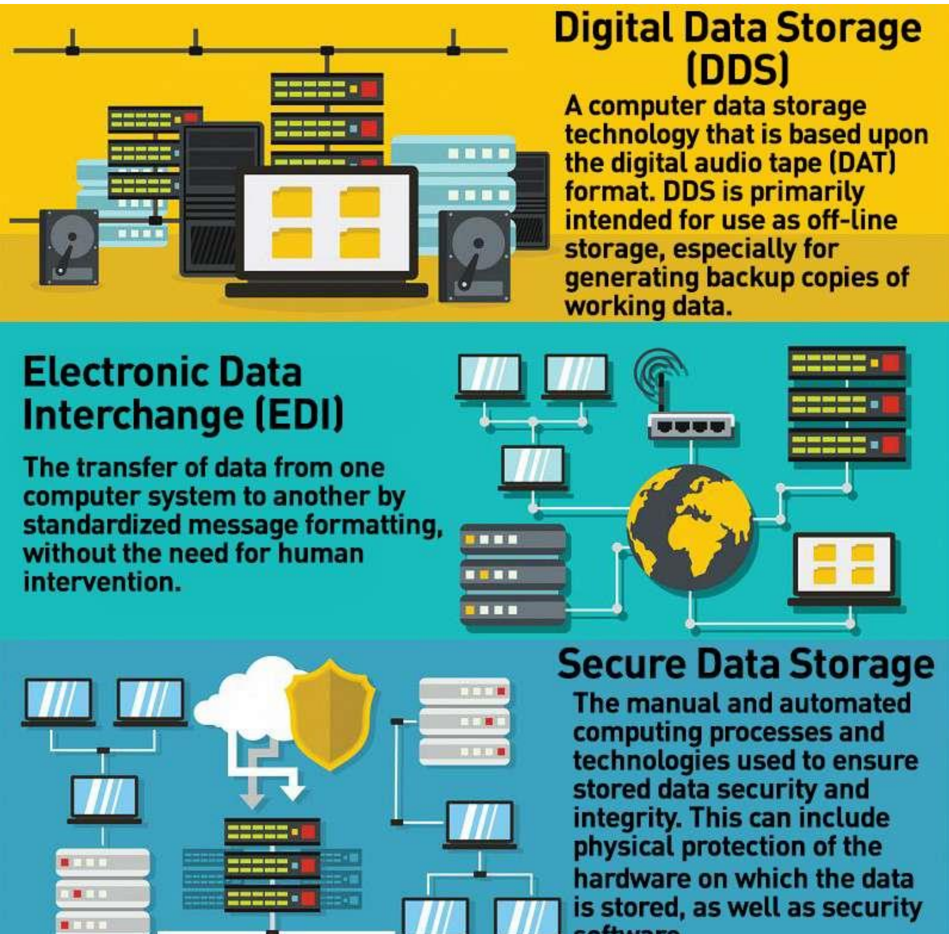
Secure Data Storage collectively refers to the manual and automated computing processes and technologies used to ensure stored data security and integrity. This can include physical protection of the hardware on which the data is stored, as well as security software.

Secure data storage applies to data at rest stored in computer/server hard disks, portable devices – like external hard drives or USB drives – as well as online/cloud, network-based storage area network (SAN) or network attached storage (NAS) systems.

How Secure Data Storage is Achieved:

- **Data encryption**
- **Access control mechanism at each data storage device/software**
- **Protection against viruses, worms and other data corruption threats**
- **Physical/manned storage device and infrastructure security**
- **Enforcement and implementation of layered/tiered storage security architecture**

Secure data storage is essential for organizations which deal with sensitive data, both in order to avoid data theft, as well as to ensure uninterrupted operations.



Data Security vs Data Protection:

Storage security and data security are closely related to data protection. Data security primarily involves keeping private information out of the hands of anyone not authorized to see it. It also includes protecting data from other types of attacks, such as ransomware that prevents access to information or attacks that alter data, making it unreliable.

Data protection is more about making sure data remains available after less nefarious incidents, like system or component failures or even natural disasters.

But the two overlap in their shared need to ensure the reliability and availability of information, as well as in the need to recover from any incidents that might threaten an organization's data.

Storage professionals often find themselves dealing with data security and data protection issues at the same time, and some of the same best practices can help address both concerns.

Threats to Data Security:

Before looking at how to implement data storage security, it is important to understand the types of threats organizations face.

Threat agents can be divided into two categories: external and internal.

External threat agents include:

- **Nation states**
- **Terrorists**
- **Hackers, cybercriminals, organized crime groups**
- **Competitors carrying out "industrial espionage"**

Internal threat agents include:

- **Malicious insiders**
- **Poorly trained or careless staff**
- **Disgruntled employees**

Other threats include:

- **Fire, flooding and other natural disasters**
- **Power outages**

Storage Vulnerabilities:

Another huge driver of interest in data storage security is the vulnerabilities inherent in storage systems. They include the following:

- **Lack of encryption — While some high-end NAS and SAN devices include automatic encryption, plenty of products on the market do not include these capabilities. That means organizations need to install separate software or an encryption appliance in order to make sure that their data is encrypted.**
- **Cloud storage — A growing number of enterprises are choosing to store some or all of their data in the cloud. Although some argue that cloud storage is more secure than on-premises storage, the cloud adds complexity to storage environments and often requires storage personnel to learn new tools and implement new procedures in order to ensure that data is adequately secured.**
- **Incomplete data destruction — When data is deleted from a hard drive or other storage media, it may leave behind traces that could allow unauthorized individuals to recover that information. It's up to storage administrators and managers to ensure that any data erased from storage is overwritten so that it cannot be recovered.**

- **Lack of physical security — Some organizations don't pay enough attention to the physical security of their storage devices. In some cases they fail to consider that an insider, like an employee or a member of a cleaning crew, might be able to access physical storage devices and extract data, bypassing all the carefully planned network-based security measures.**

Data Storage Security Principles:

At the highest level, data storage security seeks to ensure "CIA" – confidentiality, integrity, and availability.

- **Confidentiality: Keeping data confidential by ensuring that it cannot be accessed either over a network or locally by unauthorized people is a key storage security principle for preventing data breaches.**
- **Integrity: Data integrity in the context of data storage security means ensuring that the data cannot be tampered with or changed.**
- **Availability: In the context of data storage security, availability means minimizing the risk that storage resources are destroyed or made inaccessible either deliberately – say during a DDoS attack – or accidentally, due to a natural disaster, power failure, or mechanical breakdown.**

Data Security Best Practices:

In order to respond to these technology trends and deal with the inherent security vulnerabilities in their storage systems, experts recommend that organizations implement the following data security best practices:

Data Security Protection Layers, Risk Mitigation and Controls

Protection Layer	Risks Mitigated	Data Security Contr
Cloud SaaS/Service	**Lower levels (minus IaaS) +:** • Cloud/SaaS privileged access • Cloud environment compromise or failure • Remote legal access	**Encryption at Cloud SaaS/Servi** • BYOK to cloud • Cloud encryption key management
Application/ Database	**Lower levels +:** • Inside DB: DB Admins, DB Users • Inside App: Application Admins and Users • External compromise of these accounts	• Application encryption • Database column encryptio • TDE Key Management • Tokenization • Data masking • Database Access Monitoring
File System / Volume	**Lower level + system level threats:** • External threats/breach of system/ privileged accounts • Enterprise Privileged User/Insider • Cloud IaaS privileged user/breach	• File level encryption and access controls • Privileged user managemen
Disk or other Media	**Physical device level:** • Loss, theft or improper retirement of physical media	• Full disk encryption • SAN/Storage array/other KMIP key management • Tape encryption • VM encryption

1. **Data storage security policies — Enterprises should have written policies specifying the appropriate levels of security for the different types of data that it has. Obviously, public data needs far less security than restricted or confidential data, and the organization needs to have security models, procedures and tools in place to apply appropriate protections. The policies should also include details on the security measures that should be deployed on the storage devices used by the organization.**
2. **Access control — Role-based access control is a must-have for a secure data storage system, and in some cases, multi-factor authentication may be appropriate. Administrators should also be sure to change any default passwords on their storage devices and to enforce the use of strong passwords by users.**
3. **Encryption — Data should be encrypted both while in transit and at rest in the storage systems. Storage administrators also need to have a secure key management systems for tracking their encryption keys.**
4. **Data loss prevention — Many experts say that encryption alone is not enough to provide full data security. They recommend that organizations also deploy data loss prevention (DLP) solutions that can help find and stop any attacks in progress.**
5. **Strong network security — Storage systems don't exist in a vacuum; they should be surrounded by strong network security systems, such as firewalls, anti-malware protection, security gateways, intrusion detection systems and possibly advanced analytics and machine learning based security solutions. These measures should prevent most cyberattacks from ever gaining access to the storage devices.**

6. **Strong endpoint security — Similarly, organizations also need to make sure that they have appropriate security measures in place on the PCs, smartphones and other devices that will be accessing the stored data. These endpoints, particularly mobile devices, can otherwise be a weak point in an organization's cyberdefenses.**
7. **Redundancy — Redundant storage, including RAID technology, not only helps to improve availability and performance, in some cases, it can also help organizations mitigate security incidents.**
8. **Backup and recovery — Some successful malware or ransomware attacks compromise corporate networks so completely that the only way to recover is to restore from backups. Storage managers need to make sure that their backup systems and processes are adequate for these type of events, as well as for disaster recovery purposes. In addition, they need to make sure that backup systems have the same level of data security in place as primary systems.**

- Data encryption is a way of translating data from plaintext (unencrypted) to ciphertext (encrypted). Users can access encrypted data with an encryption key and decrypted data with a decryption key.

  Benefits of data encryption

- Effective data encryption
- Related Solutions
- Resources

Protecting your data

There are massive amounts of sensitive information managed and stored online in the cloud or on connected servers. Encryption uses cybersecurity to defend against brute-force and cyber-attacks, including malware and ransomware. Data encryption works by securing transmitted digital data on the cloud and computer systems. There are two kinds of digital data, transmitted data or in-flight data and stored digital data or data at rest.

Modern encryption algorithms have replaced the outdated Data Encryption Standard to protect data. These algorithms guard information and fuel security initiatives including integrity, authentication, and non-repudiation. The algorithms first authenticate a message to verify the origin. Next. they check the integrity to verify that contents have remained unchanged. Finally, the non-repudiation initiative stops sends from denying legitimate activity.

Types of data encryption: asymmetric vs symmetric

There are several different encryption methods, each developed with different security and security needs in mind. The two main types of data encryption are asymmetric encryption and symmetric encryption.

Asymmetric encryption methods:

Asymmetric encryption, also known as Public-Key Cryptography, encrypts and decrypts the data using two separate cryptographic asymmetric keys. These two keys are known as a "public key" and a "private key".

Common asymmetric encryption methods:

- RSA: RSA, named after computer scientists Ron Rivest, Adi Shamir, and Leonard Adleman, is a popular algorithm used to encrypt data with a public key and decrypt with a private key for secure data transmission.
- Public key infrastructure (PKI): PKI is a way of governing encryption keys through the issuance and management of digital certificates.

Symmetric encryption methods:

Symmetric encryption is a type of encryption where only one secret symmetric key is used to encrypt the plaintext and decrypt the ciphertext.

Common symmetric encryption methods:

- Data Encryption Standards (DES): DES is a low-level encryption block cipher algorithm that converts plain text in blocks of 64 bits and converts them to ciphertext using keys of 48 bits.
- Triple DES: Triple DES runs DES encryption three different times by encrypting, decrypting, and then encrypting data again.
- Advanced Encryption Standard (AES): AES is often referred to as the gold standard for data encryption and is used worldwide as the U.S. government standard.
- Twofish: Twofish is considered one of the fastest encryption algorithms and is free to use.

Explore asymmetric vs symmetric encryption

Benefits of data encryption

With more and more organizations moving to hybrid and multicloud environments, concerns are growing about public cloud security and protecting data across complex environments. Enterprise-wide data encryption and encryption key management can help protect data on-premises and in the cloud.

Cloud service providers (CSPs) may be responsible for the security of the cloud, but customers are responsible for security in the cloud, especially the security of any data. An organization's sensitive data must be protected, while allowing authorized users to perform their job functions. This protection should not only encrypt data, but also provide robust encryption key management, access control and audit logging capabilities.

Robust data encryption and key management solutions should offer:

- A centralized management console for data encryption and encryption key policies and configurations
- Encryption at the file, database and application levels for on-premise and cloud data
- Role and group-based access controls and audit logging to help address compliance
- Automated key lifecycle processes for on-premise and cloud encryption keys

Learn about future-proofing data - This link opens in a new tab

Effective data encryption

New homomorphic encryption toolkit

IBM© synthesized 11 years of cryptography research into a streamlined fully homomorphic encryption (FHE) toolkit for Mac OS and iOS.

See what's next

IBM Blockchain Platform 2.5

The newly launched multi-party network called IBM Blockchain Platform 2.5 includes the latest innovations to improve the IBM Blockchain Platform.

Discover the IBM Platform

IBM Z Enhancements

IBM Fibre Channel Endpoint Security for IBM z15™ helps protect data in flight with pervasive encryption and without the costly application changes.

Read the blog

Related Solutions

Data encryption solutions

Protect enterprise data and address regulatory compliance with data-centric security solutions and services

Explore data encryption solutions

Pervasive encryption solutions

Encrypting data with IBM encryption technology will ensure your data is protected, even in the event of a data breach.

Explore pervasive encryption

Protect sensitive data

IBM Data Privacy Passports protects sensitive data and maintains privacy by policy as the data moves from its source across hybrid multiclouds.

Explore IBM Data Privacy Passports

Data encryption and cryptographic services

IBM Cryptographic Services protects and retains full control of your sensitive data.

Explore IBM Cryptographic Services

Enterprise key management

IBM Enterprise Key Management Foundation (EKMF) is a highly secure and flexible key management system for enterprise.

Explore IBM Enterprise Key Management

Data encryption software

Protect your file and database data from misuse with IBM Security Guardium Data Encryption, an integrated suite of products built on a common infrastructure.

Explore Guardium Data Encryption software

Data security solutions

Protect your data, meet privacy regulations, and simplify operational complexity with IBM Cloud Pak for security.

Explore data security solutions

Flash storage solutions

Simplify data and infrastructure management with the unified IBM FlashSystem® platform family, which streamlines administration and operational complexity across on-premises, hybrid cloud, virtualized and containerized environments.

Explore flash storage solutions

Data Encryption Methods & Types: Beginner's Guide To Encryption

Types of encryption

Due to multiple types of data and various security use cases, many different methods of encryption exist. We can broadly group data encryption methods into two categories: symmetric and asymmetric data encryption.

Symmetric encryption

When using symmetrical encryption methods, a single secret key is used to encrypt plaintext and decrypt ciphertext. Both the sender and receiver have private access to the key, which can only be used by authorized recipients. Symmetric encryption is also known as **private key cryptography**.

Some common symmetric encryption algorithms include:

- Advanced Encryption Standard (AES)
- Data Encryption Standard (DES)
- Triple DES (TDES)
- Twofish

And we'll look at each of these shortly.

Asymmetric encryption

This method of encryption is known as **public key cryptography**. In asymmetric encryption, two keys are used: a public key and a private key. Separate keys are used for both the encryption and decryption processes:

- The **public key**, as the name suggests, is either publicly available or shared with authorized recipients.

- The corresponding **private key** is required to access data encrypted by the public key. The same public key will *not* work to decrypt the data in this technique.

Asymmetric encryption offers another level of security to the data which makes online transfers safer. Common asymmetric encryption methods include Rivest Shamir Adleman (RSA) and Elliptic Curve Cryptography (ECC)

Comparing symmetric vs asymmetric encryption

Aside from the fact both techniques use different key combinations, there are other differences between symmetric and asymmetric encryption.

- **Asymmetric encryption** is a newer method that eliminates the need to share a private key with the receiver. Importantly, however, this approach takes longer in practice than symmetric encryption.
- **Symmetric encryption techniques** are best suited to larger data sets but use smaller ciphertexts in comparison to the original plaintext file. (The opposite is true of asymmetric encryption.)

Within the categories of asymmetric and symmetric encryption methods are unique algorithms that all use different tactics to conceal sensitive data. We'll explore these below.

Quick note: how hashing works

Hashing is a technique that uses a mathematical function to convert inputs of any size (files, messages, etc.) into a fixed length value.

Many people mistake hashing for being an encryption technique, but this is an important distinction to make. In hashing, there is no key, which means you cannot ensure complete privacy. Additionally, a hash can be recreated.

Hashing is typically used alongside cryptography, as a method of storing and retrieving data. It is most commonly used for:

- Document verification
- Digital signatures
- Integrity controls

Common data encryption algorithms and techniques

Encryption methods vary based on a number of factors, including:

- The type of keys used
- Encryption key length
- The size of the encrypted data blocks

Now let's look at seven common methods of encryption that you can use to safeguard sensitive data for your business.

1. Advanced Encryption Standard (AES)

The Advanced Encryption Standard is a symmetric encryption algorithm that is the most frequently used method of data encryption globally. Often referred to as the gold standard for data encryption, AES is used by many government bodies worldwide, including in the U.S.

AES encrypts 128-bit data blocks at a time and can be used for:

- File and application encryption
- Wifi security
- VPNs
- SSL/TLS protocols

(Check out our AES technical explainer.)

2. Triple Data Encryption Standard (TDES)

The Triple Data Encryption Standard, sometimes shortened to Triple DES or 3DES, is a symmetric encryption method that uses a 56-bit key to encrypt data blocks. It is a more advanced, more secure version of the Data Encryption Standard (DES) algorithm. As its name indicates, TDES applies DES to each block of data three times.

Utilized by applications like Firefox and Microsoft Office, TDES encrypts things like:

- ATM pins
- UNIX passwords
- Other payment systems

Today, some industry leaders indicate that TDES is being transitioned out of certain tools and products. The overall security of AES remains superior to TDES, per NIST.

3. Rivest Shamir Adleman (RSA)

The Rivest Shamir Adleman algorithm is an asymmetric form of encryption. Used to encrypt data from one point of communication to another (across the internet), it depends on the prime factorization of two large randomized prime numbers. This results in the creation of another large prime number — the message can be only decoded by someone with knowledge of these numbers.

It is extremely difficult for a hacker to work out the original prime numbers, so this encryption technique is a viable way to secure confidential data within an organization. There are some limitations to this method, primarily that it slows when encrypting larger volumes of data. Typically, though, RSA is used for:

- Smaller-scale documentation
- Files
- Messaging
- Payments

4. Blowfish

This symmetric encryption algorithm was originally designed to replace the Data Encryption Standard (DES). The Blowfish encryption technique uses 64-bit block sizes and encrypts them individually.

This data encryption method is known for its flexibility, speed and resilience. It's also widely available as it's in the public domain, which adds to the appeal. Blowfish is commonly used for securing:

- E-commerce platforms
- Password management systems
- Email data encryption tools

5. Twofish

The next generation version of Blowfish is Twofish, a symmetric encryption technique that encrypts 128-bit data blocks. Twofish utilizes a more complicated key schedule, encrypting data in 16 rounds no matter the size of the encryption key. It's also publicly available like its predecessor Blowfish, but it's a lot faster and can be applied to both hardware and software.

Twofish is most frequently used for file and folder encryption.

6. Format-Preserving Encryption (FPE)

Another symmetric encryption algorithm is FPE: Format-Preserving Encryption. As the name suggests, this algorithm keeps the format (and length) of your data during encryption. An example would be a phone number. If the original number is 012-345-6789, then the ciphertext would retain the format but use a different, randomized set of numbers e.g. 313-429-5072.

FPE can be used to secure cloud management software and tools. Trusted cloud platforms like Google Cloud and AWS use this method for cloud data encryption.

7. Elliptic Curve Cryptography (ECC)

The ECC encryption algorithm is a relatively new asymmetric encryption method. It uses a curve diagram to represent points that solve a mathematical equation, making it highly complex. The shorter keys make it faster and stronger than RSA encryption. ECC can be used for:

- Web communications security (SSL/TLS protocols)
- One-way email encryption
- Digital signatures in cryptocurrencies like Bitcoin or NFTs

Challenges to data encryption methods

Despite their obvious strengths, there are some drawbacks to encryption methods. Fortunately, careful adoption of best practices, which we'll cover below, help overcome and mitigate these concerns.

Key management

One of the major challenges to data encryption techniques within an organization is key management. Any keys required for decryption must be stored somewhere. Unfortunately, this

location is often less secure than people think. Hackers have a particular knack for uncovering the whereabouts of key information, posing a huge threat to enterprise and network security.

Key management also adds another layer of complexity where backup and restoration are concerned. When disaster strikes, the key retrieval and backup process can prolong your business's recovery operation.

(Understand how vulnerabilities and threats contribute to overall risk.)

Brute force attacks

Vulnerability to brute force attacks is a less common — though serious — threat to encryption. A brute force attack is the formal name of a hacker's attempts to *guess* the decryption key. Modern computer systems can generate millions or billions of possible combinations, which is why the more complex any encryption key, the better.

Today's encryption algorithms, when used in combination with strong passwords, are usually resistant to these types of attacks. However, computing technology continues to evolve, continuing to pose an existential threat to data encryption techniques in future.

Best practices for a data encryption strategy

Data encryption is one of the best ways to safeguard your organization's data. Still, like most things, successful encryption comes down to the strategy and execution. In this section, we'll look at some best practices to ensure your data encryption algorithms and techniques are as effective as possible.

1. Define security requirements

Scoping out the general security landscape of your organization is an important first step in any encryption strategy. Encryption systems vary in strength and processing capabilities, so it's important to assess your current security needs before buying into a solution.

To evaluate your security posture, you can…

- Conduct a threat assessment to uncover any system vulnerabilities.
- Speak to teams and stakeholders to learn of any business decisions, existing situations and even compliance regulations that could affect your strategy.
- Review prescriptive materials including well-established cybersecurity frameworks.

2. Classify your data

Building on the first step, you're ready to better understand the types of data you store and send. This includes anything from customer information to financial data and company account details and even your proprietary information that your business relies on. You can then classify each type of data by:

- How sensitive it is
- Whether and how it's regulated
- How often it's used and called upon

(Understand data structures & compare data lakes and data warehouses.)

3. Choose the right encryption solution

Once you've identified your data priorities and security requirements, you can look for data encryption tools to fit your needs. You'll likely need to install a range of encryption algorithms and techniques to protect different forms of data across your databases, files and applications. The best data encryption solutions are able to offer:

- Encryption at multiple levels (application, database and file) for data on-premises and in the cloud
- A centralized management dashboard for data encryption, encryption key policies and configurations
- An automated lifecycle process for encryption keys (both on-premises and cloud-based)
- Audit logging and shared group and role-based access controls (RBAC) to help address compliance

1 Introduction to the Crypto Library

Cryptography is not security. It is a tool that may be used in some cases to achieve security goals.

This library is not a turn-key solution to security. It is a library of low-level cryptographic operations—or, in other words, just enough rope for the unwary to hang themselves.

This manual assumes that you already know to use the cryptographic operations properly. Every operation has conditions that must be satisfied for the operation's security properties to hold; they are not always well-advertised in documentation or literature, and they are sometimes revised as new weaknesses or attacks are discovered. Aside from the occasional off-hand comment, this manual does not discuss them at all. You are on your own.

1.1 Cryptography Examples

In order to use a cryptographic operation, you need an implementation of it from a crypto provider. Implementations are managed through crypto factories. This introduction will use the factory for libcrypto (OpenSSL), since it is widely available and supports many useful cryptographic operations. See Cryptography Factories for other crypto providers.

```
> (require crypto)
```

```
> (require crypto/libcrypto)
```

You can configure this library with a "search path" of crypto factories:

```
> (crypto-factories (list libcrypto-factory))
```

That allows you to perform an operation by providing a crypto algorithm specifier, which is automatically resolved to an implementation using the factories in `(crypto-factories)`. For example, to compute a message digest, call the `digest` function with the name of the digest algorithm:

```
> (digest 'sha1 "Hello world!")
```

```
#"\323Hj\351\23nxV\274B!#\205\352yp\224GX\2"
```

Or, if you prefer, you can obtain an algorithm implementation explicitly:

```
> (define sha1-impl (get-digest 'sha1 libcrypto-factory))

> (digest sha1-impl "Hello world!")

#"\323Hj\351\23nxV\274B!#\205\352yp\224GX\2"
```

To encrypt using a symmetric cipher, call the `encrypt` function with a cipher specifier consisting of the name of the cipher and the cipher mode (see `cipher-spec?` for details).

```
> (define skey #"VeryVerySecr3t!!")

> (define iv (make-bytes (cipher-iv-size '(aes ctr)) 0))

> (encrypt '(aes ctr) skey iv "Hello world!")

#"wu\345\215\e\16\256\355.\242\30x"
```

Of course, using an all-zero IV is usually a very bad idea. You can generate a random IV of the right size (if a random IV is appropriate), or you can get the IV size and construct one yourself:

```
> (define iv (generate-cipher-iv '(aes ctr)))

> iv

#"\351\256\17\\f\35051\227\235\17\0007\376\vu"
> (cipher-iv-size '(aes ctr))

16
```

There are also functions to generate session keys, HMAC keys, etc. These functions use `crypto-random-bytes`, a cryptographically strong source of randomness.

When an authenticated encryption (AEAD) cipher, such as AES-GCM, is used with `encrypt` or `decrypt`, the authentication tag is automatically appended to (or taken from) the end of the cipher text, respectively. AEAD ciphers also support *additionally authenticated data*, passed with the `#:aad` keyword.

```
> (define key (generate-cipher-key '(aes gcm)))

> (define iv (generate-cipher-iv '(aes gcm)))

> (define ct (encrypt '(aes gcm) key iv #"Nevermore!" #:aad #"quoth the raven"))

> (decrypt '(aes gcm) key iv ct #:aad #"quoth the raven")

#"Nevermore!"
```

If authentication fails at the end of decryption, an exception is raised:

```
> (decrypt '(aes gcm) key iv ct #:aad #"said the bird")

decrypt: authenticated decryption failed
```

In addition to "all-at-once" operations like `digest` and `encrypt`, this library also supports algorithm contexts for incremental computation.

```
> (define sha1-ctx (make-digest-ctx 'sha1))

> (digest-update sha1-ctx #"Hello ")

> (digest-update sha1-ctx #"world!")

> (digest-final sha1-ctx)

#"\323Hj\351\23nxV\274B!#\205\352yp\224GX\2"
```

1.2 Public-Key Cryptography Examples

Public-key (PK) cryptography uses keypairs consisting of public and private keys. A keypair can be generated by calling `generate-private-key` with the desired PK cryptosystem and an association list of key-generation options. The private key consists of the whole keypair—both private and public components. A key containing only the public components can be obtained with the `pk-key->public-only-key` function.

```
> (define rsa-impl (get-pk 'rsa libcrypto-factory))

> (define privkey (generate-private-key rsa-impl '((nbits 512))))

> (define pubkey (pk-key->public-only-key privkey))
```

RSA keys support both signing and encryption. Other PK cryptosystems may support different operations; for example, DSA supports signing but not encryption, and DH only supports key agreement.

PK signature algorithms are limited in the amount of data they can sign directly, so the message is first processed with a digest function, then the digest is signed. The `digest/sign` and `digest/verify` functions compute the digest automatically. The private key signs, and the public key verifies.

```
> (define sig (digest/sign privkey 'sha1 "Hello world!"))

> (digest/verify pubkey 'sha1 "Hello world!" sig)

#t
> (digest/verify pubkey 'sha1 "Transfer $100" sig)

#f
```

It is also possible to sign a precomputed digest. The digest algorithm is still required as an argument, because some signature schemes include a digest algorithm identifier.

```
> (define dgst (digest 'sha1 "Hello world!"))

> (define sig (pk-sign-digest privkey 'sha1 dgst))

> (pk-verify-digest pubkey 'sha1 (digest 'sha1 "Hello world!") sig)

#t
> (pk-verify-digest pubkey 'sha1 (digest 'sha1 "Transfer $100") sig)
```

#f
Encryption is similar, except that the public key encrypts, and the private key decrypts.

```
> (define skey #"VeryVerySecr3t!!")
> (define e-skey (pk-encrypt pubkey skey))
> (pk-decrypt privkey e-skey)
#"VeryVerySecr3t!!"
```

The other PK operation is key agreement, or shared secret derivation. Two parties exchange public keys, and each party uses their own private key together with their peer's public key to derive a shared secret.

flutter_secure_storage

Note: usage of encryptedSharedPreference

When using the encryptedSharedPreferences parameter on Android, make sure to pass the option to the constructor instead of the function like so:

```
AndroidOptions _getAndroidOptions() => const AndroidOptions(
    encryptedSharedPreferences: true,
  );
final storage = FlutterSecureStorage(aOptions: _getAndroidOptions());
```

This will prevent errors due to mixed usage of encryptedSharedPreferences. For more info, see this issue.

Info

A Flutter plugin to store data in secure storage:

- Keychain is used for iOS
- AES encryption is used for Android. AES secret key is encrypted with RSA and RSA key is stored in KeyStore
- With V5.0.0 we can use EncryptedSharedPreferences on Android by enabling it in the Android Options like so:

```
AndroidOptions _getAndroidOptions() => const AndroidOptions(
    encryptedSharedPreferences: true,
  );
```

For more information see the example app.

- libsecret is used for Linux.

Note KeyStore was introduced in Android 4.3 (API level 18). The plugin wouldn't work for earlier versions.

Getting Started

```dart
import 'package:flutter_secure_storage/flutter_secure_storage.dart';


// Create storage
final storage = new FlutterSecureStorage();


// Read value
String value = await storage.read(key: key);


// Read all values
Map<String, String> allValues = await storage.readAll();


// Delete value
await storage.delete(key: key);


// Delete all
await storage.deleteAll();


// Write value
await storage.write(key: key, value: value);
```

This allows us to be able to fetch secure values while the app is backgrounded, by specifying first_unlock or first_unlock_this_device. The default if not specified is unlocked. An example:

```dart
final options = IOSOptions(accessibility: IOSAccessibility.first_unlock);
await storage.write(key: key, value: value, iOptions: options);
```

Configure Android version

In [project]/android/app/build.gradle set minSdkVersion to >= 18.

```
android {
   ...

   defaultConfig {
```

```
    ...
    minSdkVersion 18
    ...
  }

}
```

Note By default Android backups data on Google Drive. It can cause exception java.security.InvalidKeyException:Failed to unwrap key. You need to

- disable autobackup, details
- exclude sharedprefs FlutterSecureStorage used by the plugin, details

Configure Web Version

Flutter Secure Storage uses an experimental implementation using WebCrypto. Use at your own risk at this time. Feedback welcome to improve it. The intent is that the browser is creating the private key, and as a result, the encrypted strings in local_storage are not portable to other browsers or other machines and will only work on the same domain.

It is VERY important that you have HTTP Strict Forward Secrecy enabled and the proper headers applied to your responses or you could be subject to a javascript hijack.

Please see:

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security
- https://www.netsparker.com/blog/web-security/http-security-headers/

Configure Linux Version

You need libsecret-1-dev and libjsoncpp-dev on your machine to build the project, and libsecret-1-0 and libjsoncpp1 to run the application (add it as a dependency after packaging your app). If you using snapcraft to build the project use the following

```
parts:
 uet-lms:
  source: .
  plugin: flutter
  flutter-target: lib/main.dart
  build-packages:
   - libsecret-1-dev
   - libjsoncpp-dev
  stage-packages:
```

- libsecret-1-0
- libjsoncpp-dev

Configure Windows Version

Note The current implementation does not support readAll and deleteAll and is subject to change.

Configure MacOS Version

You also need to add Keychain Sharing as capability to your macOS runner. To achieve this, please add the following
in *both* your macos/Runner/DebugProfile.entitlements *and* macos/Runner/Release.entitlements (yo u need to change both files).

<key>keychain-access-groups</key>

<array/>

Integration Tests

Run the following command from example directory

flutter drive --target=test_driver/app.dart

How to keep user data safe in mobile app development

While developing a mobile application, what's the biggest challenge that businesses face? Users' data privacy and security! Keeping the users' data safe is one of the biggest concerns in today's digital landscape. With the rise of Cyberattacks and breaches, data security management has become mandatory during mobile app development. As mobile application security is at risk today, hackers are trying new tricks and techniques to exploit vulnerabilities in mobile applications.

So, organizations must take the best security measures to keep their applications' user data safe. According to the research, nearly **50%** of organizations' employees download one or two malicious applications. Also, approximately **40%** of mobile apps remain vulnerable to cyberattacks. For this reason, users are more afraid of using a new application and cannot trust it easily. But there is a solution to it.

If you are developing a mobile app, try the following mobile app development security practices to keep your users' data safe and protected.

What is mobile app security?

Mobile app security protects the application from external risks such as viruses and other cyber threats by incorporating standard security protocols. These malware and online dangers put financial and other sensitive data at risk of hackers.

In today's digital world, mobile application security has also become vital. Personal information about users may be accessible to hackers through a mobile security breach. Security protocols are applied during mobile app development to ensure that none of the users' data gets leaked or misused.

7 Mobile app development practices for securing user data

Want to build a mobile app that is completely safe and secure? Try the following mobile development practices for enhanced user data security.

1. Encryption of data & code

Bugs can generate through insignificant coding mistakes or testing oversights. It makes mobile apps susceptible to hacker exploitation. So, using data encryption while creating a mobile app is the best solution.

Code encryption and data encryption are the two types of encryption available. In code encryption, the text is changed into numerical code series. Data encryption, on the other hand, changes data into a form that is unreadable to hackers.

- ADVERTISEMENT -

2. Error-free codes

As per professional mobile app developers, approximately **82%** of the vulnerabilities in mobile applications are caused due to inappropriate coding practices. It demonstrates that the code requires high security, and mobile app developers should ensure it is **100%** error-free before debugging.

Comprehending its architecture is one of the best techniques to safeguard the application or software. You can hire dedicated developers who test the application from different perspectives and ensure the code is safe. It will help you develop a secure application and protect your app users' data.

3. Ensure high-level authentication

High-level encryption is a must when developing mobile apps. So, create an application that reminds users to periodically change their passwords. Add features like alphanumeric passwords as well. It increases security and adds a layered authentication procedure.

Developers can also add capabilities for biometric authentication or retina scanning. These two features leverage high-level authentication that is virtually impossible to circumvent. Another efficient technique to safeguard the app is OTP (one-time password).

4. Perform a strong security check

Verifying the mobile app's usability and security before the launch is mandatory. Finding the application's vulnerability is the aim. Even after the app has been released, frequent testing is advised.

To achieve a launch deadline, the developer's team might occasionally test the software even though app vulnerabilities may endanger user security. To ensure perfect authentication and authorization processes, conduct code audits and testing.

5. Beware of third-party libraries

It becomes easier for developers to code using third-party libraries rather than constructing an application from scratch. Also, there is a tonne of free libraries that make calling specific functions easier. Be cautious when using such libraries, as you could expose your app to security issues.

When it comes to third-party libraries, be twice as cautious. Before incorporating the code into your app, properly test it and examine the library's many versions. Once thoroughly examined or a new

problem arises, many of the largest coding libraries still have security issues. Use several internal controller repositories and policy controls to shield your apps from potential weaknesses in these libraries. These regulations will assist in separating the data layer.

6. Control data sharing within apps

For developers wishing to transfer data between two or more apps, data sniffing has a lot of promise. Data transfer may be problematic, particularly if hackers figure out how you accomplish it and discover that it is unprotected.

Signature-based permissions can prevent needless interference while transferring data across programs you manage. The apps' functionality is maintained as these permissions don't require user action. Instead, they see if the data-accessing apps have the same signature and signing key.

7. Insert secure APIs

Finally, secure APIs are the last method to ensure client data security within your mobile application. APIs are the main method for transferring data between applications, the cloud, and consumers. Developers must guarantee the security of the code when utilizing third-party APIs.

Only allow the necessary applications to access APIs with data authorization. To secure your app's API – Use a strong gateway or a central OAuth server to securely manage user authentication.

Implementing these strategies above will help you create a secure mobile application and save huge costs during mobile app development.

TLS/SSL

What is SSL and why is it important?

Secure Sockets Layer (SSL) certificates, sometimes called digital certificates, are used to establish an encrypted connection between a browser or user's computer and a server or website.

SSL: SECURE SOCKETS LAYER

SSL is standard technology for securing an internet connection by encrypting data sent between a website and a browser (or between two servers). It prevents hackers from seeing or stealing any information transferred, including personal or financial data.

RELATED TERMS

TLS: Transport Layer Security

TLS is an updated, more secure version of SSL. We still refer to our security certificates as SSL because it's a more common term, but when you buy SSL from DigiCert, you get the most trusted, up-to-date TLS certificates.

HTTPS: Hyper Text Protocol Secure

HTTPS appears in the URL when a website is secured by an SSL/TLS certificate. Users can view the details of the certificate, including the issuing authority and the corporate name of the website owner, by clicking the lock symbol on the browser bar.

WHY DO YOU NEED SSL?

SSL isn't just for ecommerce. It secures all types of information transferred to and from your website.

CHECKOUT PAGES

Customers are more likely to complete a purchase if they know your checkout area (and the credit card info they share) is secure.

LOGIN PANELS & FORMS

SSL encrypts and protects usernames and passwords, as well as forms used to submit personal information, documents or images.

BLOGS & INFORMATIONAL SITES

Even blogs and websites that don't collect payments or sensitive information need HTTPS to keep user activity private.

SSL IMPROVES SEO

In 2014, Google called for HTTPS everywhere to improve security across the web — and they rewarded SSL-secured sites with higher rankings. In 2018, Google went beyond search rankings and began punishing sites without SSL certificates by flagging them as "not secure" in the Chrome browser.

HOW DOES TLS/SSL INCREASE TRUST?

Not all TLS/SSL certificates are created equal. Beyond encryption, TLS certificates also authenticate the identity of a website owner. This provides an added layer of security which users can see if they look beyond the lock. Certificates are offered with three levels of this identity verification:

- Extended Validation SSL Certificates
- Organization Validated SSL Certificates
- Domain Validated SSL Certificates

LOOK BEYOND THE LOCK

Just seeing a padlock in the address bar is no longer enough

By clicking on the padlock icon in the URL bar you can verify the identity of the website owner. Unfortunately, most phishing sites today have a padlock and a DV certificate. That's why it's important to look beyond the lock in the URL bar. If a website is not willing to put their identity in the certificate, you shouldn't be willing to share any identifying information with them. If you see the organization's name, now you can make a better decision about who you trust.

HOW DO SSL CERTIFICATES WORK?

SSL certificates establish an encrypted connection between a website/server and a browser with what's known as an "SSL handshake." For visitors to your website, the process is invisible — and instantaneous.

Authentication

For every new session a user begins on your website, their browser and your server exchange and validate each other's SSL certificates.

Encryption

Your server shares its public key with the browser, which the browser then uses to create and encrypt a pre-master key. This is called the key exchange.

Decryption

The server decrypts the pre-master key with its private key, establishing a secure, encrypted connection used for the duration of the session.

What is SSL?

SSL stands for Secure Sockets Layer, and it refers to a protocol for encrypting, securing, and authenticating communications that take place on the Internet. Although SSL was replaced by an updated protocol called TLS (Transport Layer Security) some time ago, "SSL" is still a commonly used term for this technology.

The main use case for SSL/TLS is securing communications between a client and a server, but it can also secure email, VoIP, and other communications over unsecured networks.

How does SSL/TLS work?

These are the essential principles to grasp for understanding how SSL/TLS works:

- Secure communication begins with a TLS handshake, in which the two communicating parties open a secure connection and exchange the public key

- During the TLS handshake, the two parties generate session keys, and the session keys encrypt and decrypt all communications after the TLS handshake
- Different session keys are used to encrypt communications in each new session
- TLS ensures that the party on the server side, or the website the user is interacting with, is actually who they claim to be
- TLS also ensures that data has not been altered, since a message authentication code (MAC) is included with transmissions

With TLS, both HTTP data that users send to a website (by clicking, filling out forms, etc.) and the HTTP data that websites send to users is encrypted. Encrypted data has to be decrypted by the recipient using a key.

The TLS handshake

TLS communication sessions begin with a TLS handshake. A TLS handshake uses something called asymmetric encryption, meaning that two different keys are used on the two ends of the conversation. This is possible because of a technique called public key cryptography.

In public key cryptography, two keys are used: a public key, which the server makes available publicly, and a private key, which is kept secret and only used on the server side. Data encrypted with the public key can only be decrypted with the private key.

During the TLS handshake, the client and server use the public and private keys to exchange randomly generated data, and this random data is used to create new keys for encryption, called the session keys.

Symmetric encryption with session keys

Unlike asymmetric encryption, in symmetric encryption the two parties in a conversation use the same key. After the TLS handshake, both sides use the same session keys for encryption. Once session keys are in use, the public and private keys are not used anymore. Session keys are temporary keys that are not used again once the session is terminated. A new, random set of session keys will be created for the next session.

Authenticating the origin server

TLS communications from the server include a message authentication code, or MAC, which is a digital signature confirming that the communication originated from the actual website. This authenticates the server, preventing on-path attacks and domain spoofing. It also ensures that the data has not been altered in transit.

What is an SSL certificate?

An SSL certificate is a file installed on a website's origin server. It's simply a data file containing the public key and the identity of the website owner, along with other information. Without an SSL certificate, a website's traffic can't be encrypted with TLS.

Technically, any website owner can create their own SSL certificate, and such certificates are called self-signed certificates. However, browsers do not consider self-signed certificates to be as trustworthy as SSL certificates issued by a certificate authority.

How does a website get an SSL certificate?

Website owners need to obtain an SSL certificate from a certificate authority, and then install it on their web server (often a web host can handle this process). A certificate authority is an outside party who can confirm that the website owner is who they say they are. They keep a copy of the certificates they issue.

Is it possible to get a free SSL certificate?

Many certificate authorities charge for SSL certificates. To help make the Internet more secure, Cloudflare offers free SSL certificates. Cloudflare was the first Internet security and performance company to do so. Cloudflare also has worked to optimize SSL/TLS performance so that websites moving from HTTP to HTTPS do not have their performance impacted. For more information about SSL options with Cloudflare, see our Developer documentation.

What is the difference between HTTP and HTTPS?

The S in "HTTPS" stands for "secure." HTTPS is just HTTP with SSL/TLS. A website with an HTTPS address has a legitimate SSL certificate issued by a certificate authority, and traffic to and from that website is authenticated and encrypted with the SSL/TLS protocol.

To encourage the Internet as a whole to move to the more secure HTTPS, many web browsers have started to mark HTTP websites as "not secure" or "unsafe." Thus, not only is HTTPS essential for keeping users safe and user data secure, it has also become essential for building trust with users.

How can you secure mobile app communication with a server during authentication?

Powered by AI and the LinkedIn community

Mobile apps often need to communicate with a server to authenticate users, access data, or perform other functions. However, this communication can be vulnerable to various attacks, such as eavesdropping, tampering, or spoofing. Therefore, it is essential to secure mobile app communication with a server during authentication. In this article, you will learn some best practices and techniques to achieve this goal.

Top experts in this article

Selected by the community from 1 contribution. Learn more

Acknowledge great insights

Access advice in these articles from experts and let them know you find their contribution valuable.

Get started

See what others are saying

Use HTTPS

The first and most basic step to secure mobile app communication with a server is to use HTTPS, which stands for Hypertext Transfer Protocol Secure. HTTPS is a protocol that encrypts the data exchanged between the app and the server, making it harder for attackers to intercept or modify it. HTTPS also verifies the identity of the server, preventing the app from connecting to a fake or malicious one. To use HTTPS, you need to obtain a valid SSL/TLS certificate from a trusted authority and configure your server to accept HTTPS requests.

Configuring SSL or TLS certificates

Last Updated: 2023-01-12

If you use an LDAP directory server for user authentication and you want to enable SSL encryption or TLS authentication, you must configure your SSL or TLS certificate. QRadar LDAP authentication uses TLS 1.2.

Procedure

1. Using SSH, log in to your system as the root user.
2. Type the following command to create the `/opt/qradar/conf/trusted_certificates/` directory:

```
mkdir -p /opt/qradar/conf/trusted_certificates
```

3. Copy the SSL or TLS certificate from the LDAP server to the `/opt/qradar/conf/trusted_certificates` directory on your system.
4. Verify that the certificate file name extension is `.cert`, which indicates that the certificate is trusted.

   The QRadar system loads only `.cert` files.

Network security configuration

bookmark_border

The Network Security Configuration feature lets you customize your app's network security settings in a safe, declarative configuration file without modifying app code. These settings can be configured for specific domains and for a specific app. The key capabilities of this feature are:

- **Custom trust anchors:** Customize which Certificate Authorities (CA) are trusted for an app's secure connections. For example, trusting particular self-signed certificates or restricting the set of public CAs that the app trusts.
- **Debug-only overrides:** Safely debug secure connections in an app without added risk to the installed base.
- **Cleartext traffic opt-out:** Protect apps from accidental usage of cleartext (unencrypted) traffic.
- **Certificate pinning:** Restrict an app's secure connection to particular certificates.

Add a Network Security Configuration file

The Network Security Configuration feature uses an XML file where you specify the settings for your app. You must include an entry in your app's manifest to point to this file. The following code excerpt from a manifest demonstrates how to create this entry:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
  <application android:networkSecurityConfig="@xml/network_security_config"
              ... >
    ...
  </application>
</manifest>
```

Customize trusted CAs

You might want your app to trust a custom set of CAs instead of the platform default. The most common reasons for this are:

- Connecting to a host with a custom CA, such as a CA that is self-signed or is issued internally within a company.
- Limiting the set of CAs to only the CAs you trust instead of every pre-installed CA.
- Trusting additional CAs not included in the system.

By default, secure connections (using protocols like TLS and HTTPS) from all apps trust the pre-installed system CAs, and apps targeting Android 6.0 (API level 23) and lower also trust the user-added CA store by default. You can customize your app's connections using base-config (for app-wide customization) or domain-config (for per-domain customization).

Configure a custom CA

You might want to connect to a host that uses a self-signed SSL certificate or to a host whose SSL certificate is issued by a non-public CA that you trust, such as your company's internal CA. The following code excerpt demonstrates how to configure your app for a custom CA in res/xml/network_security_config.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
  <domain-config>
    <domain includeSubdomains="true">example.com</domain>
    <trust-anchors>
      <certificates src="@raw/my_ca"/>
    </trust-anchors>
  </domain-config>
</network-security-config>
```

Add the self-signed or non-public CA certificate, in PEM or DER format, to res/raw/my_ca.

Limit the set of trusted CAs

If you don't want your app to trust all CAs trusted by the system, you can instead specify a reduced set of CAs to trust. This protects the app from fraudulent certificates issued by any of the other CAs.

The configuration to limit the set of trusted CAs is similar to trusting a custom CA for a specific domain except that multiple CAs are provided in the resource. The following code excerpt demonstrates how to limit your app's set of trusted CAs in res/xml/network_security_config.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
  <domain-config>
    <domain includeSubdomains="true">secure.example.com</domain>
    <domain includeSubdomains="true">cdn.example.com</domain>
    <trust-anchors>
      <certificates src="@raw/trusted_roots"/>
    </trust-anchors>
  </domain-config>
</network-security-config>
```

Add the trusted CAs, in PEM or DER format, to res/raw/trusted_roots. Note that if you use PEM format, the file must contain *only* PEM data and no extra text. You can also provide multiple <certificates> elements instead of one.

Trust additional CAs

You might want your app to trust additional CAs that aren't trusted by the system, such as if the system doesn't yet include the CA or the CA doesn't meet the requirements for inclusion in the Android system. You can specify multiple certificate sources for a configuration in res/xml/network_security_config.xml using code like the following excerpt.

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
  <base-config>
    <trust-anchors>
      <certificates src="@raw/extracas"/>
      <certificates src="system"/>
    </trust-anchors>
  </base-config>
</network-security-config>
```

Configure CAs for debugging

When debugging an app that connects over HTTPS, you may want to connect to a local development server that does not have the SSL certificate for your production server. To support this without any modification to your app's code, you can specify debug-only CAs, which are trusted *only* when android:debuggable is true, by using debug-overrides. Normally, IDEs and build tools set this flag automatically for non-release builds.

This is safer than the usual conditional code because, as a security precaution, app stores do not accept apps that are marked debuggable.

The excerpt below shows how to specify debug-only CAs in res/xml/network_security_config.xml:

```xml
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
   <debug-overrides>
      <trust-anchors>
         <certificates src="@raw/debug_cas"/>
      </trust-anchors>
   </debug-overrides>
</network-security-config>
```

Opt out of cleartext traffic

Note: The guidance in this section applies only to apps that target Android 8.1 (API level 27) or lower. Starting with Android 9 (API level 28), cleartext support is disabled by default.

If you intend for your app to connect to destinations using only secure connections, you can opt out of supporting cleartext (using the unencrypted HTTP protocol instead of HTTPS) to those destinations. This option helps prevent accidental regressions in apps due to changes in URLs provided by external sources such as backend servers.
See NetworkSecurityPolicy.isCleartextTrafficPermitted() for more details.

For example, you might want your app to ensure that connections to secure.example.com are always done over HTTPS to protect sensitive traffic from hostile networks.

The excerpt below shows how to opt out of cleartext in res/xml/network_security_config.xml:

```xml
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
   <domain-config cleartextTrafficPermitted="false">
      <domain includeSubdomains="true">secure.example.com</domain>
   </domain-config>
</network-security-config>
```

Pin certificates

Normally, an app trusts all pre-installed CAs. If any of these CAs were to issue a fraudulent certificate, the app would be at risk from an on-path attacker. Some apps choose to limit the set of certificates they accept by either limiting the set of CAs they trust or by certificate pinning.

Certificate pinning is done by providing a set of certificates by hash of the public key (SubjectPublicKeyInfo of the X.509 certificate). A certificate chain is then valid only if the certificate chain contains at least one of the pinned public keys.

Note that, when using certificate pinning, you should always include a backup key so that if you are forced to switch to new keys or change CAs (when pinning to a CA certificate or an intermediate of that CA), your app's connectivity is unaffected. Otherwise, you must push out an update to the app to restore connectivity.

Additionally, it is possible to set an expiration time for pins after which pinning is not performed. This helps prevent connectivity issues in apps which have not been updated. However, setting an expiration time on pins may enable attackers to bypass your pinned certificates.

The excerpt below shows how to pin certificates in res/xml/network_security_config.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
  <domain-config>
    <domain includeSubdomains="true">example.com</domain>
    <pin-set expiration="2018-01-01">
      <pin digest="SHA-256">7HIpactkIAq2Y49orFOOQKurWxmmSFZhBCoQYcRhJ3Y=</pin>
      <!-- backup pin -->
      <pin digest="SHA-256">fwza0LRMXouZHRC8Ei+4PyuldPDcf3UKgO/04cDM1oE=</pin>
    </pin-set>
  </domain-config>
</network-security-config>
```

Configuration inheritance behavior

Values not set in a specific configuration are inherited. This behavior allows more complex configurations while keeping the configuration file readable.

For example, values not set in a domain-config are taken from the parent domain-config, if nested, or from the base-config, if not. Values not set in the base-config use the platform default values.

For example, consider a case where all connections to subdomains of example.com must use a custom set of CAs. Additionally, cleartext traffic to these domains is permitted *except* when connecting to secure.example.com. By nesting the configuration for secure.example.com inside the configuration for example.com, the trust-anchors does not need to be duplicated.

The excerpt below shows how this nesting would look in res/xml/network_security_config.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
  <domain-config>
    <domain includeSubdomains="true">example.com</domain>
    <trust-anchors>
      <certificates src="@raw/my_ca"/>
    </trust-anchors>
    <domain-config cleartextTrafficPermitted="false">
      <domain includeSubdomains="true">secure.example.com</domain>
    </domain-config>
  </domain-config>
</network-security-config>
```

Configuration file format

The Network Security Configuration feature uses an XML file format. The overall structure of the file is shown in the following code sample:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
  <base-config>
```

```
        <trust-anchors>
            <certificates src="..."/>
            ...
        </trust-anchors>
    </base-config>

    <domain-config>
        <domain>android.com</domain>
        ...
        <trust-anchors>
            <certificates src="..."/>
            ...
        </trust-anchors>
        <pin-set>
            <pin digest="...">...</pin>
            ...
        </pin-set>
    </domain-config>
    ...
    <debug-overrides>
        <trust-anchors>
            <certificates src="..."/>
            ...
        </trust-anchors>
    </debug-overrides>
</network-security-config>
```

The following sections describe the syntax and other details of the file format.

<network-security-config>

can contain:

0 or 1 of [<base-config>]()
Any number of [<domain-config>]()
0 or 1 of [<debug-overrides>]()

<base-config>

syntax:

```
<base-config cleartextTrafficPermitted=["true" | "false"]>
    ...
</base-config>
```

can contain:

[<trust-anchors>]()

description:

The default configuration used by all connections whose destination is not covered by a domain-config.

Any values that are not set use the platform default values.

The default configuration for apps targeting Android 9 (API level 28) and higher is as follows:

```
<base-config cleartextTrafficPermitted="false">
    <trust-anchors>
        <certificates src="system" />
    </trust-anchors>
</base-config>
```

The default configuration for apps targeting Android 7.0 (API level 24) to Android 8.1 (API level 27) is as follows:

```
<base-config cleartextTrafficPermitted="true">
    <trust-anchors>
        <certificates src="system" />
    </trust-anchors>
</base-config>
```

The default configuration for apps targeting Android 6.0 (API level 23) and lower is as follows:

```
<base-config cleartextTrafficPermitted="true">
    <trust-anchors>
        <certificates src="system" />
        <certificates src="user" />
    </trust-anchors>
</base-config>
```

<domain-config>

syntax:

```
<domain-config cleartextTrafficPermitted=["true" | "false"]>
    ...
</domain-config>
```

can contain:

1 or more <domain>
0 or 1 <trust-anchors>
0 or 1 <pin-set>
Any number of nested <domain-config>

description:

Configuration used for connections to specific destinations, as defined by the domain elements.

Note that if multiple domain-config elements cover a destination, the configuration with the most specific (longest) matching domain rule is used.

<domain>

syntax:

```
<domain includeSubdomains=["true" | "false"]>example.com</domain>
```

attributes:

includeSubdomains

If "true", then this domain rule matches the domain and all subdomains, including subdomains of subdomains. Otherwise, the rule only applies to exact matches.

<debug-overrides>

syntax:

```
<debug-overrides>
    ...
</debug-overrides>
```

can contain:

0 or 1 <trust-anchors>

description:

Overrides to be applied when android:debuggable is "true", which is normally the case for non-release builds generated by IDEs and build tools. Trust anchors specified in debug-overrides are added to all other configurations, and certificate pinning is not performed when the server's certificate chain uses one of these debug-only trust anchors. If android:debuggable is "false", then this section is completely ignored.

<trust-anchors>

syntax:

```
<trust-anchors>
...
</trust-anchors>
```

can contain:

Any number of <certificates>

description:

Set of trust anchors for secure connections.

<certificates>

syntax:

```
<certificates src=["system" | "user" | "raw resource"]
        overridePins=["true" | "false"] />
```

description:

Set of X.509 certificates for trust-anchors elements.

attributes:

src

The source of CA certificates. Each certificate can be one of the following:

- a raw resource ID pointing to a file containing X.509 certificates. Certificates must be encoded in DER or PEM format. In the case of PEM certificates, the file *must not* contain extra non-PEM data such as comments.
- "system" for the pre-installed system CA certificates
- "user" for user-added CA certificates

overridePins

Specifies if the CAs from this source bypass certificate pinning. If "true", then pinning is not performed on certificate chains which are signed by one of the CAs from this source. This can be useful for debugging CAs or for testing man-in-the-middle attacks on your app's secure traffic.

Default is "false" unless specified in a debug-overrides element, in which case the default is "true".

<pin-set>

syntax:

```
<pin-set expiration="date">
...
</pin-set>
```

can contain:

Any number of <pin>

description:

A set of public key pins. For a secure connection to be trusted, one of the public keys in the chain of trust must be in the set of pins. See <pin> for the format of pins.

attributes:

expiration

The date, in yyyy-MM-dd format, on which the pins expire, thus disabling pinning. If the attribute is not set, then the pins do not expire.

Expiration helps prevent connectivity issues in apps which do not get updates to their pin set, such as when the user disables app updates.

<pin>

syntax:

```
<pin digest=["SHA-256"]>base64 encoded digest of X.509
   SubjectPublicKeyInfo (SPKI)</pin>
```

attributes:

digest

The digest algorithm used to generate the pin. Currently, only "SHA-256" is supported.

Additional resources

For more information about Network Security Configuration, consult the following resources.

Obtain user consent

One of the most important steps to respect user privacy and protect their data is to obtain their consent before you collect, use, or share their data. Consent means that the user has given you clear, informed, and voluntary permission to process their data for a specific purpose. You need to provide the user with a clear and concise privacy notice that explains what data you collect, why you collect it, how you use it, who you share it with, how long you keep it, and what rights they have over it. You also need to provide the user with a simple and easy way to opt-in or opt-out of your data processing activities, and to withdraw or modify their consent at any time.

Privacy Compliance Law

The area of **privacy compliance law** addresses how organizations meet legal and regulatory requirements for collecting, processing, or maintaining personal information. Data privacy breaches can lead to regulatory investigations and fines. When privacy is compromised, consumers or employees may respond with civil lawsuits. It is recommended, but not required by a federal law, that companies create and post privacy policies on websites and mobile apps. Once posted, companies must follow these policies or face scrutiny by the Federal Trade Commission. (California and Delaware state law does require privacy policies to be posted on websites and mobile applications, if the site collects personally identifiable information.)

The majority of U.S. states have passed security breach disclosure laws, meaning companies must follow privacy compliance laws at the state levels. Federal laws such as the healthcare-related HIPAA law and the financial law, Gramm-Leach-Bliley Act, also have precise privacy regulations that must be followed by companies in order to avoid fines and legal liability.

General Data Protection Regulation (GDPR)

The General Data Protection Regulation (GDPR), implemented by the European Union (EU), has completely transformed the data privacy landscape around the world. Although it is a European privacy law, it impacts businesses around the world – for example, an American business has to consider GDPR if it has customers in the EU.

GDPR places an emphasis on individuals' consent and control over their personal data. If you're a marketer targeting citizens in the EU, it's imperative that you're compliant with GDPR.

Under the GDPR, you must obtain clear consent from users before collecting or processing their personal data. You must also provide clear information about how their data will be used, and make it easy for users to decide what will happen to their data.

California Consumer Privacy Act (CCPA) and California Privacy Rights Act (CPRA)

It's not just the EU that has data protection laws. In the United States, effective privacy regulations have been established through the California Consumer Privacy Act (CCPA) and the California Privacy Rights Act (CPRA).

These acts provide protection for individuals' privacy rights, and grant residents of California more control over their personal data. It also imposes obligations on organizations that collect or sell this data.

The CCPA ensures businesses inform consumers about what data they are collecting, why they ate collecting the data, as well as the categories of third parties they're sharing the data with.

According to the CCPA, users can opt out of their personal data being sold. The CPRA further touches on consumer rights and introduces even stricter rules for businesses to obey.

Personal Information Protection and Electronic Documents Act (PIPEDA)

The Personal Information Protection and Electronic Document Act (PIPEDA) is a Canadian data privacy law that governs how businesses use, collect and disclose personal data. It applies to organizations that conduct commercial activities across Canada.

If you're a marketer that conducts business activity in Canada, you must obtain clear consent, disclose the reason you're collecting data, and provide users with access to the data you collect. PIPEDA also requires businesses to implement safeguards on data security and have a clear process for addressing data breaches.

Key considerations for marketers

As a marketer, there's much to consider outside of your marketing strategy – compliance being a key point to consider. Compliance goes far beyond legal obligations – it's essential for maintaining customer trust and loyalty.

First of all, consider transparency, especially in regard to consent. Be sure to obtain explicit consent from users, and clearly communicate the reason you're collecting their data. Another way to increase transparency is to provide easy-to-understand privacy policies.

You should only collect and retain the necessary data required for marketing purposes, which can minimize the risk of data breaches. It's important that you have security measures in place that protect personal data – for example, access controls, audits and encryption.

When dealing with third parties, ensure that anybody handling personal data complies with the relevant data protection/ data privacy laws and that they also have the correct data protection processes in place.

Likewise, ensure that you have established processes that address user rights. This includes the right to access, correct, delete and restrict how you process their personal data.

Finally, consider cross-border data transfers. This involves learning the rules regarding transferring data across borders (e.g the EU-US Privacy Shield).

Mobile App Permissions - Best Practices for Developers

Whenever we install an application on iOS or Android - we get a notification from the developers to give ‘permissions to access location, contacts, photos, etc’ - these **mobile app permissions** vary from application to application. Meaning if we have downloaded a ride-hailing app, we would need to permit the app to access our location and contacts, if we have downloaded a photo editing app we would need to give access to our photo gallery, and so on and so forth.

These necessary permissions only access the information for the proper functioning of the app. Many times we may have noticed that if we ignore it, the app won’t open. These permissions protect our valuable information but sometimes if we ignore paying attention - we might end up giving access to information not even needed by the app. Collecting related information makes sense but people still should keep an eye out and avoid giving out unnecessary info for example a photo editing application accessing your contacts.

This is where mobile app developers need to be very cautious when developing and deploying apps because it can make or break your business. So if you are developing an app, make sure you will ask only for the information that is needed - it is very important for customer trust-building.

What Are Mobile App Permissions?

Permissions are when users consent to allow your mobile application to access the information that it needs to function fully after they have installed it.

The category of the application dictates and defines the permissions that it needs such as location, gallery, contact, etc. Additionally, they are enabled based on consent, after the user has installed the application, the app provides them with a list of permissions needed, if the users agree they will click ‘I agree’ to run the app.

The app permissions are presented to be more transparent on the part of the developers, meaning it proves that developers are not doing something shady, they are transparent with nothing to hide, and that there is an authentic reason why they need the permissions. But sometimes we get permissions that have nothing to do with the app, in that case, customers are supposed to be very careful before giving out any information that is irrelevant to the app.

Android App Permissions

- **Storage:** for internal and external file storage
- **Body sensors:** permissions to allow access to health data such as heart rate monitors, etc.
- **Calendar:** for creating, editing, and deleting events
- **Camera:** to take photos and record videos
- **Contacts:** for creating, editing, and deleting contacts, also accessing contact lists of linked accounts on your devices
- **Location:** for high wifi proximity
- **Microphone:** to record audio
- **Phone:** for making calls, voicemail, call redirection et al.
- **SMS:** for reading, receiving, and sending MMS and SMS messages.

Best Practices For Developers For In-App Permissions

Ask for permissions right after the app installation

The best thing to do is to ask for permissions right after the customer has onboarded your app. It is probably one of the most effective and easiest ways. Because this way there would be no surprises for your customers after they have started using the application plus they would know beforehand about the things that the app will be accessing to function.

Just so you know, some users will accept the mobile app permissions and others might still be skeptical, now this is where you need to come up with the best strategy moving forward.

Send a reminder for in-app permissions

If the customers have refused to give the permit once you have asked, you can always send out a reminder. But since you can only ask for permissions once, because you don't want to annoy your customers or make them turn away, it does not mean that you can not send a polite, friendly, gentle reminder. Here you may need to come up with striking content, which after reading the users would have no other option but to agree.

You can either tell them why you need the permissions or ask them a question about what they think about permitting and the things that they are missing out on. Asking twice can actually do you more good than harm, because maybe the second request was sent out at a more appropriate time and users will accept the prompt, and maybe they will know they are missing out on some really cool app features and are eager to accept the second prompt.

Send a permission prompt only when needed

The most important thing is to make your customers trust you first because just as you wouldn't want anyone to enter your room without permission as it's your personal space similarly your customers need to trust you first.

Therefore only ask for permission requests when seriously needed and unless they are critical without which the app wouldn't even open and then ask for secondary mobile app permissions along with the way as the user familiarizes himself with your application.

Educating users about the need for mobile app permissions

Not everyone blindly follows what's asked of them. Therefore it is important you give your users some context beforehand and the best thing to do is during the onboarding.

Through an explainer video on onboarding, you can not only explain the benefits of the application and the reasons why customers should use it but also what they need in order for the app to function smoothly. You can educate them about the importance of app permissions while highlighting your core values of transparency.

In short, you need to clarify why you need access. But still, chances are that even after listening to the benefits, customers still may want to decline the request yet you shouldn't force them to stick around, instead, you should openly give a decline option, that's another reason why your users would trust you.

Pro tip

Always respect your users, just as you wouldn't appreciate anyone disturbing your personal space similarly customers wouldn't either. Therefore, you have to find a perfect strategy that should not annoy your users with unexpected permission requests but also make it easier for you to implement your processes in order for apps to function properly.

Mobile App Authentication Architectures

Authentication and authorization problems are prevalent security vulnerabilities. In fact, they consistently rank second highest in the OWASP Top 10.

Most mobile apps implement some kind of user authentication. Even though part of the authentication and state management logic is performed by the backend service, authentication is such an integral part of most mobile app architectures that understanding its common implementations is important.

Since the basic concepts are identical on iOS and Android, we'll discuss prevalent authentication and authorization architectures and pitfalls in this generic guide. OS-specific authentication issues, such as local and biometric authentication, will be discussed in the respective OS-specific chapters.

General Assumptions

Appropriate Authentication is in Place

Perform the following steps when testing authentication and authorization:

- Identify the additional authentication factors the app uses.
- Locate all endpoints that provide critical functionality.
- Verify that the additional factors are strictly enforced on all server-side endpoints.

Authentication bypass vulnerabilities exist when authentication state is not consistently enforced on the server and when the client can tamper with the state. While the backend service is processing requests from the mobile client, it must consistently enforce authorization checks: verifying that the user is logged in and authorized every time a resource is requested.

Consider the following example from the [OWASP Web Testing Guide](). In the example, a web resource is accessed through a URL, and the authentication state is passed through a GET parameter:

http://www.site.com/page.asp?authenticated=no

The client can arbitrarily change the GET parameters sent with the request. Nothing prevents the client from simply changing the value of the authenticated parameter to "yes", effectively bypassing authentication.

Although this is a simplistic example that you probably won't find in the wild, programmers sometimes rely on "hidden" client-side parameters, such as cookies, to maintain authentication state. They assume that these parameters can't be tampered with. Consider, for example, the following classic vulnerability in Nortel Contact Center Manager. The administrative web application of Nortel's appliance relied on the cookie "isAdmin" to determine whether the logged-in user should be granted administrative privileges. Consequently, it was possible to get admin access by simply setting the cookie value as follows:

isAdmin=True

Security experts used to recommend using session-based authentication and maintaining session data on the server only. This prevents any form of client-side tampering with the session state. However, the whole point of using stateless authentication instead of session-based authentication is to *not* have session state on the server. Instead, state is stored in client-side tokens and transmitted with every request. In this case, seeing client-side parameters such as isAdmin is perfectly normal.

To prevent tampering cryptographic signatures are added to client-side tokens. Of course, things may go wrong, and popular implementations of stateless authentication have been vulnerable to attacks. For example, the signature verification of some JSON Web Token (JWT) implementations could be deactivated by setting the signature type to "None".

Best Practices for Passwords

Password strength is a key concern when passwords are used for authentication. The password policy defines requirements to which end users should adhere. A password policy typically specifies password length, password complexity, and password topologies. A "strong" password policy makes manual or automated password cracking difficult or impossible. For further information please consult the OWASP Authentication Cheat Sheet.

General Guidelines on Testing Authentication

There's no one-size-fits-all approach to authentication. When reviewing the authentication architecture of an app, you should first consider whether the authentication method(s) used are appropriate in the given context. Authentication can be based on one or more of the following:

- Something the user knows (password, PIN, pattern, etc.)
- Something the user has (SIM card, one-time password generator, or hardware token)
- A biometric property of the user (fingerprint, retina, voice)

The number of authentication procedures implemented by mobile apps depends on the sensitivity of the functions or accessed resources. Refer to industry best practices when reviewing authentication functions. Username/password authentication (combined with a reasonable password policy) is generally considered sufficient for apps that have a user login and aren't very sensitive. This form of authentication is used by most social media apps.

For sensitive apps, adding a second authentication factor is usually appropriate. This includes apps that provide access to very sensitive information (such as credit card numbers) or allow users to transfer funds. In some industries, these apps must also comply with certain standards. For example, financial apps have to ensure compliance with the Payment Card Industry Data Security Standard (PCI DSS), the Gramm Leach Bliley Act, and the Sarbanes-Oxley Act (SOX). Compliance considerations for the US health care sector include the Health Insurance Portability and Accountability Act (HIPAA) and the Patient Safety Rule.

Stateful vs. Stateless Authentication

You'll usually find that the mobile app uses HTTP as the transport layer. The HTTP protocol itself is stateless, so there must be a way to associate a user's subsequent HTTP requests with that user. Otherwise, the user's log in credentials would have to be sent with every request. Also, both the server and client need to keep track of user data (e.g., the user's privileges or role). This can be done in two different ways:

- With *stateful* authentication, a unique session id is generated when the user logs in. In subsequent requests, this session ID serves as a reference to the user details stored on the server. The session ID is *opaque*; it doesn't contain any user data.
- With *stateless* authentication, all user-identifying information is stored in a client-side token. The token can be passed to any server or micro service, eliminating the need to maintain session state on the server. Stateless authentication is often factored out to an authorization server, which produces, signs, and optionally encrypts the token upon user login.

Web applications commonly use stateful authentication with a random session ID that is stored in a client-side cookie. Although mobile apps sometimes use stateful sessions in a similar fashion, stateless token-based approaches are becoming popular for a variety of reasons:

- They improve scalability and performance by eliminating the need to store session state on the server.
- Tokens enable developers to decouple authentication from the app. Tokens can be generated by an authentication server, and the authentication scheme can be changed seamlessly.

As a mobile security tester, you should be familiar with both types of authentication.

Stateful Authentication

Stateful (or "session-based") authentication is characterized by authentication records on both the client and server. The authentication flow is as follows:

1. 1.

The app sends a request with the user's credentials to the backend server.

2. 2.

The server verifies the credentials. If the credentials are valid, the server creates a new session along with a random session ID.

3. 3.

The server sends to the client a response that includes the session ID.

4. 4.

The client sends the session ID with all subsequent requests. The server validates the session ID and retrieves the associated session record.

5. 5.

After the user logs out, the server-side session record is destroyed and the client discards the session ID.

When sessions are improperly managed, they are vulnerable to a variety of attacks that may compromise the session of a legitimate user, allowing the attacker to impersonate the user. This may result in lost data, compromised confidentiality, and illegitimate actions.

Best Practices:

Locate any server-side endpoints that provide sensitive information or functions and verify the consistent enforcement of authorization. The backend service must verify the user's session ID or token and make sure that the user has sufficient privileges to access the resource. If the session ID or token is missing or invalid, the request must be rejected.

Make sure that:

- Session IDs are randomly generated on the server side.
- The IDs can't be guessed easily (use proper length and entropy).
- Session IDs are always exchanged over secure connections (e.g. HTTPS).
- The mobile app doesn't save session IDs in permanent storage.
- The server verifies the session whenever a user tries to access privileged application elements (a session ID must be valid and must correspond to the proper authorization level).
- The session is terminated on the server side and session information deleted within the mobile app after it times out or the user logs out.

Authentication shouldn't be implemented from scratch but built on top of proven frameworks. Many popular frameworks provide ready-made authentication and session management functionality. If the app uses framework APIs for authentication, check the framework security documentation for best practices. Security guides for common frameworks are available at the following links:

- Spring (Java)
- Struts (Java)
- Laravel (PHP)
- Ruby on Rails
- ASP.Net

A great resource for testing server-side authentication is the OWASP Web Testing Guide, specifically the Testing Authentication and Testing Session Management chapters.

Stateless Authentication

Token-based authentication is implemented by sending a signed token (verified by the server) with each HTTP request. The most commonly used token format is the JSON Web Token, defined in RFC7519. A JWT may encode the complete session state as a JSON object. Therefore, the server doesn't have to store any session data or authentication information.

JWT tokens consist of three Base64Url-encoded parts separated by dots. The Token structure is as follows:

base64UrlEncode(header).base64UrlEncode(payload).base64UrlEncode(signature)

The following example shows a Base64Url-encoded JSON Web Token:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva

G4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

The *header* typically consists of two parts: the token type, which is JWT, and the hashing algorithm being used to compute the signature. In the example above, the header decodes as follows:

{"alg":"HS256","typ":"JWT"}

The second part of the token is the *payload*, which contains so-called claims. Claims are statements about an entity (typically, the user) and additional metadata. For example:

{"sub":"1234567890","name":"John Doe","admin":true}

The signature is created by applying the algorithm specified in the JWT header to the encoded header, encoded payload, and a secret value. For example, when using the HMAC SHA256 algorithm the signature is created in the following way:

HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)

Note that the secret is shared between the authentication server and the backend service - the client does not know it. This proves that the token was obtained from a legitimate authentication service. It also prevents the client from tampering with the claims contained in the token.

Best Practices:

Verify that the implementation adheres to JWT best practices:

- Verify that the HMAC is checked for all incoming requests containing a token.
- Verify that the private signing key or HMAC secret key is never shared with the client. It should be available for the issuer and verifier only.
- Verify that no sensitive data, such as personal identifiable information, is embedded in the JWT. For example, by decoding the base64-encoded JWT and find out what kind of data it transmits and whether that data is encrypted. If, for some reason, the architecture requires transmission of such information in the token, make sure that payload encryption is being applied. See the sample Java implementation on the OWASP JWT Cheat Sheet.

- Make sure that replay attacks are addressed with the jti (JWT ID) claim, which gives the JWT a unique identifier.
- Make sure that cross service relay attacks are addressed with the aud (audience) claim, which defines for which application the token is entitled.
- Verify that tokens are stored securely on the mobile phone, with, for example, KeyChain (iOS) or KeyStore (Android).
- Verify that the hashing algorithm is enforced. A common attack includes altering the token to use an empty signature (e.g., signature = "") and set the signing algorithm to none, indicating that "the integrity of the token has already been verified". Some libraries might treat tokens signed with the none algorithm as if they were valid tokens with verified signatures, so the application will trust altered token claims.
- Verify that tokens include an "exp" expiration claim and the backend doesn't process expired tokens. A common method of granting tokens combines access tokens and refresh tokens. When the user logs in, the backend service issues a short-lived *access token* and a long-lived *refresh token*. The application can then use the refresh token to obtain a new access token, if the access token expires.

There are two different Burp Plugins that can help you for testing the vulnerabilities listed above:

- JSON Web Token Attacker
- JSON Web Tokens

Also, make sure to check out the OWASP JWT Cheat Sheet for additional information.

OAuth 2.0

OAuth 2.0 is an authorization framework that enables third-party applications to obtain limited access to user accounts on remote HTTP services such as APIs and web-enabled applications.

Common uses for OAuth2 include:

- Getting permission from the user to access an online service using their account.
- Authenticating to an online service on behalf of the user.
- Handling authentication errors.

According to OAuth 2.0, a mobile client seeking access to a user's resources must first ask the user to authenticate against an *authentication server*. With the users' approval, the authorization server then issues a token that allows the app to act on behalf of the user. Note that the OAuth2 specification doesn't define any particular kind of authentication or access token format.

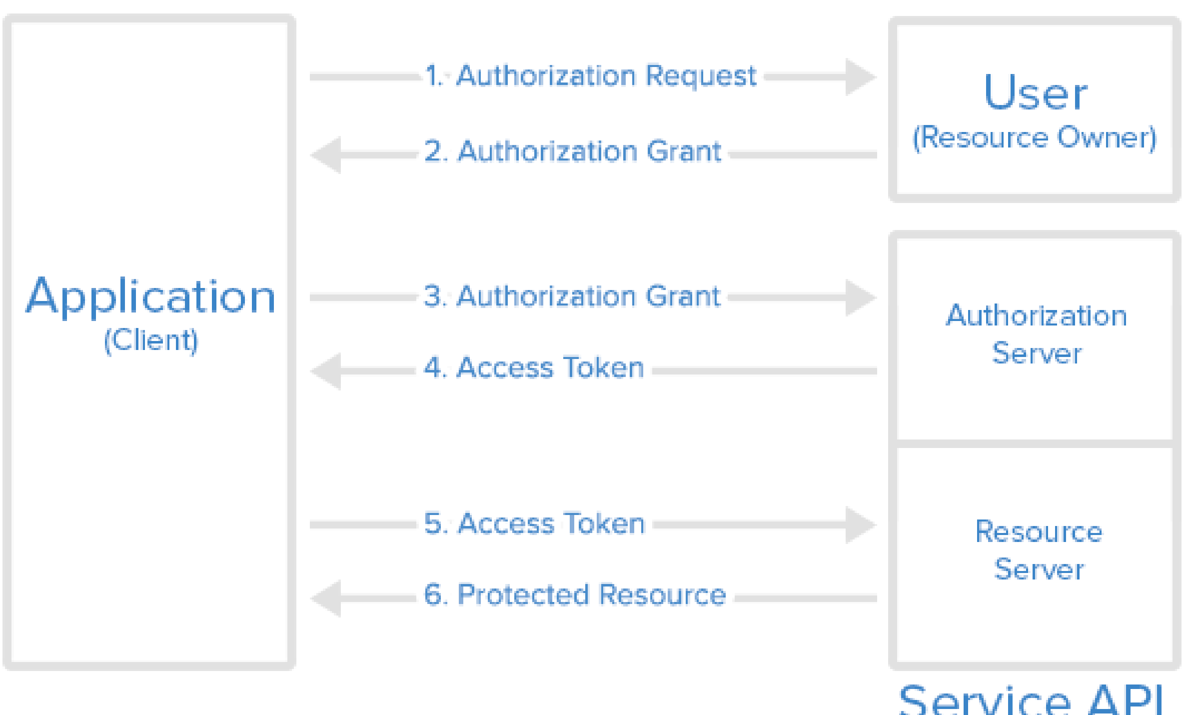
Protocol Overview

OAuth 2.0 defines four roles:

- Resource Owner: the account owner
- Client: the application that wants to access the user's account with the access tokens
- Resource Server: hosts the user accounts

- Authorization Server: verifies user identity and issues access tokens to the application

Note: The API fulfills both the Resource Owner and Authorization Server roles. Therefore, we will refer to both as the API.



Here is a more detailed explanation of the steps in the diagram:

1. 1.

The application requests user authorization to access service resources.

2. 2.

If the user authorizes the request, the application receives an authorization grant. The authorization grant may take several forms (explicit, implicit, etc.).

3. 3.

The application requests an access token from the authorization server (API) by presenting authentication of its own identity along with the authorization grant.

4. 4.

If the application identity is authenticated and the authorization grant is valid, the authorization server (API) issues an access token to the application, completing the authorization process. The access token may have a companion refresh token.

5. 5.

The application requests the resource from the resource server (API) and presents the access token for authentication. The access token may be used in several ways (e.g., as a bearer token).

6. 6.

If the access token is valid, the resource server (API) serves the resource to the application.

In OAuth2, the *user agent* is the entity that performs the authentication. OAuth2 authentication can be performed either through an external user agent (e.g. Chrome or Safari) or in the app itself (e.g. through a WebView embedded into the app or an authentication library). None of the two modes is intrinsically "better" than the other. The choice depends on the app's specific use case and threat model.

External User Agent: Using an *external user agent* is the method of choice for apps that need to interact with social media accounts (Facebook, Twitter, etc.). Advantages of this method include:

- The user's credentials are never directly exposed to the app. This guarantees that the app cannot obtain the credentials during the login process ("credential phishing").
- Almost no authentication logic must be added to the app itself, preventing coding errors.

On the negative side, there is no way to control the behavior of the browser (e.g. to activate certificate pinning).

Embedded User Agent: Using an *embedded user agent* is the method of choice for apps that need to operate within a closed ecosystem, for example to interact with corporate accounts. For example, consider a banking app that uses OAuth2 to retrieve an access token from the bank's authentication server, which is then used to access a number of micro services. In that case, credential phishing is not a viable scenario. It is likely preferable to keep the authentication process in the (hopefully) carefully secured banking app, instead of placing trust on external components.

Best Practices

For additional best practices and detailed information please refer to the following source documents:

- RFC6749 - The OAuth 2.0 Authorization Framework (October 2012)
- RFC8252 - OAuth 2.0 for Native Apps (October 2017)
- RFC6819 - OAuth 2.0 Threat Model and Security Considerations (January 2013)

Some of the best practices include but are not limited to:

- **User agent:**
  - The user should have a way to visually verify trust (e.g., Transport Layer Security (TLS) confirmation, website mechanisms).
  - To prevent man-in-the-middle attacks, the client should validate the server's fully qualified domain name with the public key the server presented when the connection was established.
- **Type of grant:**

- On native apps, code grant should be used instead of implicit grant.
- When using code grant, PKCE (Proof Key for Code Exchange) should be implemented to protect the code grant. Make sure that the server also implements it.
- The auth "code" should be short-lived and used immediately after it is received. Verify that auth codes only reside on transient memory and aren't stored or logged.

- **Client secrets:**
  - Shared secrets should not be used to prove the client's identity because the client could be impersonated ("client_id" already serves as proof). If they do use client secrets, be sure that they are stored in secure local storage.
- **End-User credentials:**
  - Secure the transmission of end-user credentials with a transport-layer method, such as TLS.
- **Tokens:**
  - Keep access tokens in transient memory.
  - Access tokens must be transmitted over an encrypted connection.
  - Reduce the scope and duration of access tokens when end-to-end confidentiality can't be guaranteed or the token provides access to sensitive information or transactions.
  - Remember that an attacker who has stolen tokens can access their scope and all resources associated with them if the app uses access tokens as bearer tokens with no other way to identify the client.
  - Store refresh tokens in secure local storage; they are long-term credentials.

User Logout

Failing to destroy the server-side session is one of the most common logout functionality implementation errors. This error keeps the session or token alive, even after the user logs out of the application. An attacker who gets valid authentication information can continue to use it and hijack a user's account.

Many mobile apps don't automatically log users out. There can be various reasons, such as: because it is inconvenient for customers, or because of decisions made when implementing stateless authentication. The application should still have a logout function, and it should be implemented according to best practices, destroying all locally stored tokens or session identifiers.

If session information is stored on the server, it should be destroyed by sending a logout request to that server. In case of a high-risk application, tokens should be invalidated. Not removing tokens or session identifiers can result in unauthorized access to the application in case the tokens are leaked. Note that other sensitive types of information should be removed as well, as any information that is not properly cleared may be leaked later, for example during a device backup.

Here are different examples of session termination for proper server-side logout:

- Spring (Java)

- Ruby on Rails
- PHP

If access and refresh tokens are used with stateless authentication, they should be deleted from the mobile device. The refresh token should be invalidated on the server.

The OWASP Web Testing Guide (WSTG-SESS-06) includes a detailed explanation and more test cases.

Supplementary Authentication

Authentication schemes are sometimes supplemented by passive contextual authentication, which can incorporate:

- Geolocation
- IP address
- Time of day
- The device being used

Ideally, in such a system the user's context is compared to previously recorded data to identify anomalies that might indicate account abuse or potential fraud. This process is transparent to the user, but can become a powerful deterrent to attackers.

Two-factor Authentication

Two-factor authentication (2FA) is standard for apps that allow users to access sensitive functions and data. Common implementations use a password for the first factor and any of the following as the second factor:

- One-time password via SMS (SMS-OTP)
- One-time code via phone call
- Hardware or software token
- Push notifications in combination with PKI and local authentication

Whatever option is used, it always must be enforced and verified on the server-side and never on client-side. Otherwise the 2FA can be easily bypassed within the app.

The 2FA can be performed at login or later in the user's session.

For example, after logging in to a banking app with a username and PIN, the user is authorized to perform non-sensitive tasks. Once the user attempts to execute a bank transfer, the second factor ("step-up authentication") must be presented.

Best Practices:

- Don't roll your own 2FA: There are various two-factor authentication mechanisms available which can range from third-party libraries, usage of external apps to self implemented checks by the developers.
- Use short-lived OTPs: A OTP should be valid for only a certain amount of time (usually 30 seconds) and after keying in the OTP wrongly several times (usually 3 times) the provided

OTP should be invalidated and the user should be redirected to the landing page or logged out.

- Store tokens securely: To prevent these kind of attacks, the application should always verify some kind of user token or other dynamic information related to the user that was previously securely stored (e.g. in the Keychain/KeyStore).

SMS-OTP

Although one-time passwords (OTP) sent via SMS are a common second factor for two-factor authentication, this method has its shortcomings. In 2016, NIST suggested: "Due to the risk that SMS messages may be intercepted or redirected, implementers of new systems SHOULD carefully consider alternative authenticators.". Below you will find a list of some related threats and suggestions to avoid successful attacks on SMS-OTP.

Threats:

- Wireless Interception: The adversary can intercept SMS messages by abusing femtocells and other known vulnerabilities in the telecommunications network.
- Trojans: Installed malicious applications with access to text messages may forward the OTP to another number or backend.
- SIM SWAP Attack: In this attack, the adversary calls the phone company, or works for them, and has the victim's number moved to a SIM card owned by the adversary. If successful, the adversary can see the SMS messages which are sent to the victim's phone number. This includes the messages used in the two-factor authentication.
- Verification Code Forwarding Attack: This social engineering attack relies on the trust the users have in the company providing the OTP. In this attack, the user receives a code and is later asked to relay that code using the same means in which it received the information.
- Voicemail: Some two-factor authentication schemes allow the OTP to be sent through a phone call when SMS is no longer preferred or available. Many of these calls, if not answered, send the information to voicemail. If an attacker was able to gain access to the voicemail, they could also use the OTP to gain access to a user's account.

You can find below several suggestions to reduce the likelihood of exploitation when using SMS for OTP:

- **Messaging**: When sending an OTP via SMS, be sure to include a message that lets the user know 1) what to do if they did not request the code 2) your company will never call or text them requesting that they relay their password or code.
- **Dedicated Channel**: When using the OS push notification feature (APN on iOS and FCM on Android), OTPs can be sent securely to a registered application. This information is, compared to SMS, not accessible by other applications. Alternatively of a OTP the push notification could trigger a pop-up to approve the requested access.
- **Entropy**: Use authenticators with high entropy to make OTPs harder to crack or guess and use at least 6 digits. Make sure that digits are separates in smaller groups in case people have to remember them to copy them to your app.

- **Avoid Voicemail**: If a user prefers to receive a phone call, do not leave the OTP information as a voicemail.

SMS-OTP Research:

- [#dmitrienko] Dmitrienko, Alexandra, et al. "On the (in) security of mobile two-factor authentication." International Conference on Financial Cryptography and Data Security. Springer, Berlin, Heidelberg, 2014.
- [#grassi] Grassi, Paul A., et al. Digital identity guidelines: Authentication and lifecycle management (DRAFT). No. Special Publication (NIST SP)-800-63B. 2016.
- [#grassi2] Grassi, Paul A., et al. Digital identity guidelines: Authentication and lifecycle management. No. Special Publication (NIST SP)-800-63B. 2017.
- [#konoth] Konoth, Radhesh Krishnan, Victor van der Veen, and Herbert Bos. "How anywhere computing just killed your phone-based two-factor authentication." International Conference on Financial Cryptography and Data Security. Springer, Berlin, Heidelberg, 2016.
- [#mulliner] Mulliner, Collin, et al. "SMS-based one-time passwords: attacks and defense." International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, Berlin, Heidelberg, 2013.
- [#siadati] Siadati, Hossein, et al. "Mind your SMSes: Mitigating social engineering in second factor authentication." Computers & Security 65 (2017): 14-28.
- [#siadati2] Siadati, Hossein, Toan Nguyen, and Nasir Memon. "Verification code forwarding attack (short paper)." International Conference on Passwords. Springer, Cham, 2015.

Transaction Signing with Push Notifications and PKI

Another alternative and strong mechanisms to implement a second factor is transaction signing.

Transaction signing requires authentication of the user's approval of critical transactions. Asymmetric cryptography is the best way to implement transaction signing. The app will generate a public/private key pair when the user signs up, then registers the public key on the backend. The private key is securely stored in the KeyStore (Android) or KeyChain (iOS). To authorize a transaction, the backend sends the mobile app a push notification containing the transaction data. The user is then asked to confirm or deny the transaction. After confirmation, the user is prompted to unlock the Keychain (by entering the PIN or fingerprint), and the data is signed with user's private key. The signed transaction is then sent to the server, which verifies the signature with the user's public key.

Login Activity and Device Blocking

It is a best practice that apps should inform the user about all login activities within the app with the possibility of blocking certain devices. This can be broken down into various scenarios:

1. 1.

The application provides a push notification the moment their account is used on another device to notify the user of different activities. The user can then block this device after opening the app via the push-notification.

2. 2.

The application provides an overview of the last session after login. If the previous session was with a different configuration (e.g. location, device, app-version) compared to the current configuration, then the user should have the option to report suspicious activities and block devices used in the previous session.

3. 3.

The application provides an overview of the last session after login at all times.

4. 4.

The application has a self-service portal in which the user can see an audit-log. This allows the user to manage the different devices that are logged in.

The developer can make use of specific meta-information and associate it to each different activity or event within the application. This will make it easier for the user to spot suspicious behavior and block the corresponding device. The meta-information may include:

- Device: The user can clearly identify all devices where the app is being used.
- Date and Time: The user can clearly see the latest date and time when the app was used.
- Location: The user can clearly identify the latest locations where the app was used.

The application can provide a list of activities history which will be updated after each sensitive activity within the application. The choice of which activities to audit needs to be done for each application based on the data it handles and the level of security risk the team is willing to have. Below is a list of common sensitive activities that are usually audited:

- Login attempts
- Password changes
- Personal Identifiable Information changes (name, email address, telephone number, etc.)
- Sensitive activities (purchase, accessing important resources, etc.)
- Consent to Terms and Conditions clauses

Paid content requires special care, and additional meta-information (e.g., operation cost, credit, etc.) might be used to ensure user's knowledge about the whole operation's parameters.

In addition, non-repudiation mechanisms should be applied to sensitive transactions (e.g. paid content access, given consent to Terms and Conditions clauses, etc.) in order to prove that a specific transaction was in fact performed (integrity) and by whom (authentication).

Lastly, it should be possible for the user to log out specific open sessions and in some cases it might be interesting to fully block certain devices using a device identifier.

The Developer’s Guide to Mobile Authentication

What is Mobile Authentication?

Mobile authentication is a security method to verify a user’s identity through mobile devices and mobile apps. It caters to one or more authentication methods to provide secure access to any particular app, resource, or service.

Let’s look at the various mobile authentication methods developers can utilize depending on their business use case.

Mobile Authentication Methods

Password-based Authentication

Email-Password and Username-Password are common types of password-based authentication. While utilizing these methods, developers should consider setting secure and robust password policies in their authentication mechanism, such as:

- Mandatory use of symbols and numbers
- Restricting the use of common passwords
- Blocking the use of profile information in passwords

These measures ensure better quality passwords and prevent user accounts from brute force and dictionary password attacks.

Limitation: Passwords are hard to remember, and typing in passwords on a small mobile screen degrades the user experience. Hence, developers must use authentication that does not compromise the security postures yet provide an appropriate user experience.

Patterns and Digit-based Authentication

The user must set a pattern or a digit-based PIN (typically 4 or 6 digits). Developers can utilize this as an authentication factor for their mobile application, as this authentication method is faster and more comfortable than entering passwords on a mobile screen.

Limitation: Both patterns and 4 or 6 digits PINs are limited. Also, users tend to use simple patterns and PINs like L or S patterns and 1234, 987654, date of birth as their password.

OTP-based Login

Users use an OTP received via SMS or email to authenticate themself. Thus, users do not have to remember a password, pattern, or PIN to access their account. At the same time, developers don’t have to implement password-based security mechanisms.

Biometric Authentication

Biometric authentication uses unique biological traits of users for mobile authentication. Some common examples of biometric authentication are fingerprint scanning, face unlocks, retina scans, and vocal cadence.

Developers can implement pre-coded libraries and modules to enable authentication through mobile components like the finger scanner, camera (for facial recognition), and microphone (for voice-based identification).

Social Login

It acts as a single sign-on authentication mechanism. Developers can implement this in mobile apps to use users’ login tokens from other social networking sites to allow access to the app.

Also, with social login, developers don't need to worry about storing passwords securely and managing the password recovery option. It helps the user sign in to the mobile app without creating a separate account from within the app, hence increasing the user experience (UX).

User Interface (UI) and User Experience (UX) in Mobile Authentication

Login and registration screens are a gateway to your mobile applications; if they are a hassle, the user might not bother using the application. Thus, developers should pay a lot of attention to these screens regarding user experience and usage.

Here are some quick tips for mobile authentication screens:

- **Simple Registration Process:** Lengthy registration forms are a big no-no. Brainstorm essential information for creating an account via mobile application and only include those fields.
- **External or Social Login:** Allow users to log in via external or social accounts. This way, users don't have to remember another password or credentials for your app.
- **Facilitate Resetting:** Include forget password on the login screen for good visibility and reach if the app provides password-based login. Also, setting the new password should be seamless and fast.
- **Keep Users Logged In:** Not logging out users on app close is helpful in a good experience. However, this depends on the type of app you offer. Developers should include MFA for better security if the app stores sensitive information or skip the stay logged-in feature altogether.
- **Meaningful Error Messages:** Errors and how they are handled directly impact user experience. Thus, developers should keep error messages meaningful and clearly state what went wrong and how to fix it.

Tip: Customize the mobile app keyboard for the type of input field. For example – display a numeric keyboard when asking for a PIN and include @ button when asking for an email address.

Conclusion

Considering the above points would result in a great and secure user experience for your mobile app users. However, if you feel executing these guidelines would take ample time, be informed that CIAM solutions are available in the market to handle all these requirements for you.

Secure API Development Best Practices

We can't imagine our business without digital. When we send a Twitter, log into an account with our social network, make an online purchase, or check real-time traffic via an app, behind-the-scenes APIs (Application Programming Interfaces) are running.

In our world of connected devices and applications, APIs play a crucial role in facilitating communication between different systems and applications. In addition to being the connector for B2B or B2C businesses, we are talking about enabling the connection for thousands of devices for the internet of things (IoT).

APIs revolutionize the world of system integration and are major players in our Digital Transformation era. Therefore, the secure development of APIs is of utmost relevance for organizations to provide secure and reliable access to data and services. In this context, there are several security

technologies and best practices that can be applied to ensure the protection of API-based communication, in this article we highlight OAuth2 and JSON Web Token (JWT).

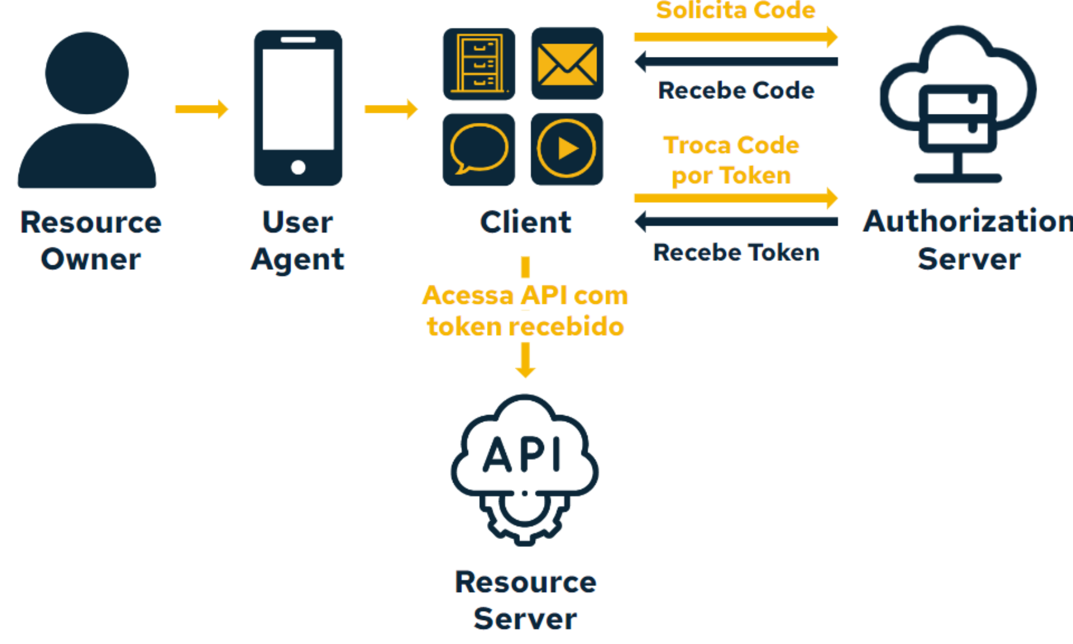
We talk about **OAuth2** because it is an open standard for authorization that delegates the process to an authorization server, allowing third-party applications to securely access user data. And we include the efficient and compact form of request, or token, **JWT** used for authentication and authorization in APIs.

We believe that awareness and incorporation of these security best practices increase the reliability, integrity of information, and availability of API services, decreasing risk and increasing the credibility and image of organizations.

OAUTH 2

Gone are the days when the access control process was based on presenting a user and password to the application. Today both authentication and authorization have their complex steps to establish the user's identity and access profile respectively.

While some APIs are authenticated with username and password, usually in the form of basic authentication in the header, this is not our recommendation. As a best practice for working with device authorization and server-to-server support, OAuth 2.0 is our recommendation, allowing for a great user experience and enforcing secure development in your API.



But what is OAuth? We stick with the definition present in the OWASP Cheat Sheets Series recommendation list on Authentication, under "Use of authentication protocols that require no password":

"Open Authorization (OAuth) is a protocol that allows an application to authenticate against a server as a user, without requiring passwords or any third party server that acts as an identity provider. It uses a token generated by the server and provides how the authorization flows most occur, so that a client, such as a mobile application, can tell the server what user is using the service."

I would like to take this opportunity to inform you that we have discussed the difference between OAuth2 and SAML in a previous article.

OAuth 2.0 Advantages

As positive points when choosing to use Oauth 2 as an authorization protocol, we can cite the simplicity of use and the comprehensiveness of the technology. It can be used for web, desktop, and mobile applications. Besides these, it is worth mentioning the advantages of:

- The user does not need to create another profile to access the application;
- Few passwords to remember;
- Time savings, since no authentication needs to be developed;
- Lower risk of identity theft since authentication occurs at the provider;
- Tokens are transmitted in encrypted form and not stored in the application;
- Fewer data to store on servers.

Threats to OAuth 2.0

It is possible for attackers to threaten Oauth flows with CSRF, XSS, and

Open Redirect. The following threats can be listed:

- Threats on the endpoint;
- Redirect Hijack;
- Man-In-The-Middle (MITM) Attack.

Examples of vulnerabilities:

- Authorization Code Injection Attack;
- OAuth Phishing via Unchecked Redirect URI;
- OAuth Client Secret Disclosure.

More information about the weaknesses can be found in RFC-6819 Threat Modeling and Security Considerations for OAuth 2.0

Security Best Practices for OAuth 2.0

Because of these possible attack scenarios, it is important to pass on the best practices of secure API development when using OAuth 2.0:

- Always validate the “redirect_uri” parameter on the server side to allow only approved URLs and thus avoid open redirect attacks;
- Always try to make the exchange with code, instead of getting the token directly (avoid “response_type=token”);
- Use the “state” parameter with a random hash to prevent CSRF attacks in the OAuth authentication process;
- Define and validate the scope for each application;
- Always use secure communication, such as HTTPS, when transmitting OAuth2 tokens and authorization codes.

JSON Web Token (JWT)

The JWT standard defines how to transmit and store JSON objects between applications securely, simply, and in a compact form. It is an open standard documented in RFC-7519. Widely used in the validation of services in Web Services, for the advantages that we will see below, already highlighting the possibility of local validation.

Knowing the structure of JWT we see that it is a JSON structure composed of a Header, Payload, and Signature.

But what is OAuth? We stick with the definition present in the OWASP Cheat Sheets Series recommendation list on Authentication, under “Use of authentication protocols that require no password”:

“Open Authorization (OAuth) is a protocol that allows an application to authenticate against a server as a user, without requiring passwords or any third party server that acts as an identity provider. It uses a token generated by the server and provides how the authorization flows most occur, so that a client, such as a mobile application, can tell the server what user is using the service.”

I would like to take this opportunity to inform you that we have discussed the difference between OAuth2 and SAML in a previous article.

OAuth 2.0 Advantages

As positive points when choosing to use Oauth 2 as an authorization protocol, we can cite the simplicity of use and the comprehensiveness of the technology. It can be used for web, desktop, and mobile applications. Besides these, it is worth mentioning the advantages of:

- The user does not need to create another profile to access the application;
- Few passwords to remember;
- Time savings, since no authentication needs to be developed;
- Lower risk of identity theft since authentication occurs at the provider;
- Tokens are transmitted in encrypted form and not stored in the application;
- Fewer data to store on servers.

Threats to OAuth 2.0

It is possible for attackers to threaten Oauth flows with CSRF, XSS, and

Open Redirect. The following threats can be listed:

- Threats on the endpoint;
- Redirect Hijack;
- Man-In-The-Middle (MITM) Attack.

Examples of vulnerabilities:

- Authorization Code Injection Attack;
- OAuth Phishing via Unchecked Redirect URI;
- OAuth Client Secret Disclosure.

More information about the weaknesses can be found in [RFC-6819 Threat Modeling and Security Considerations for OAuth 2.0]

Security Best Practices for OAuth 2.0

Because of these possible attack scenarios, it is important to pass on the best practices of secure API development when using OAuth 2.0:

- Always validate the "redirect_uri" parameter on the server side to allow only approved URLs and thus avoid open redirect attacks;
- Always try to make the exchange with code, instead of getting the token directly (avoid "response_type=token");
- Use the "state" parameter with a random hash to prevent CSRF attacks in the OAuth authentication process;
- Define and validate the scope for each application;
- Always use secure communication, such as HTTPS, when transmitting OAuth2 tokens and authorization codes.

JSON Web Token (JWT)

The JWT standard defines how to transmit and store JSON objects between applications securely, simply, and in a compact form. It is an open standard documented in RFC-7519. Widely used in the validation of services in Web Services, for the advantages that we will see below, already highlighting the possibility of local validation.

Knowing the structure of JWT we see that it is a JSON structure composed of a Header, Payload, and Signature.

Being digitally signed, JWT naturally guarantees the integrity issue. But it is worth remembering that if the data transmitted in the payload is sensitive, this will require implementing encryption in the payload to make it confidential.

More details about secure JSON Web token (JWT) implementation can be found in our blog.

JWT Advantages

That the JSON Web Token (JWT) is a compact and secure way to transmit requests, we already know. Now as advertised, let's look at other advantages of secure API development of this signed encoding mechanism:

- Because it is stateless, there is no need for session storage on the server;
- Because it is compact, it can be transmitted in the HTTP header or as a query parameter;
- Cross-domain compatibility. That is, it can be used in building decentralized cross-domain systems.

Threats to JWT

- Store sensitive information in the payload;
- Expiration time not configured;
- Expiration time too high;
- Use of weak algorithm for signature;
- Lack of use of the other reserved claims of JWT;
- Bug in some libraries;
- The algorithm "None";
- Algorithm Change;
- Entering an invalid signature;
- Brute force on weak keys.

Examples of vulnerabilities:

- JWT algorithm "None" allowed;
- JWT signature not verified;
- JWT validity not checked.

Security Best Practices for JWT

- Restrict accepted algorithms;
- Verify all tokens before processing payload data to prevent tampering and unauthorized access;
- Store JWTs tokens securely, such as in secure HTTP-only cookies, avoiding access control failures;
- Always sanitize data that the user can manipulate;
- Implement rate-limiting to prevent brute-force and denial-of-service attacks.
- Restrict URLs from any JWKS / X509 certificates;

- Use the most robust signing process for which you can spare CPU time;
- Use asymmetric keys if tokens are used on more than one server;
- Use strong and unique keys to sign JWTs and protect against brute force attacks;
- Have a short expiration time, which decreases attackers' time and prevents reuse;
- Always try to use the latest version of the JWT library you choose, and perform a search for possible CVEs to verify that the version you use has no publicly known vulnerabilities;
- Use secure protocols like HTTPS to avoid eavesdropping and man-in-the-middle attacks.

JWT is a widely used and convenient way to authenticate and authorize users in RESTful APIs. To ensure JWT security, it is important to follow best practices, such as using strong secret keys, verifying signatures, and storing tokens securely. By taking the necessary security precautions, you can reduce the risk of security incidents and protect your APIs and the sensitive data that they handle.

Authorization vs. Access Control

If authorization involves defining a policy, access control puts the policies to work. These two terms aren't interchangeable. But they do work hand in hand.

Once you've completed the authorization process, the system knows who you are and what you should see. The access control system unlocks the assets, so you can do the work you need to do.

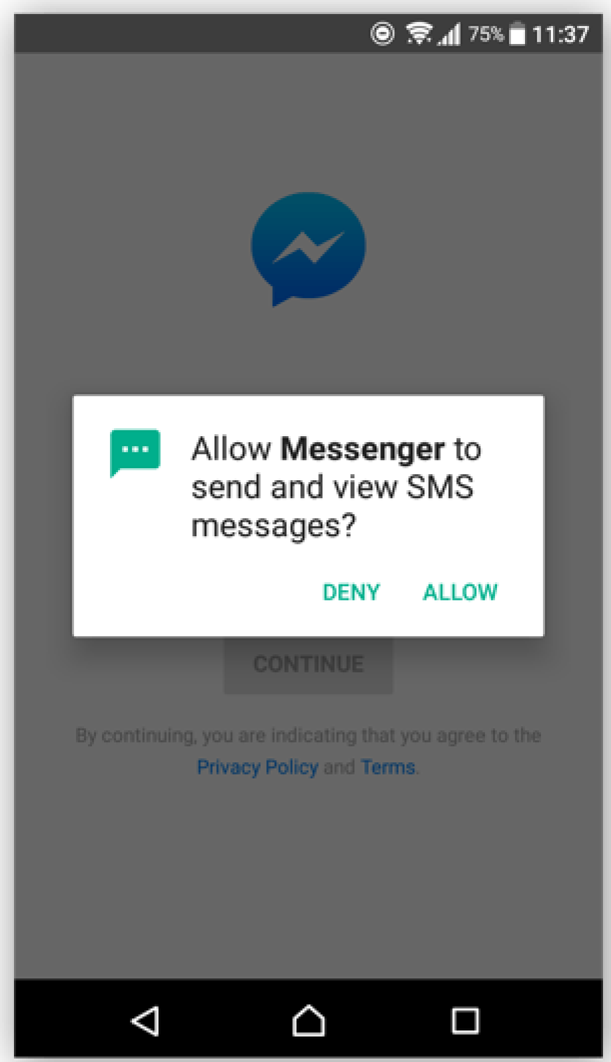
What are Android app permissions?

Android app permissions can give apps control of your phone and access to your camera, microphone, private messages, conversations, photos, and more. App permission requests pop up the first time an app needs access to sensitive hardware or data on your phone or tablet and are usually privacy-related.

Anytime you install an app from Google Play, you'll likely see an app permission request. If you install a camera app, for example, it will need your permission to access your device's camera before it can actually take photos.

Other permissions can include monitoring your location, saving data, sending and receiving calls and texts, reading sensitive log data, or accessing your contacts, calendar, or browsing history.

A typical Android app permissions request looks like this:



The familiar Android app permissions request.

Before Facebook Messenger can access your text messages, for example, you need to approve or deny the permission request.

What is the Android permissions controller?

The Android permissions controller is a part of the Android operating system that tells apps what they can and can't access. When you install a new app, the Android permissions controller is what gives you the option to allow or deny permissions for that app.

Android app permissions to avoid

You should avoid app permissions that aren't necessary for an app to work. If the app shouldn't need access to something — like your camera or location — don't allow it. Consider your privacy when deciding whether to avoid or accept an app permission request.

Android system permissions are divided between "normal" and "dangerous" permissions. Android allows "normal" permissions — such as giving apps access to the internet — by default. That's because normal permissions shouldn't pose a risk to your privacy or your device's functionality.

It's the "dangerous" permissions that Android requires your permission to use. These "dangerous" permissions include access to your calling history, private messages, location, camera, microphone, and more. These permissions are not inherently dangerous, but have the **potential for misuse**. That's why Android gives you the opportunity to accept or refuse them.

Some apps need these permissions. In those cases, check that an app is safe *before* you install it, and make sure the app comes from a reputable developer.

How to tell if an app permission is dangerous

Android classifies permissions as "dangerous" if they might affect your privacy, the functionality of other apps, or your device's operation. Watch out for apps that request access to at least one of these nine permission groups:

- Body sensors
- Calendar
- Camera
- Contacts
- GPS location
- Microphone
- Calling
- Texting
- Storage

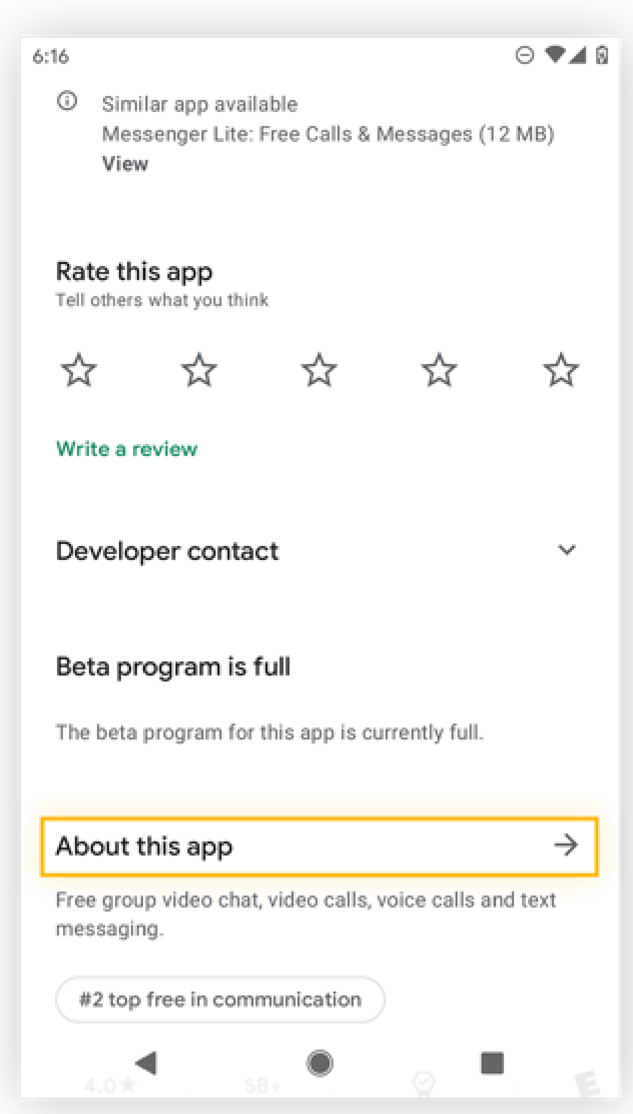
How to change Android app permissions

You can manage Android app permissions by checking which ones you currently have allowed and modifying them if necessary. You can also check Android app permissions in the Google Play store before you download an app. Here are four ways to change your app permissions on Android.

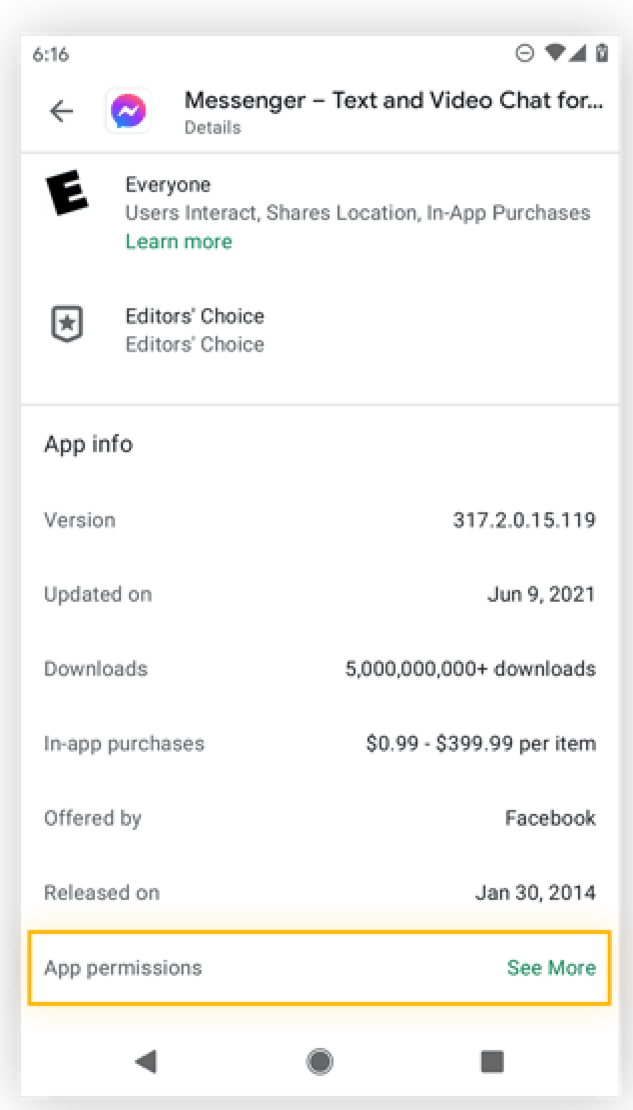
Check an app's permissions before installing it

Maintain strict privacy standards by reviewing an app's permissions before you install it. Here's how to check Android app permissions in the Google Play store:

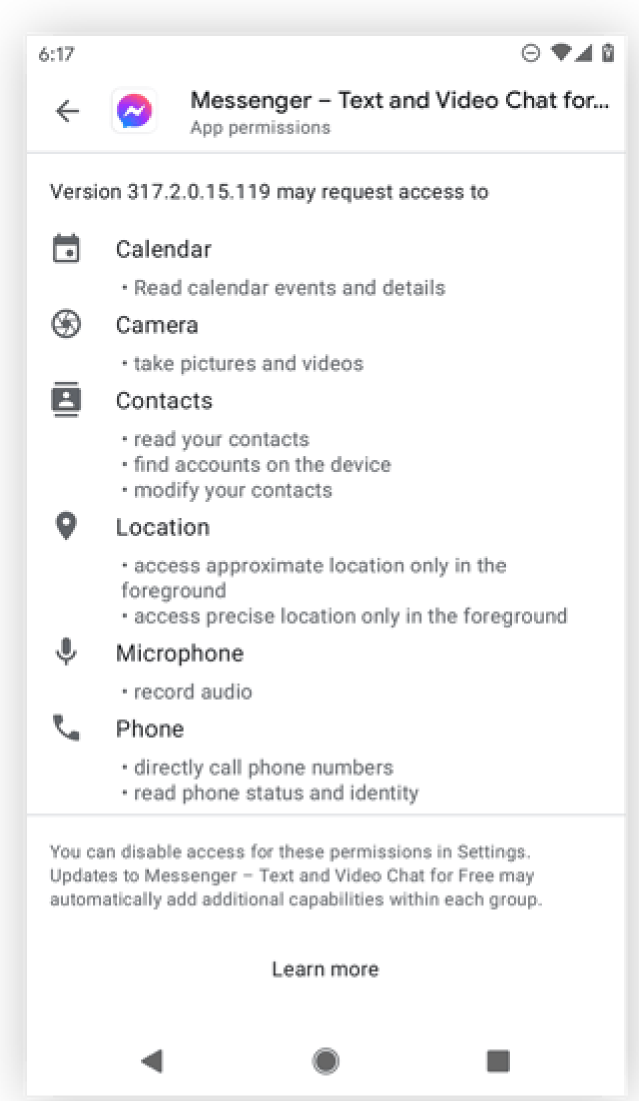
1. Open **Google Play** and find the app you're interested in.
2. Scroll down and tap **About this app**.



3. Scroll down to the bottom and tap **App permissions**.



4. Here you can see all the permissions the app will request.

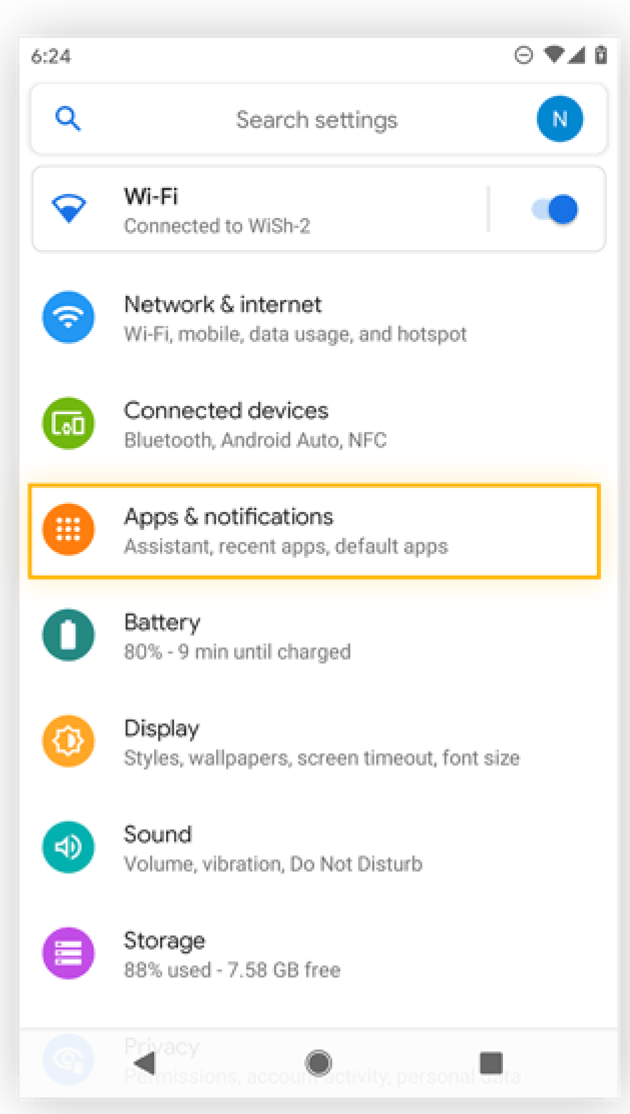


From here, you can decide whether you trust the app developer and feel comfortable with the app using these permissions. Choosing to use only apps with appropriate permissions is a great way to control Android app permissions right from the start.

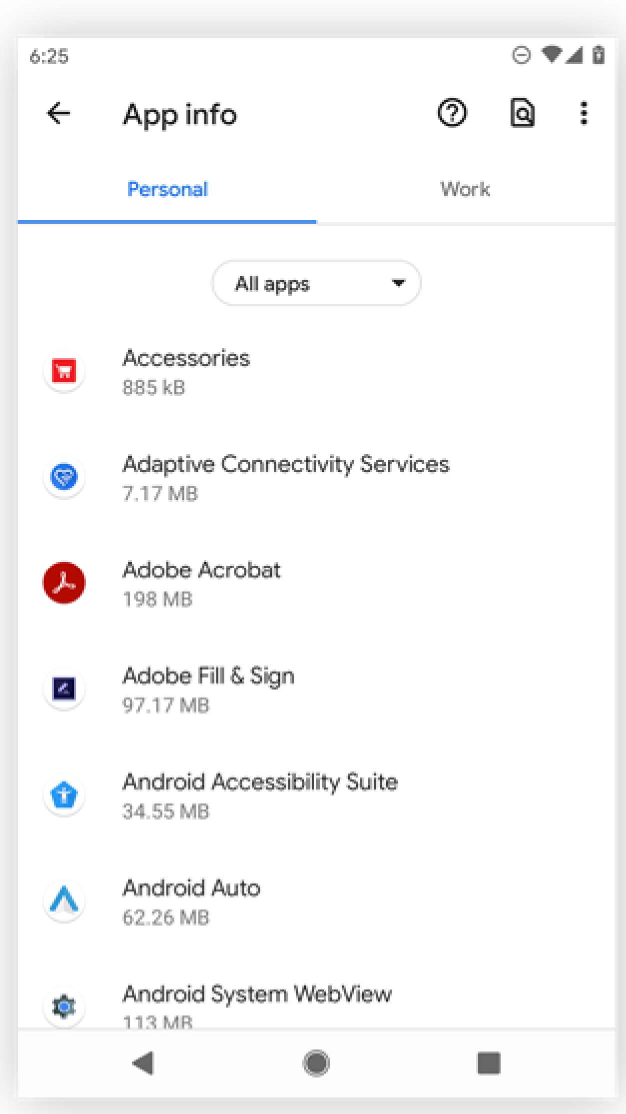
See all permissions used by a specific app

Concerned about what a particular app can access on your phone? Here's how to manage permissions on a specific app:

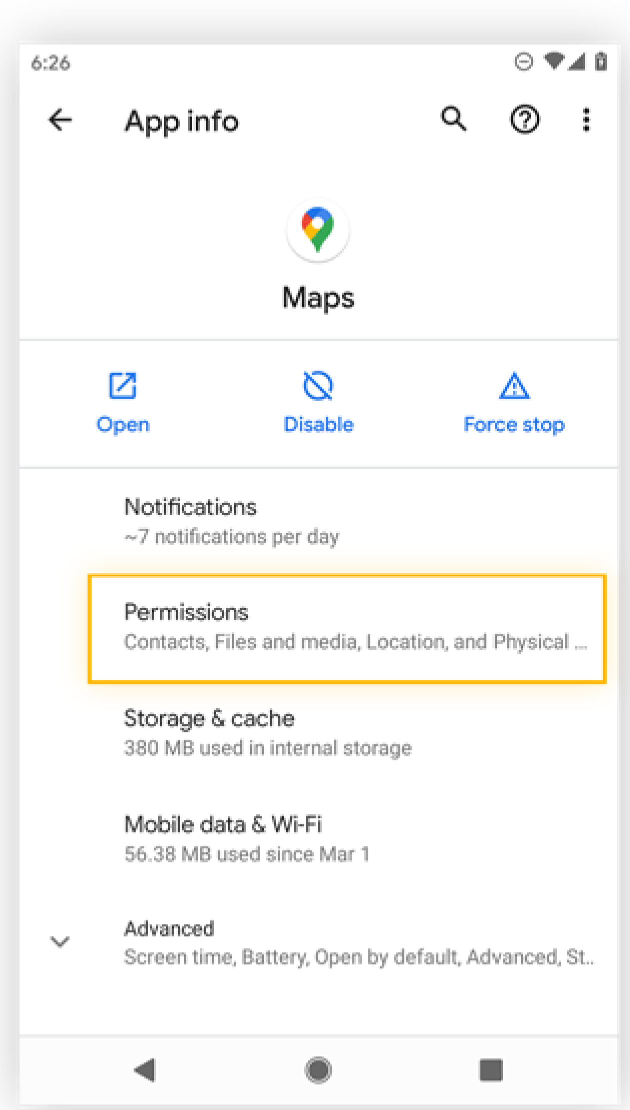
1. Open **Settings** and choose **Apps & notifications**.



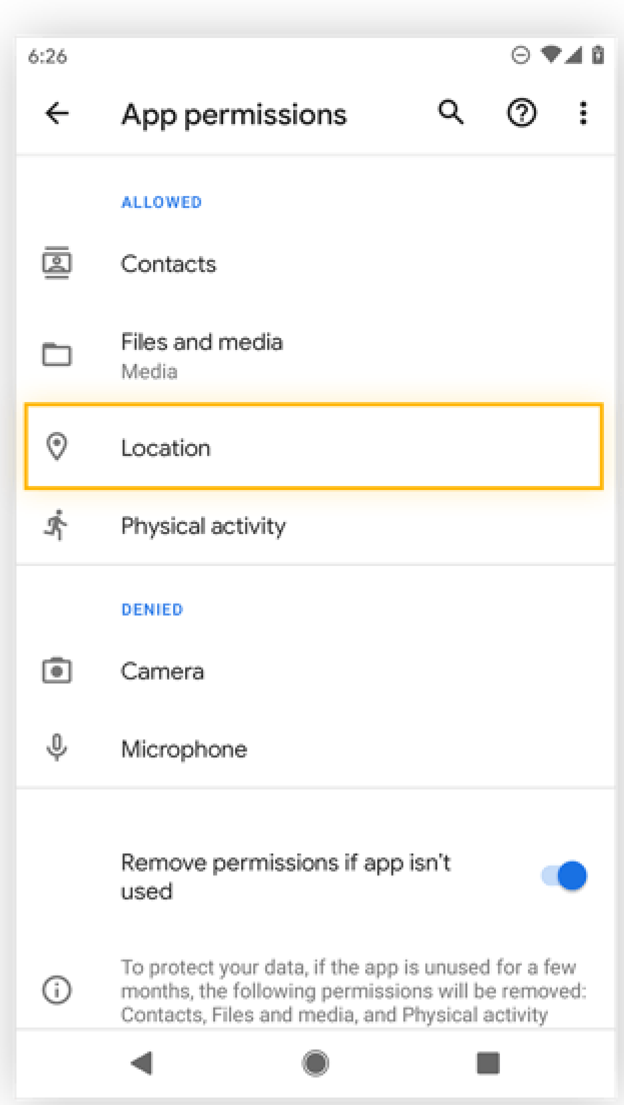
2. Find and select the app you want to check permissions for.



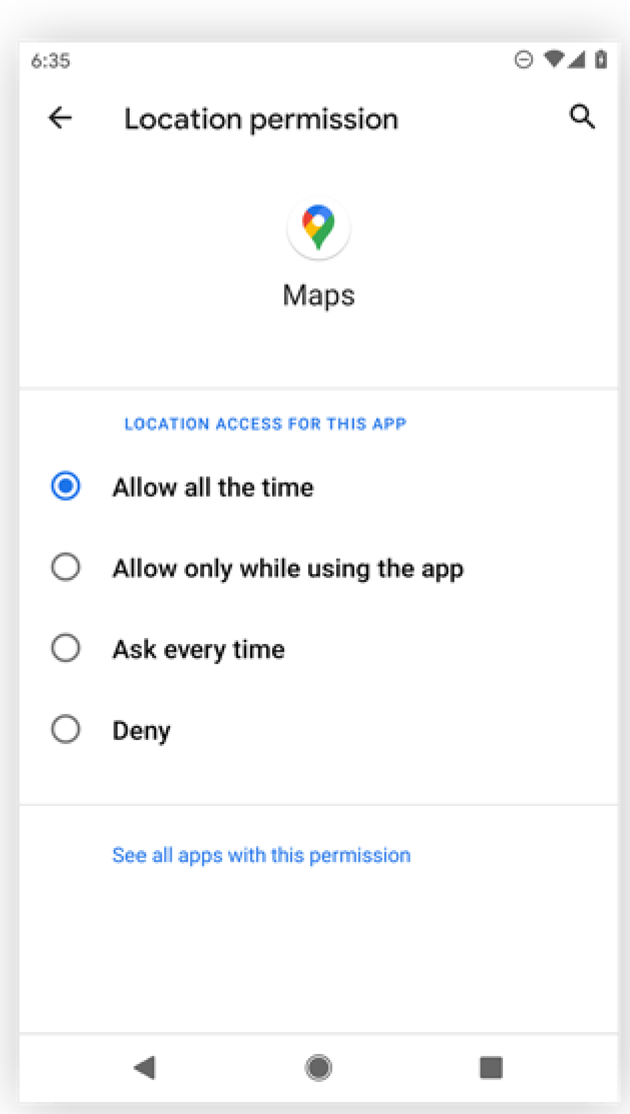
3. Tap **Permissions**.



4. Now you can see all the app's permissions. To change a specific permission, tap it.



5. Here you can delete any permissions you aren't comfortable with.



Apps *do* require some permissions to work properly. If you deny Google Maps access to your location, it can't give you directions and also won't be able to personalize your map searches based on your location.

Here you can also choose to allow permissions all the time, only when the app is in use, or only if you allow it each time.

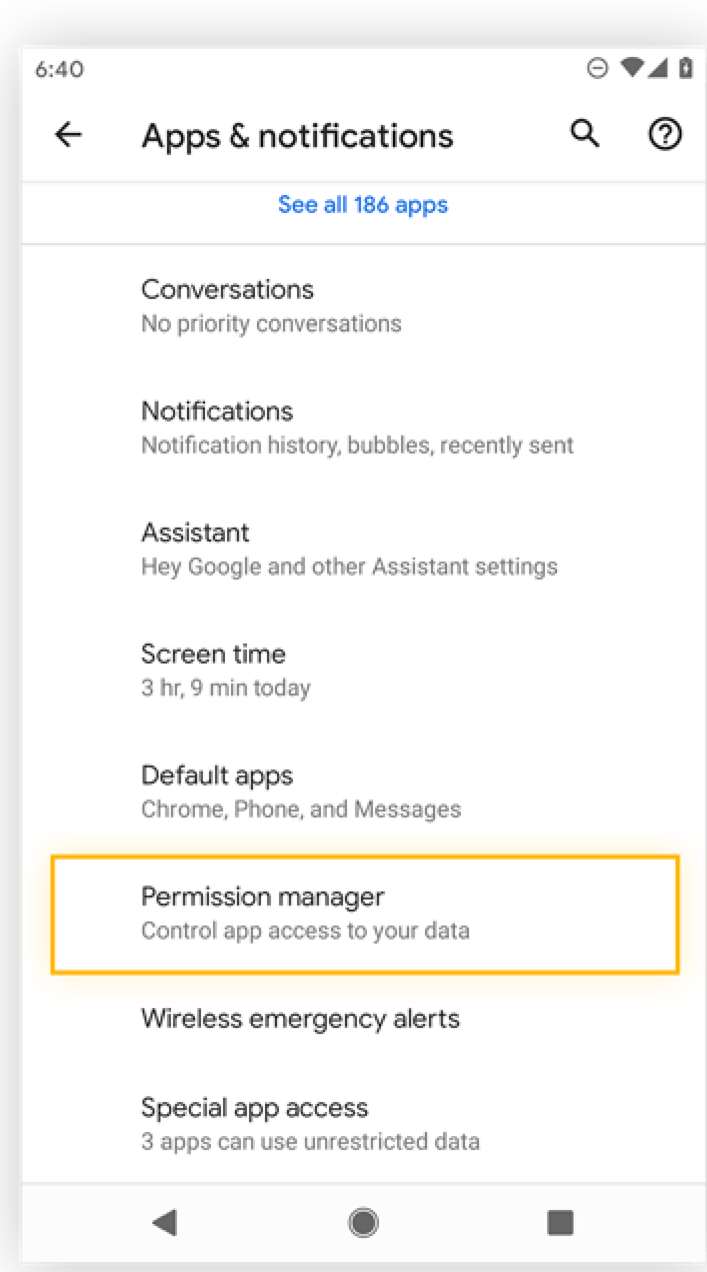
See all apps that use a specific permission

If you'd rather take a look at the Android app permissions list and choose something specific — like access to your location or contacts — and then view all apps that have that access, this can help you get control of your privacy on Android.

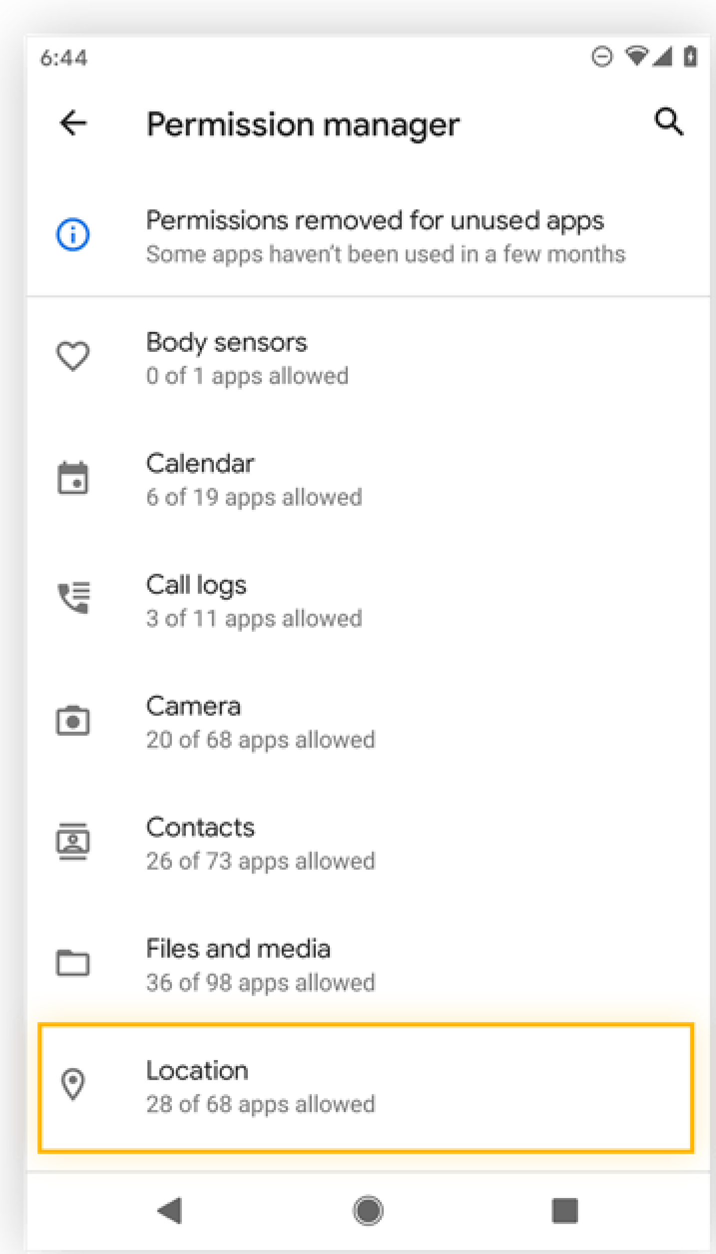
Here's how to access the app permissions list to see all apps that use a specific permission:

1. Open **Settings** and tap **Apps & notifications**.

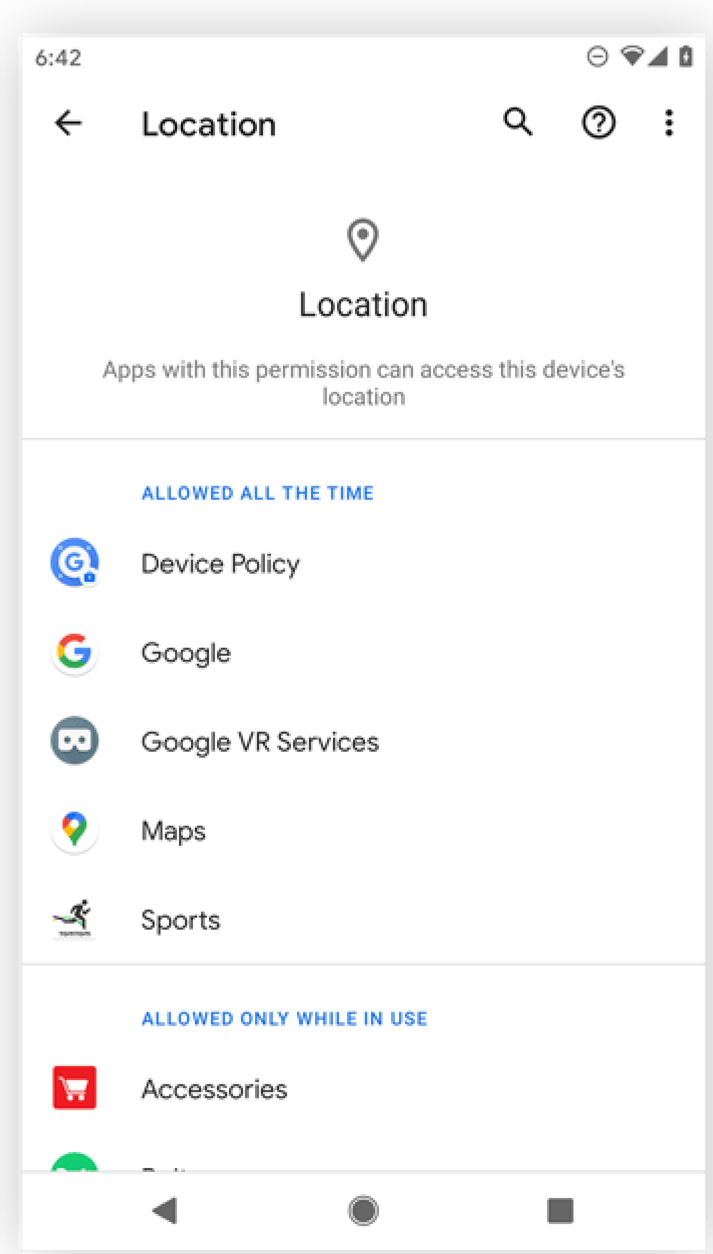
2. Tap **Permission manager** to open the Android permission controller app.



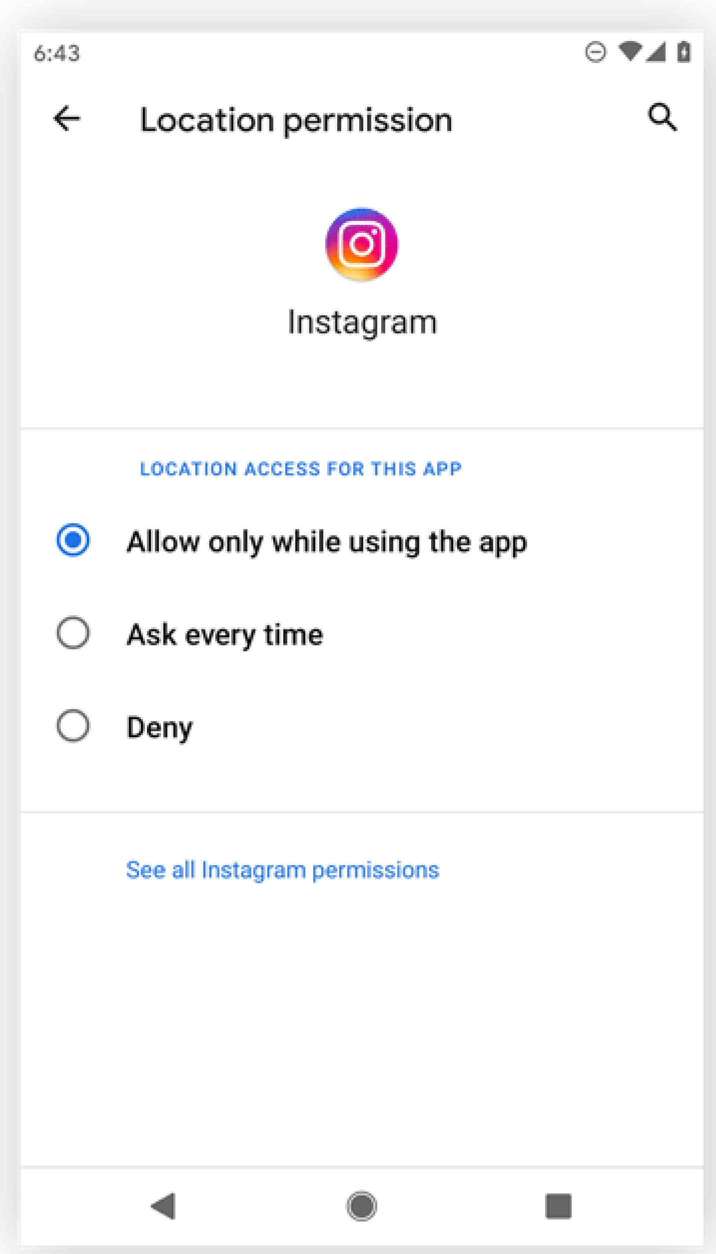
3. Click a specific permission from the app permissions list that you're interested in, like location.



4. Here you'll see apps that have access to your location all the time or only while in use. To remove access, tap a particular app.



5. Manage the Android app's permissions by choosing its level of access here.



Use a security tool to see app permissions

An easy way to manage your Android app permissions is to use a security tool to help with the process. Not only does AVG AntiVirus for Android help you take control of your Android app permissions, it also protects your phone against malware, theft, and unsafe Wi-Fi networks.

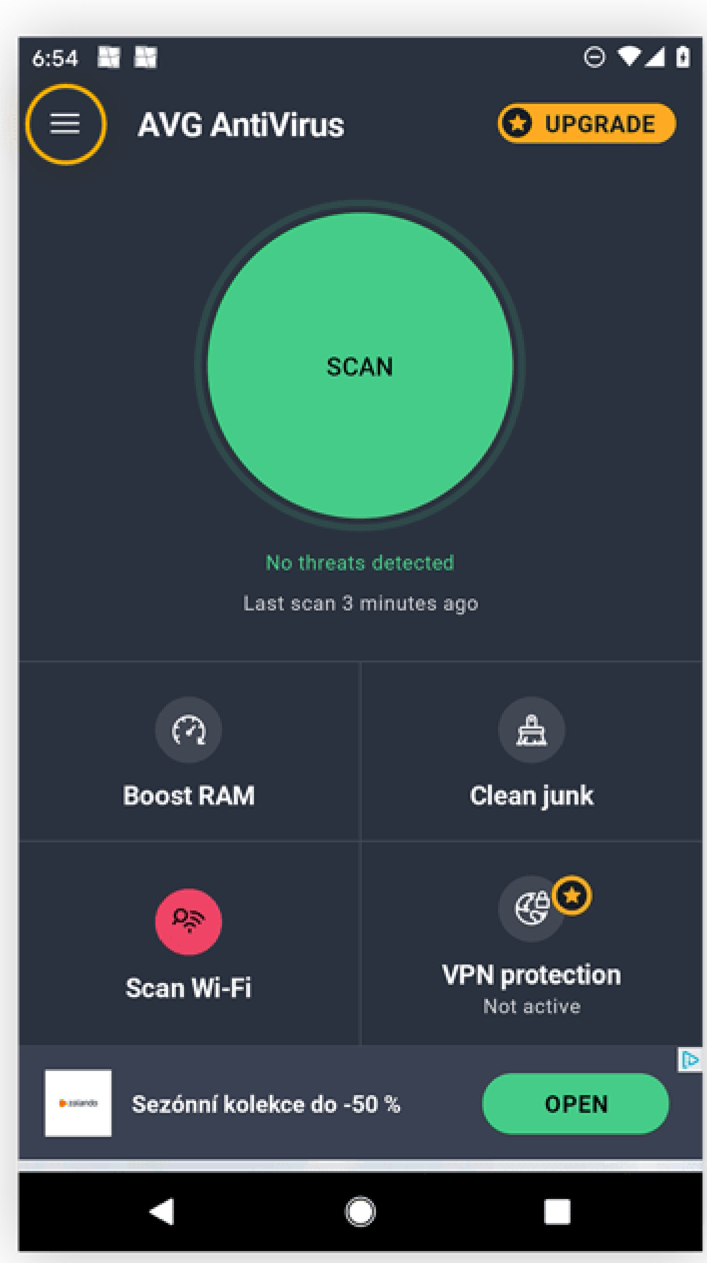
Here's how to use AVG AntiVirus to see app permissions:

1. Download and install AVG AntiVirus FREE for Android.

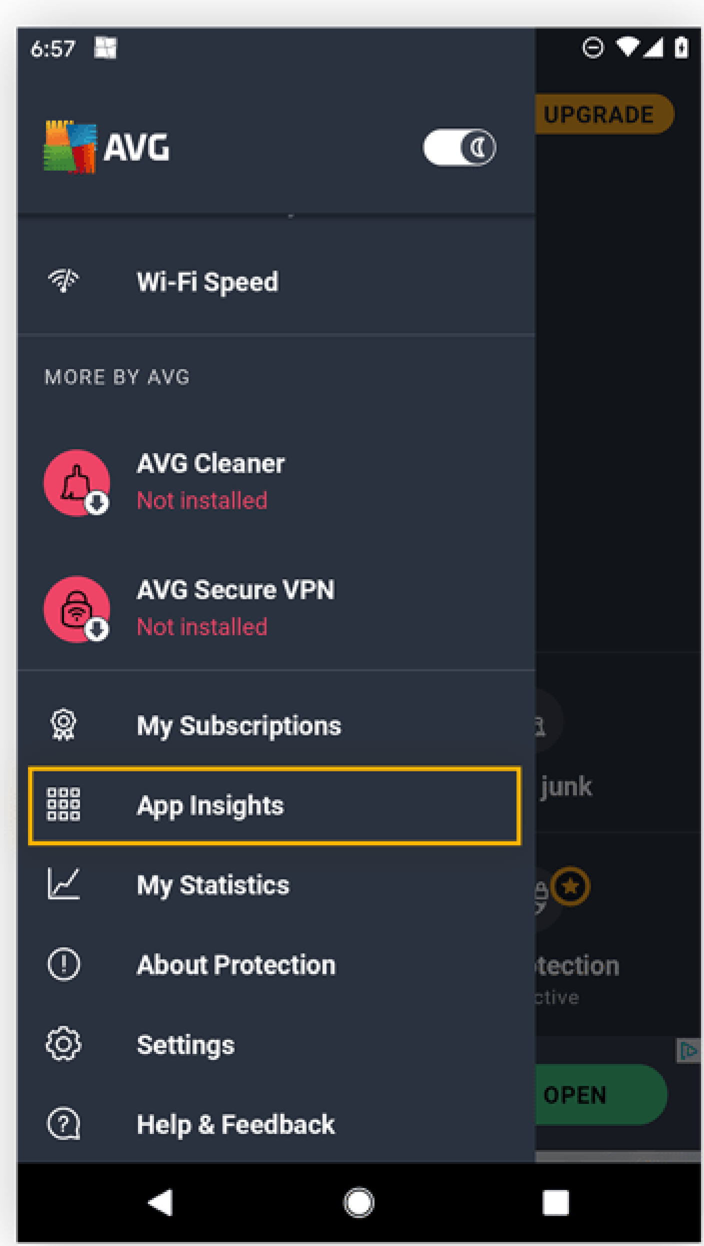
Install free AVG AntiVirus

Get it for PC, iOS, Mac

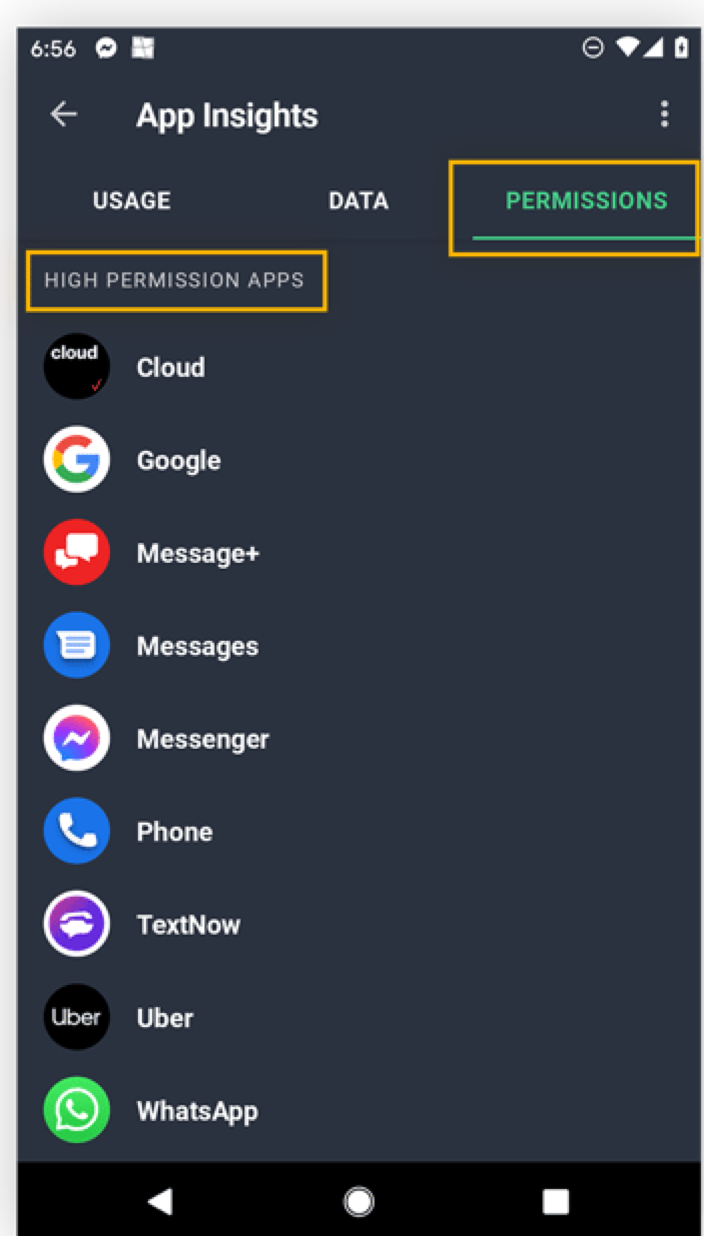
2. Allow the necessary permissions — we need access to your device folders and apps so we can properly protect them.
3. Click the hamburger menu in the top left.



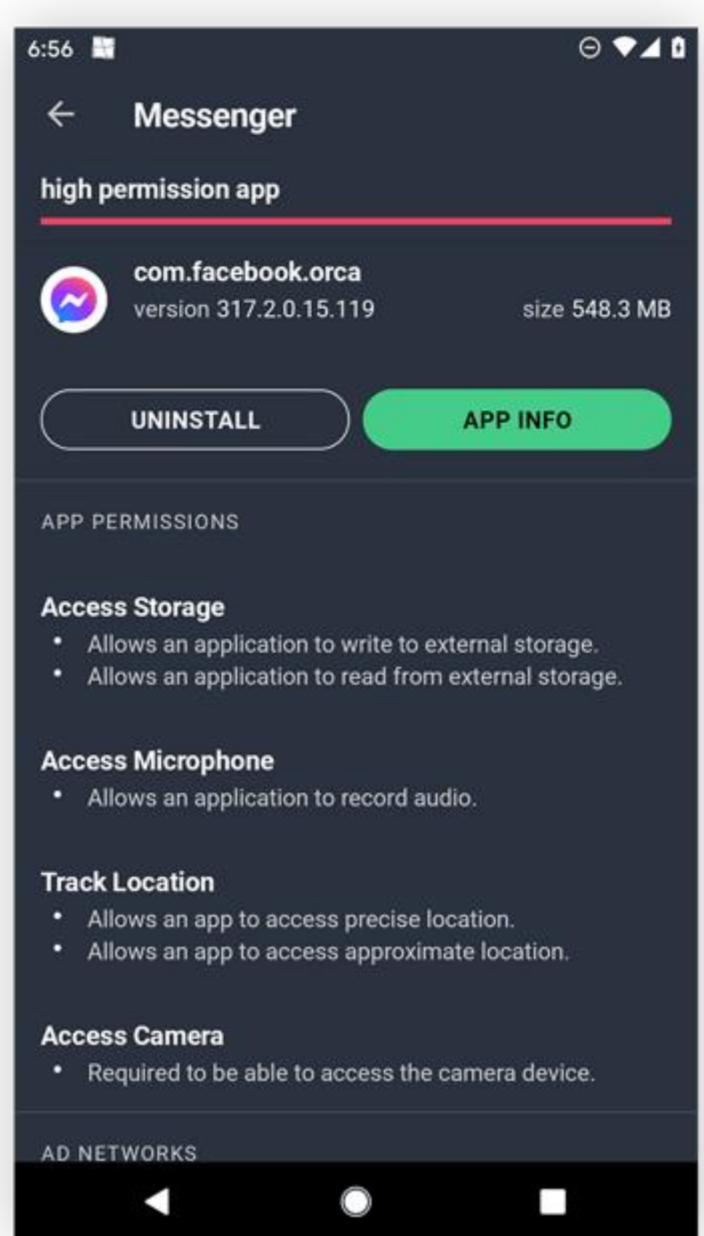
4. Scroll down and tap **App Insights**.



5. Select the **Permissions** category. Here you'll see all your high-permission apps, along with average and low-permission apps. Tap a specific app to get more info on its permissions.



6. Here you can see which permissions might be concerning from a privacy standpoint. You can also easily uninstall the app or get more info.



Additionally, you can review your apps' data usage and screen time for valuable insights into your digital habits. You'll also enjoy ongoing protection against unsafe Wi-Fi networks, password leaks, and malicious software.

Permissions to look out for

Be careful about apps requiring permissions that may compromise your privacy. You should be especially wary of an app that requests a permission that doesn't seem necessary for what the app does.

Android defines nine groups of "dangerous" permissions. Each of these dangerous permission groups contains multiple permissions, and approving one permission within a group also approves the other permissions in that same group. For example, if you allow an app to see who's calling you, you'll also allow it to make phone calls.

Here are the dangerous Android app permissions, explained:

Body sensors app permissions

Body Sensors: Allows access to your health data from heart-rate monitors, fitness trackers, and other external sensors.

- **The good:** Fitness apps need this permission to provide health tips, monitor your heart rate while you exercise, and so on.
- **The bad:** A malicious app could spy on your health data.

Calendar app permissions

Calendar: Allows apps to read, create, edit, or delete your calendar events.

- **The good:** Calendar apps need this permission to create calendar events, and so do social networking apps that let you add events and invitations to your calendar.
- **The bad:** A malicious Android app can spy on your personal routines, meeting times, and events — and even delete them from your calendar.

Camera app permissions

Camera: Allows apps to use your camera to take photos and record videos.

- **The good:** Camera apps need this permission so you can take pictures.
- **The bad:** A malicious app can secretly turn on your camera and record what's going on around you.

Contacts app permissions

Contacts: Allows apps to read, create, or edit your contact list, and access the lists of all accounts (Facebook, Instagram, Twitter, and others) used on your device.

- **The good:** A communication app can use this to help you easily text or call other people on your contact list.
- **The bad:** A malicious app can steal all your contacts and then target your friends and family with spam, phishing scams, etc.

Location app permissions

Location — Allows apps to access your approximate location (using cellular base stations and Wi-Fi hotspots) and exact location (using GPS).

- **The good:** Navigation apps help you get around, camera apps can geo-tag your photos so you know where they were taken, and shopping apps can estimate your address for delivery.

- **The bad:** A malicious app can secretly track your location to build a profile on your daily habits and digital breadcrumbs, or even let dangerous hackers or thieves know when you're not at home.

Microphone app permissions

Microphone: Allows apps to use your microphone to record audio.

- **The good:** A music recognition app like Shazam uses this to listen to any music you want to identify; a communication app uses this to let you send voice messages to your friends.
- **The bad:** A malicious app can secretly record what's going on around you, including private talks with your family, conversations with your doctor, and confidential business meetings.

Phone app permissions

Phone: Allows apps to know your phone number, current cellular network information, and ongoing call status. Apps can also make and end calls, see who's calling you, read and edit your calling logs, add voicemail, use VoIP, and even redirect calls to other numbers.

- **The good:** Communication apps can use this to let you call your friends.
- **The bad:** A malicious app could be spyware that can eavesdrop on your phone habits and make calls without your consent (including paid calls).

SMS app permissions

SMS: Allows apps to read, receive, and send SMS messages, as well as receive WAP push messages and MMS messages.

- **The good:** Communication apps can use this to let you message your friends.
- **The bad:** A malicious app can spy on your messages, use your phone to spam others (including smishing scams), and even subscribe you to unwanted paid services.

Storage app permissions

Storage: Allows apps to read and write to your internal or external storage.

- **The good:** A music app can save downloaded songs to your SD card, or a social networking app can save your friends' photos to your phone.
- **The bad:** A malicious app can secretly read, change, and delete any of your saved documents, music, photos, and other files.

The most dangerous permission types

In addition to the permissions above, Android also has **administrator privileges** and **root privileges** — the most dangerous permission types. You *definitely* don't want any malicious apps accessing these super-permissions on your device.

While Google vets apps before allowing them into their marketplace, sometimes malicious apps sneak into the Play Store. Google works quickly to correct their mistakes and remove them, but sometimes the apps get downloaded hundreds or even thousands of times first.

What are device administrator privileges?

Device administrator privileges (sometimes called *admin rights*) let apps modify your system settings, change your device password, lock your phone, or even permanently wipe all data from your device. Malicious apps can use these privileges against you, but they're also important for some legitimate apps.

For example, security apps with admin privileges are difficult to uninstall, which helps stop thieves and hackers from removing them from your phone. Our free AVG AntiVirus app uses device administrator privileges to let you remotely lock or wipe your device if it's ever lost or stolen.

What are root privileges?

Root privileges (sometimes called *root access*) are the most dangerous app permissions. Any app with root privileges can do whatever it wants — regardless of which permissions you've already blocked or enabled. Malicious apps with superuser privileges can wreak havoc on your phone. Thankfully, **Android blocks root privileges by default**.

But malware makers are always looking for sneaky ways to get root privileges. That's another reason why having a strong Android security app to defend your phone is so important.

All-or-nothing permissions

In earlier versions of Android, accepting potentially dangerous permission groups was an all-or-nothing affair. You either allowed *all* permissions requested by an app — *before* installation — or you declined them all, which meant you couldn't install the app.

Sketchy app developers could abuse this system to sneak in permissions that went beyond the scope of their app — such as calendar apps that requested access not only to your calendar, but also to your microphone.

Thankfully, that mostly changed with the release of Android 6.0 (the so-called Marshmallow update) back in October 2015. Now Android allows you to decide which permissions to accept on a case-by-case basis — *after* the app is installed.

Android app permissions to allow

Allow Android app permissions that apps legitimately need. Google Maps can't give directions without your location, and Zoom can't connect you to a video meeting without accessing your microphone and camera. But make sure to assess Android apps for safety before installing them.

How to tell if an app permission is normal

To tell if an app permission is normal, read the permission carefully and use common sense to determine whether it's a reasonable request. Does a social media app really need access to your location? Perhaps some features won't work without it. But it's up to you to find the right balance between privacy and usability.

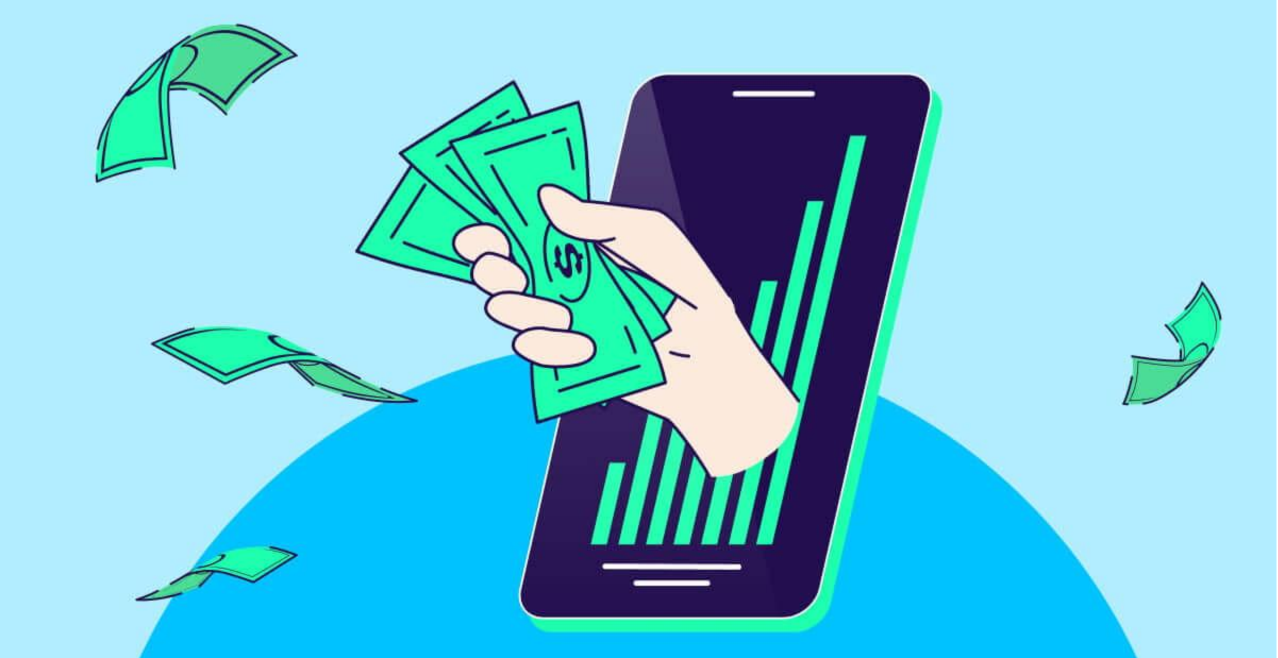
App permissions are designed to protect you. They might seem annoying at first, but you need to approve them only once per app — unless you configure apps to ask each time — and it's well worth it to carefully read and consider these pop-ups before giving access.

Why am I getting two requests for the same permission?

You might sometimes see two back-to-back notifications for the same app permission. This is because the first notification is from the app itself, explaining why it needs the permission. The second notification is from Android and is a generic request for the permission. **Only this second request actually allows or rejects the permission.**

Week-12 Project Development

App monetization guide: How to generate revenue from apps in 2022



- What is app monetization and why is it important?
- App monetization models and strategies
- App store platform fees
- Key metrics to measure the effectiveness of your app monetization model

INTRODUCTION

The world of apps will teach you that nothing in life is truly free.

Mobile apps were downloaded 230 billion times in 2021 – about 95% free to install. Yet the entire industry generated a whopping $400 billion in that year, according to Statista.

So apps definitely make the world go round, but how do they survive and thrive as businesses when they're free?

To make sense of these numbers, we have to dig into how apps make money. In this guide, we're going to break down everything you need to know about app monetization, the various revenue streams, and how you can get started and maximize your income today.

CHAPTER 1

What is app monetization and why is it important?

App monetization is the process of generating revenue from app users.

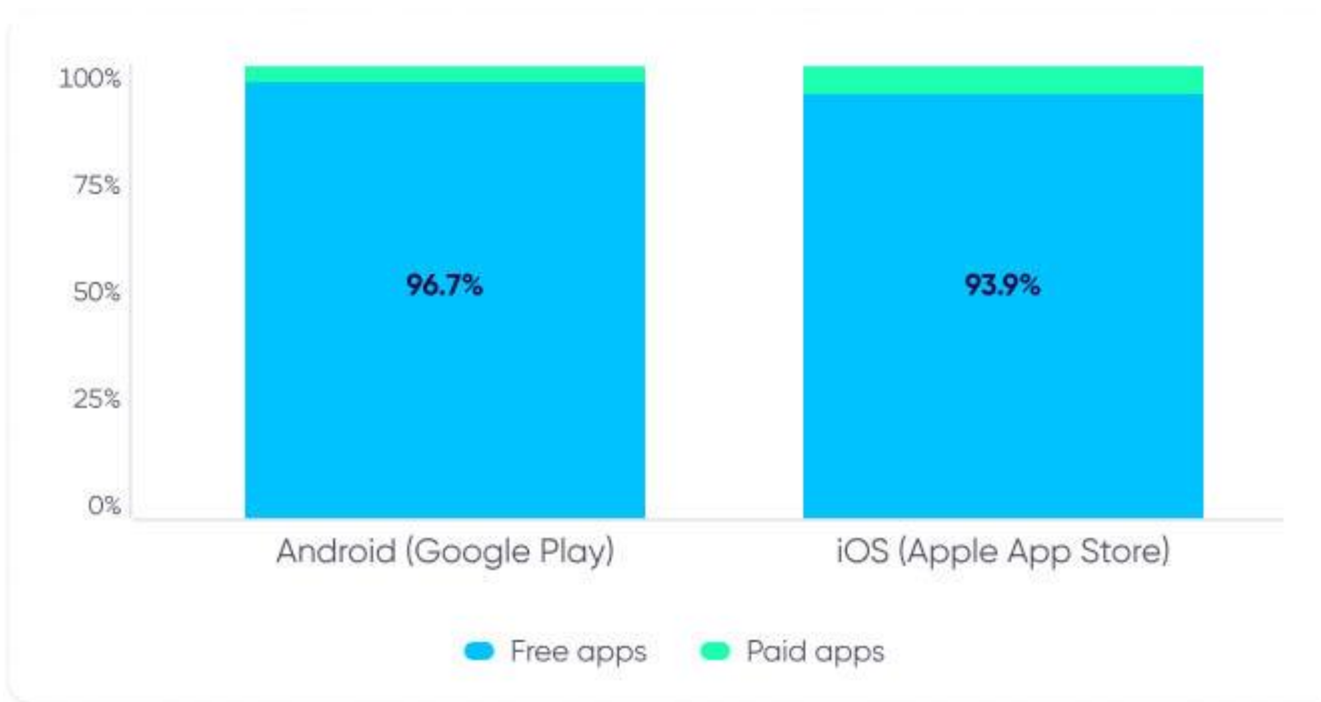
This often includes multiple strategies ranging from showing banner ads on the bottom of your app, to leveraging rewarded video ads, and applying integrated advertising methods that we'll cover more in depth later in this piece.

Understanding app monetization is crucial, because there are numerous strategies that can help app developers and advertisers generate revenue. More importantly, a well-researched app monetization strategy ensures you're creating a positive user experience to retain and grow your user base profitably.

More specifically, owning the metrics around average purchase value, average number of purchases, and retention rates helps measure a user's lifetime value (LTV) and serves as a crucial piece of measuring a marketer's ultimate KPI, return on ad spend (ROAS).

What is the difference between free and paid apps?

As of December 2021, 97% of apps on the Google Play store were free. 94% of iOS apps on the App Store were free. Why is that?



Source: Statista

Making your app free has several benefits that can help you generate more revenue in the long run, because it lowers the risk involved of trying your app out. A free app requires very little commitment, which in turn, makes it easier and cheaper

to acquire new users. While the standard for free apps has risen, there's still a less critical expectation than paid ones.

In contrast, paid apps are heavily reliant on strong brand recognition, unique content, and reputable reviews. Users who commit to paying for an app naturally have higher and more immediate expectations, especially when it comes to user experience. This puts limitations on monetization strategy options, and potentially losing out on long-term revenue.

CHAPTER 2

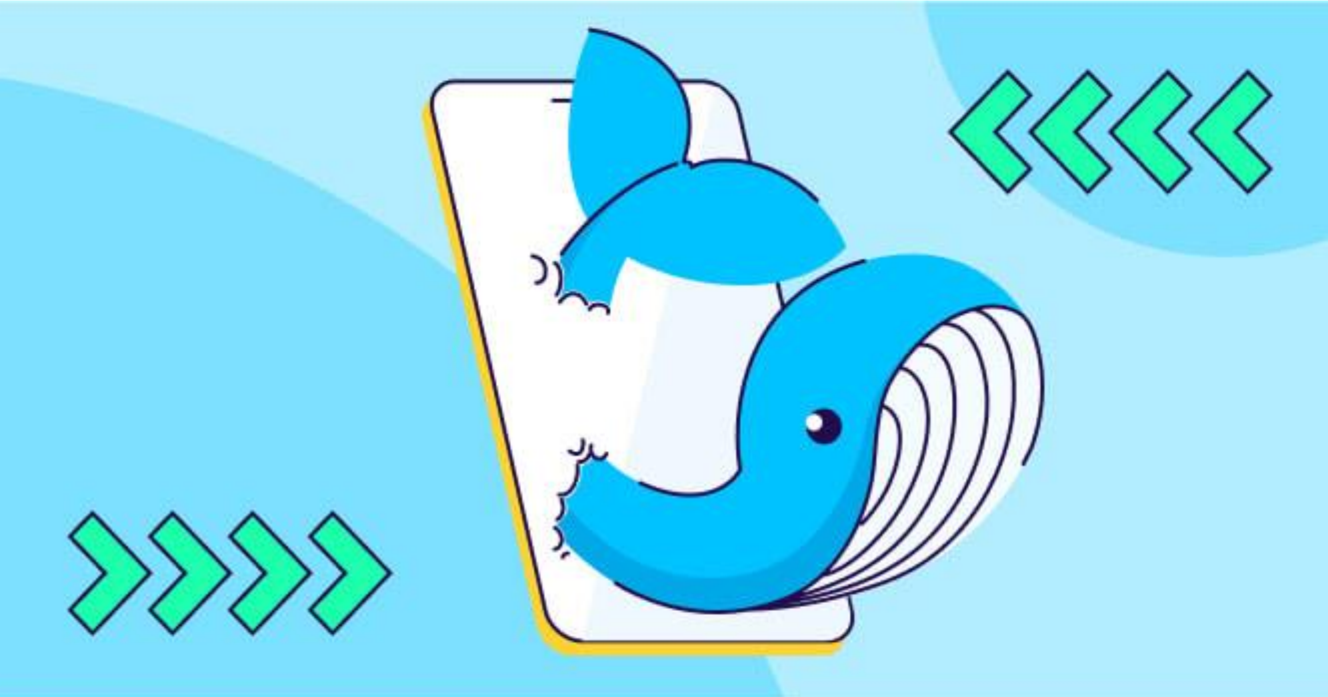
App monetization models and strategies

Apps weren't always free. When app marketplaces first opened their doors, most popular apps cost money. With little to no data around how to generate revenue, asking for an up-front commitment was the best way for developers to generate revenue at the time.

As data became accessible, developers, advertisers, and marketers began to understand that in-app purchases, in-app advertising, and other diversified monetization models proved to be much more lucrative. This is why “free” apps are now the norm.

But since the introduction of heightened privacy measured through Apple’s App Tracking Transparency (ATT) framework with iOS14.5+, it’s become more difficult to measure, target, and optimize marketing activities because of limited data.

Gaming apps were hit particularly hard, because they were highly reliant on a) marketing to drive growth, and b) user-level data to drive positive return on ad spend and identify the ***whales*** (a small share of users who generate most of the revenue).



Despite these changes making mobile marketing more complicated than before, overall mobile app usage has continued to grow across the board. COVID-19 interrupted our normal way of life but accelerated digital transformation in the process.

Phones have become crucial to how we stay entertained and interact with those outside of our homes, which on top of a growing market, exponentially grew mobile usage around the world. As the new normal sank in, the elevated baseline remained intact.

App monetization strategies have also helped fuel the fire. Mobile marketers are utilizing numerous ways to build profitable apps, so let’s break down each app monetization model and identify which one is the best fit for you.

In-app purchases (IAP)

The in-app purchase monetization model focuses on selling a variety of virtual goods, services, and bonuses within the app.

In-app purchase revenue can be divided into two:

1. **In-store revenue:** In-app purchases generated through the app store on which fees apply (see ahead for more). The lion's share is driven by gaming and subscription apps. In 2021 alone, in-store revenue generated $133 billion of consumer spend – a 20% increase compared to 2020.

2. **Out-of-store revenue:** Any revenue that is generated directly in the app (eCommerce, travel, food delivery, transportation, etc.). It therefore generates even more revenue than in-store.

Benefits of in-app purchases

We are living through a microtransaction economy, where apps are successfully generating profit by selling in-app experiences. These can include everything from selling sticker packs on messaging apps to plugins that unlock new design features in a video editing app. Dating apps can sell features that allow you to reach more people or boost your social profile.

IAP works effectively for gaming apps in particular when it enhances an already-fun gaming experience. It's a low cost way to generate exclusive items and in-game benefits that ultimately drive a loyal user base.

What kind of apps should use in-app purchases?

In-app purchases can be effectively utilized across almost every type of app: from mobile games like Clash of Clans, Candy Crush, Pokemon Go, and Harry Potter: Wizards Unite, to non-gaming apps in eCommerce, transportation, social, and productivity.

Best practices

Offer a valuable upgrade

The most effective way to drive paying customers is to offer an improved experience on top of an app they already enjoy using. In-app purchases must be robust and engaging enough to generate long-term revenue. Whether it's offering an ad-free experience or unlocking premium content, test out what your users find to be the biggest value-add.

Compliment this by continuously optimize pricing depending on different geographies and user segments. Experiment with promotions to see how different segments react to unique price points.

Keep a pulse on performance with predictive modeling

Utilize predictive modeling to make quick campaign optimization decisions. Machine learning algorithms can predict and prevent users from churning by identifying key patterns that your most profitable users make.

For example, it can predict that your most profitable users complete the first five levels of a game within the first hour of play. You may utilize that data to promote your in-app purchases and tailor your messaging accordingly.

Create multiple currencies in games

Create a soft and hard currency. A soft currency can be rewarded through certain actions within a game, which provides a sense of growth and advancement. Hard currency can be earned in very limited amounts through difficult in-game actions, but should be bought with real cash. Whether it be granting extra draws in a lottery, exclusive skins, or leveling up your character, this helps users speed up their progression in the game.

Avoid being labeled as a pay-to-win game

Splitting in game purchases can help maintain the value of in-game currency with real-world money that generates the actual revenue. It's integral to maintain the value of in-game currency that everyone can earn to keep the game challenging and accessible to all. An imbalance could cause higher churn if other players feel a game is pay-to-win.

Subscription and freemium models

A subscription model relies on an app being free to download and charges a recurring fee to use over time. Oftentimes, subscription-based apps take a ‘freemium’ approach, which offers limited features for free and asks users to pay to unlock the rest of the app.

Benefits of subscriptions

When executed well, subscriptions can be an extremely lucrative advertising model. Monthly subscriptions can help create predictable cash flow, especially if developers have a good sense of churn and retention rates or in other words, how many users are joining or leaving the app. Similarly to users who pay for apps up front, subscribers are actively invested in using the apps that they pay for.

Additionally, Apple App Store lowers commissions fees from 30% to 15% for subscriptions after a year. The Google Play store will take only 15% on all subscriptions from day one.

What kind of apps should use the subscription model?

Apps that provide unique content on a regular basis can be successful with the subscription model. This can include video streaming apps, dating apps, fitness apps, news apps, and productivity apps.

Best practices

Focus on content first

Provide fresh, new, and exclusive content on a regular basis, especially if you’re a video streaming service. Competition is fierce, so it’s integral that you provide a unique experience that cannot be found elsewhere.

Offer discounts for long term subscriptions

Offer users a significant discount for 6 to 12 month subscriptions that are paid in a lump-sum. It’s a win for your most loyal customers and also provides you guaranteed up front cash flow.

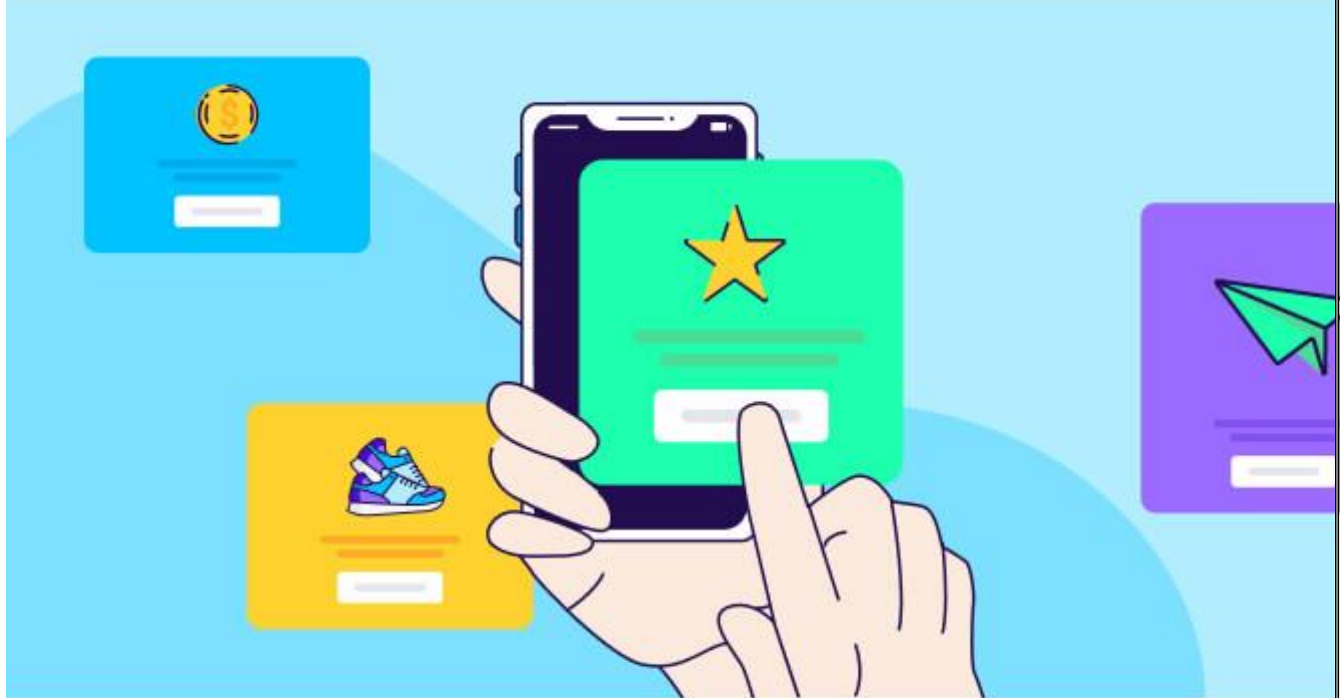
Provide an exceptional onboarding experience

A great onboarding experience can help your users extract the most out of the app as early as possible. It's essential that you clarify the most valuable aspects of your app and that they're using all your best available features.

Gift free trials

A no-frills free trial that allows users to identify the value of a product before making a purchase decision. At the end of the day, a great product sells itself. While there's no guarantee a user will convert to becoming a paying customer, you'll make the decision so much easier by doing so. Even if your conversion rates are low, you'll gather data to identify key opportunities to improve your app.

In-app advertising (IAA)



In-app advertising, or IAA for short, is a monetization strategy that leverages an app's real estate, mainly in gaming, to show ads to their users. In short, advertising space is sold to ad buyers to display ads.

Because the vast majority of users do not complete in-app purchases, having ads in your app can drive revenue from this large segment of users (while not showing ads to users who do monetize through in-app purchases). It is therefore a highly popular revenue stream, especially in games.

Even though it's tempting to show as many ads as possible in your app, developers need to strike a delicate balance between serving the right number of relevant ads while not harming user experience with overexposure.

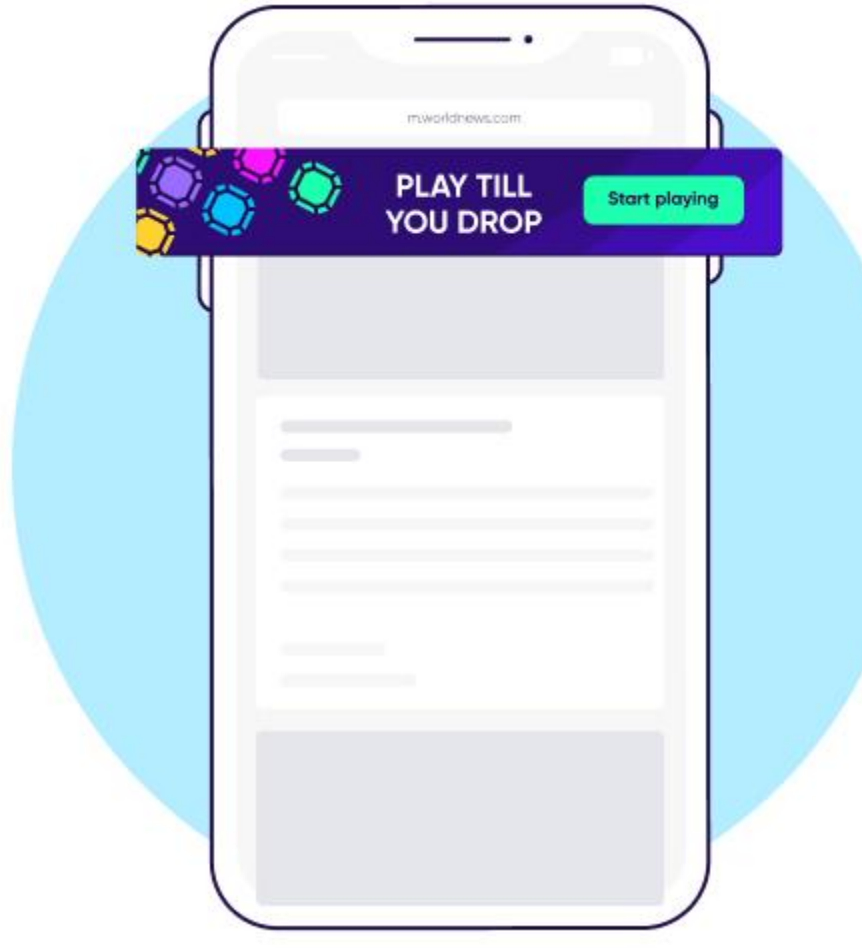
What kind of apps should be using in-app advertising?

In-app advertising is ubiquitous in the app world and can be utilized effectively with almost any free app. However, they are particularly popular among gaming apps to monetize the non-paying audience.

Apps, especially those with IAP serving as the main revenue stream (e.g. eCommerce, travel, and hardcore games), should carefully consider if and when to use in-app advertising to avoid any friction in the path to purchase.

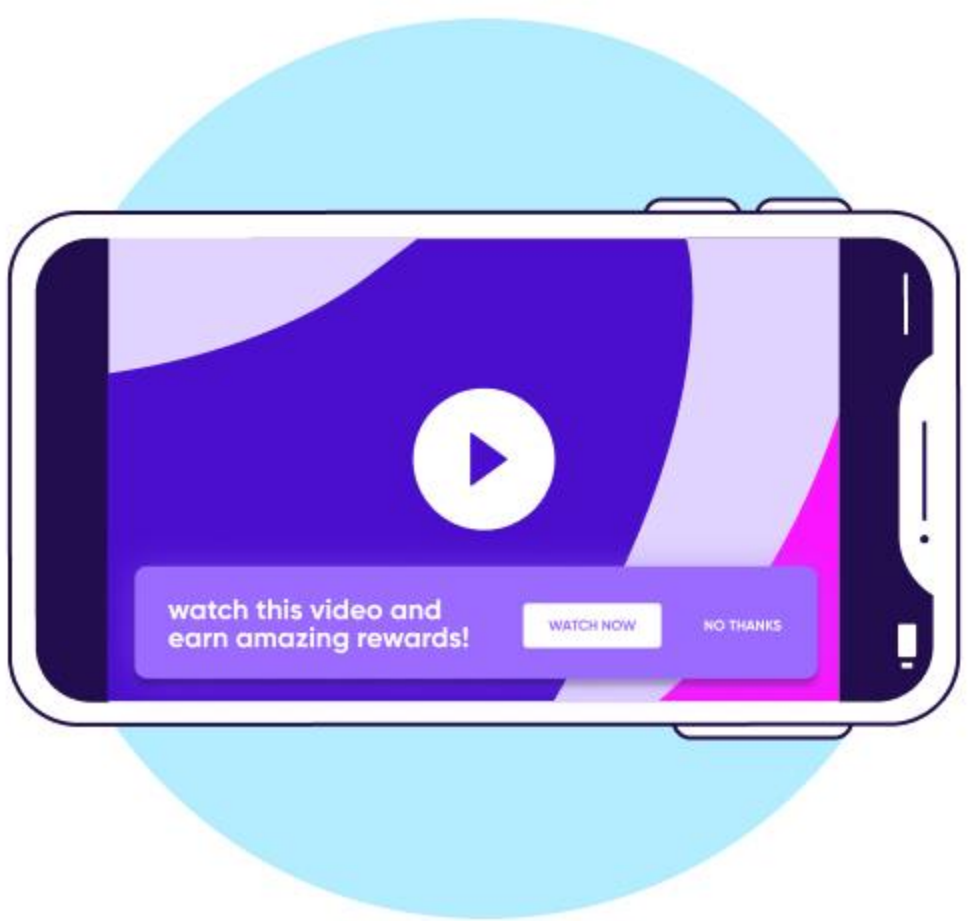
In-app advertising formats

Banner Ads



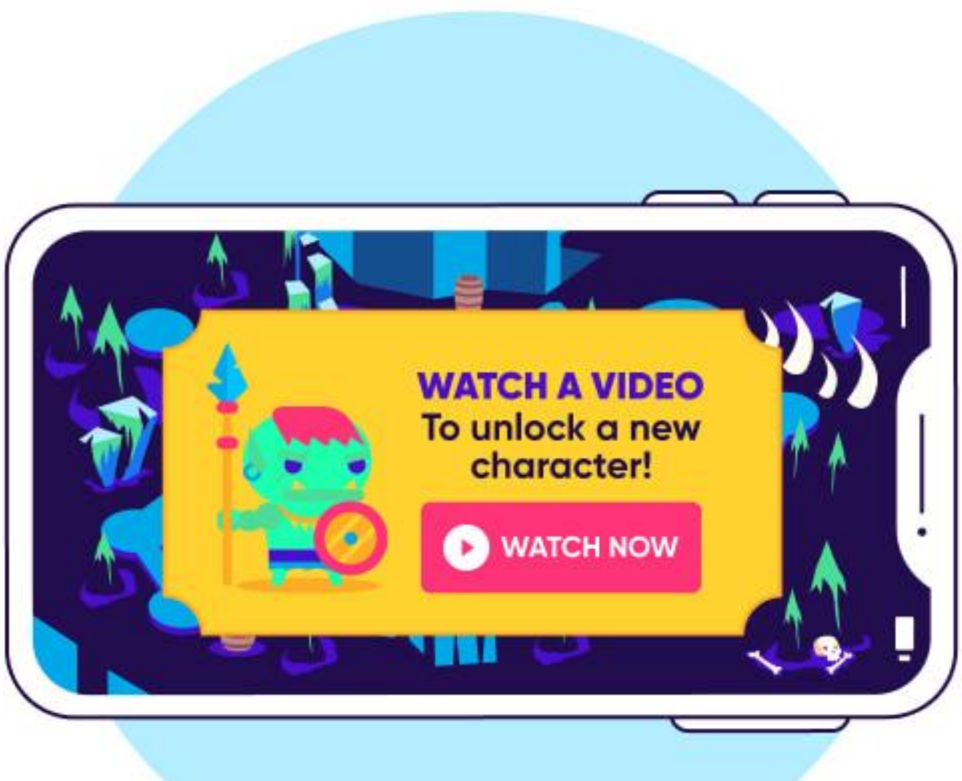
Banner ads are made up of graphics and text components that appear usually at the top or bottom of the screen with a clear call-to-action button.

Video Ads



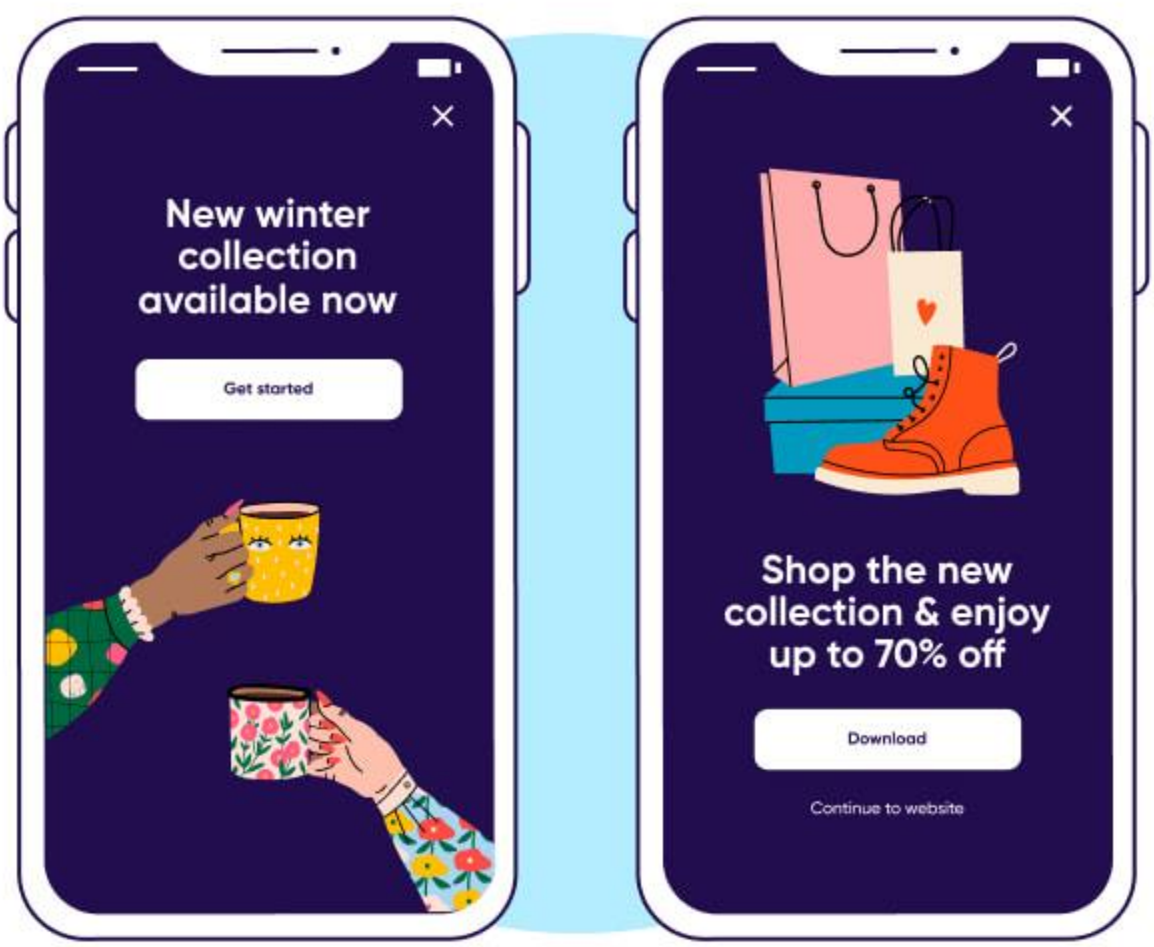
Video ads are video clips that appear before another video. An example of this is a pre-roll video that appears before you watch your video. They often have high engagement and click through rates.

Rewarded video ads



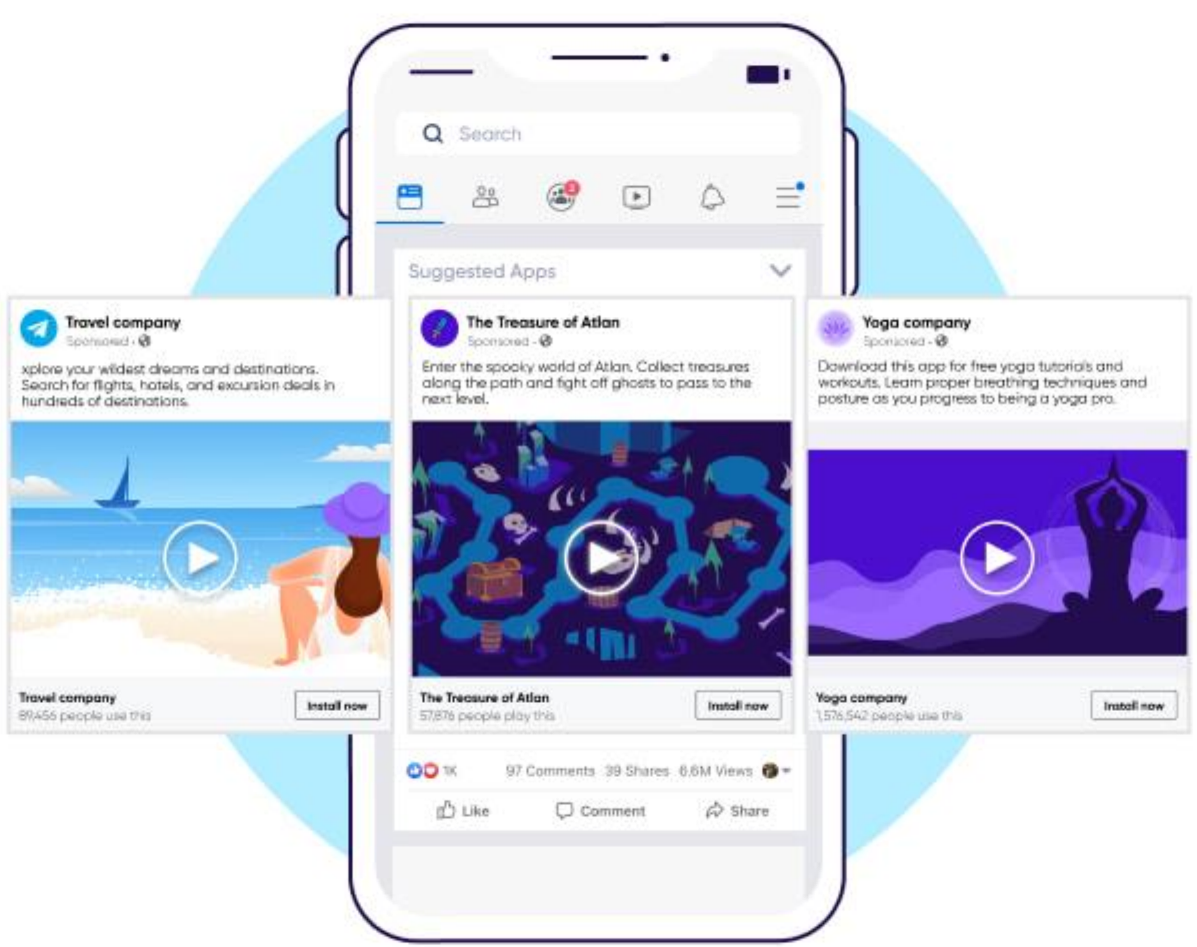
Rewarded video ads offer users an in-app benefit for watching a video in its entirety, which is a lucrative ad format for gaming apps. Rewards can be anything from unlocking new characters to in-app currency or extra lives – as such they are used almost entirely in gaming apps.

Interstitial ads



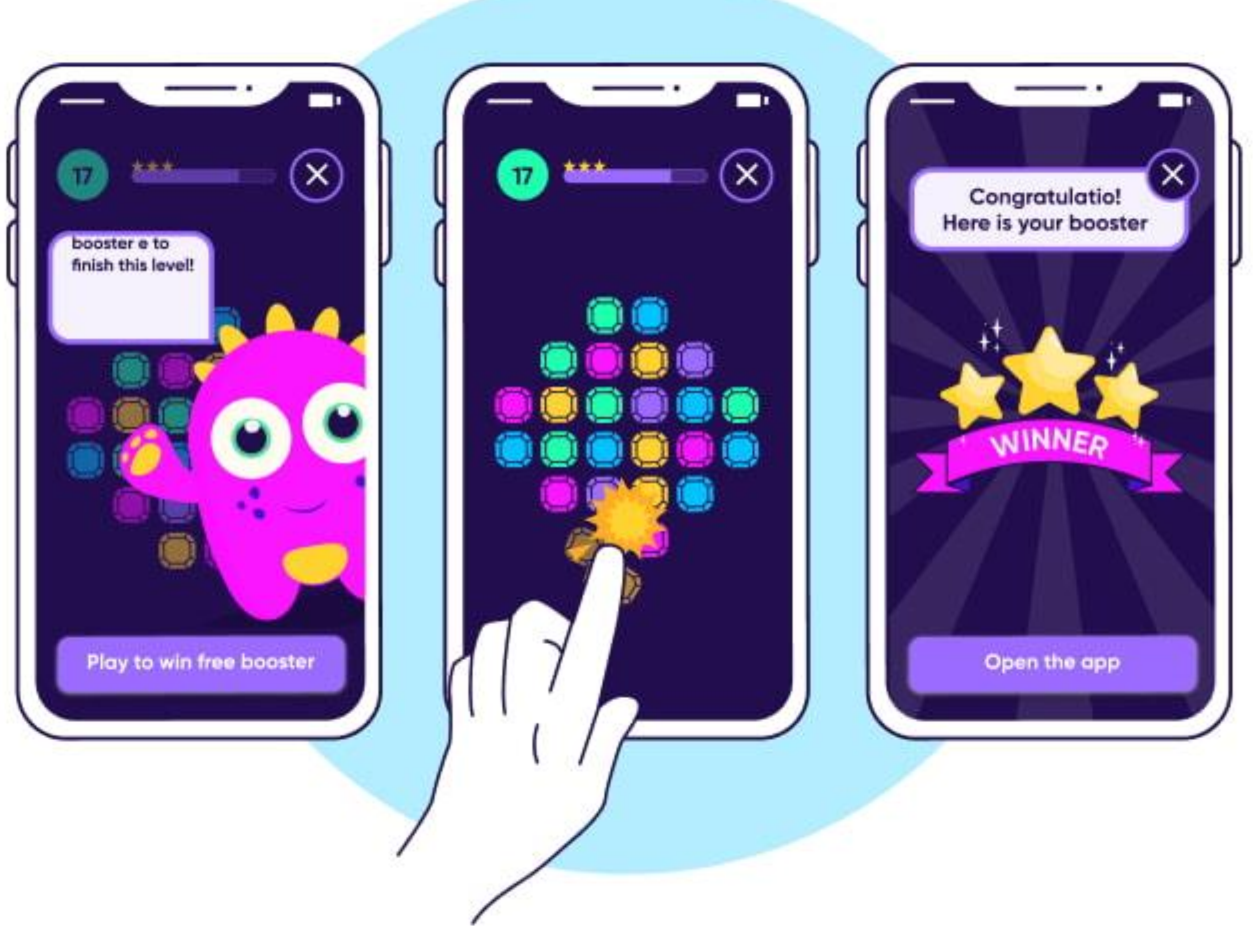
Interstitial ads are full screen ads that typically appear during a transitional phase of the app, such as a level completion. They are often rich media and can be in the form of videos, images, and/or text.

Native Ads



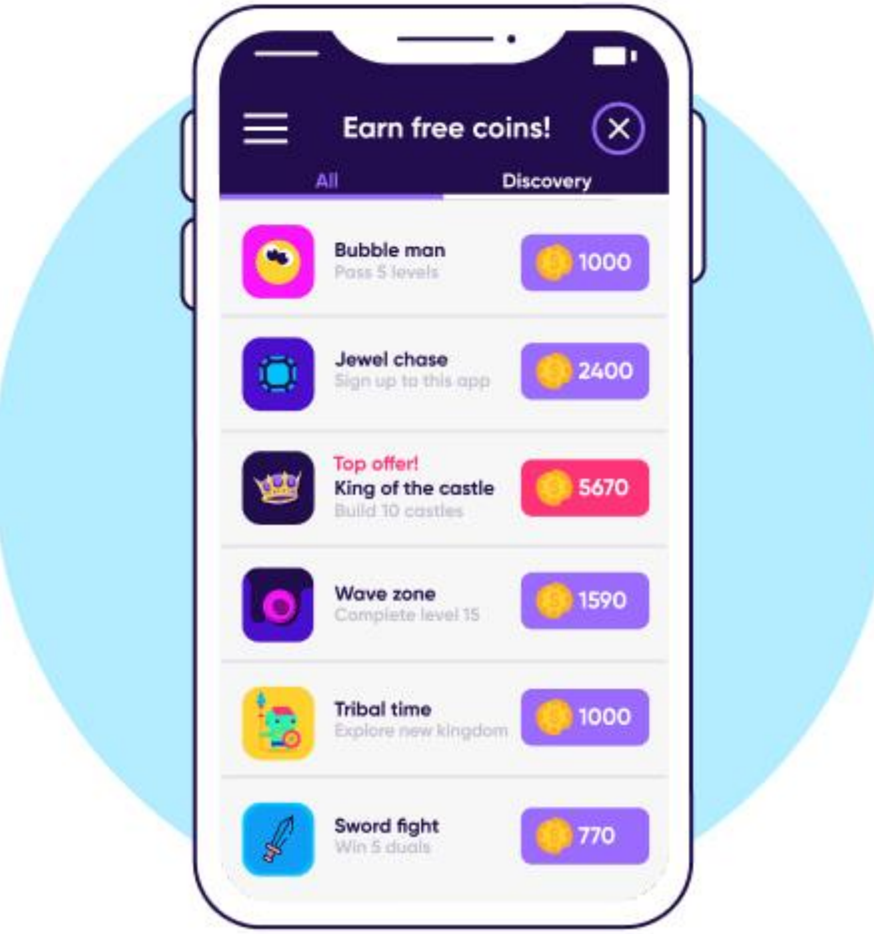
Native ads appear to look like organic content, such as a news feed or content recommendation engines that are marked as sponsored content. They're designed to fit naturally within the app without hindering user experience.

Playable Ads



Playable ads are playable mini games that briefly show a key mechanic of another game. They're engaging and drive one of the highest CPM rates for gaming apps.

Offerwall Ads



Offewalls provide users rewards in exchange for specific actions such as filling out surveys, playing a game, downloading an app, or reaching a certain level in another game.

Benefits of in-app advertising

Ease of implementation

In a world of free apps, in-app purchases and in-app advertising are the highest grossing revenue channels. One of the biggest reasons for this is the accessibility of it. Not all apps can sell in-app upgrades or purchases, while most apps can easily implement video ads, banner ads, and native ads.

Integrate flexibly without ruining the user experience

There are diverse IAA formats that can be implemented flexibly and seamlessly within apps. Integrating the ad into the natural flow of your app will make it less disruptive to the user and better for the app as a whole. Especially in casual gaming apps, you can utilize rewarded video ads to offer in-game benefits in exchange for watching ads.

Best practices

Test everything – especially frequency

Like any other advertising method, ensure you're testing which formats and pricing models are the most profitable for your app. Always measure your users' behavior and don't be afraid to test multiple ad platforms to identify which generates the highest CPM.

Improve viewability

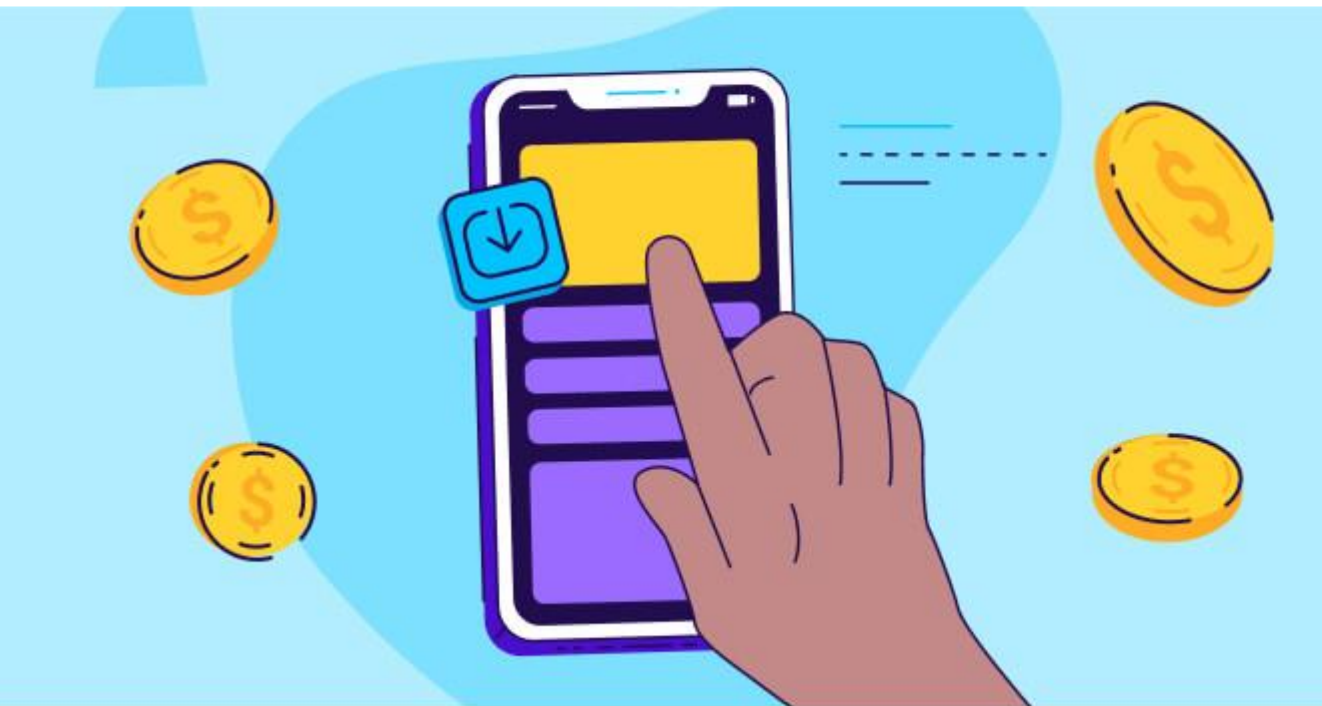
Not checking for viewability is a surefire way to lose out on advertising revenue. Mobile display ad impressions are only viewable when 50% or more of the pixels are within the user's view. They also need to be shown for at least one continuous second after rendering. Regularly test these requirements so you get paid accordingly.

Invest in creative

Despite the data focus in recent years, creative was and remains one of the most important factors in marketing success. Investing in top designers, measuring creative success, and testing every component with multiple variations are vital.

Recommended reading**: The complete guide to in-app advertising**

Paid Download or Pay Per Download (PPD)



One of the oldest monetization strategies is to charge a one-time fee to download your app. Paid downloads effectively generate revenue, but as mentioned above, may leave long-term revenue opportunities on the table.

Benefits of the paid download model

Paid apps heavily depend on brand recognition, unique content, and trusted reviews, especially if it doesn't come with a free trial. However, once a paid app presents undeniable and differentiated benefits, you can generate predictable and consistent revenue.

Because you're only measuring app downloads over time, revenue is a lot more easily measured but at the cost of not being able to upsell your users with other monetization strategies. Additionally, not having interrupting banner ads or interstitial ads ensures a much higher quality and uncluttered user experience. Lastly, paid app users may have higher expectations for your app, but they're more likely to be dedicated to extracting the most value out of your app, which may help with user retention.

What kind of apps should be pay per download?

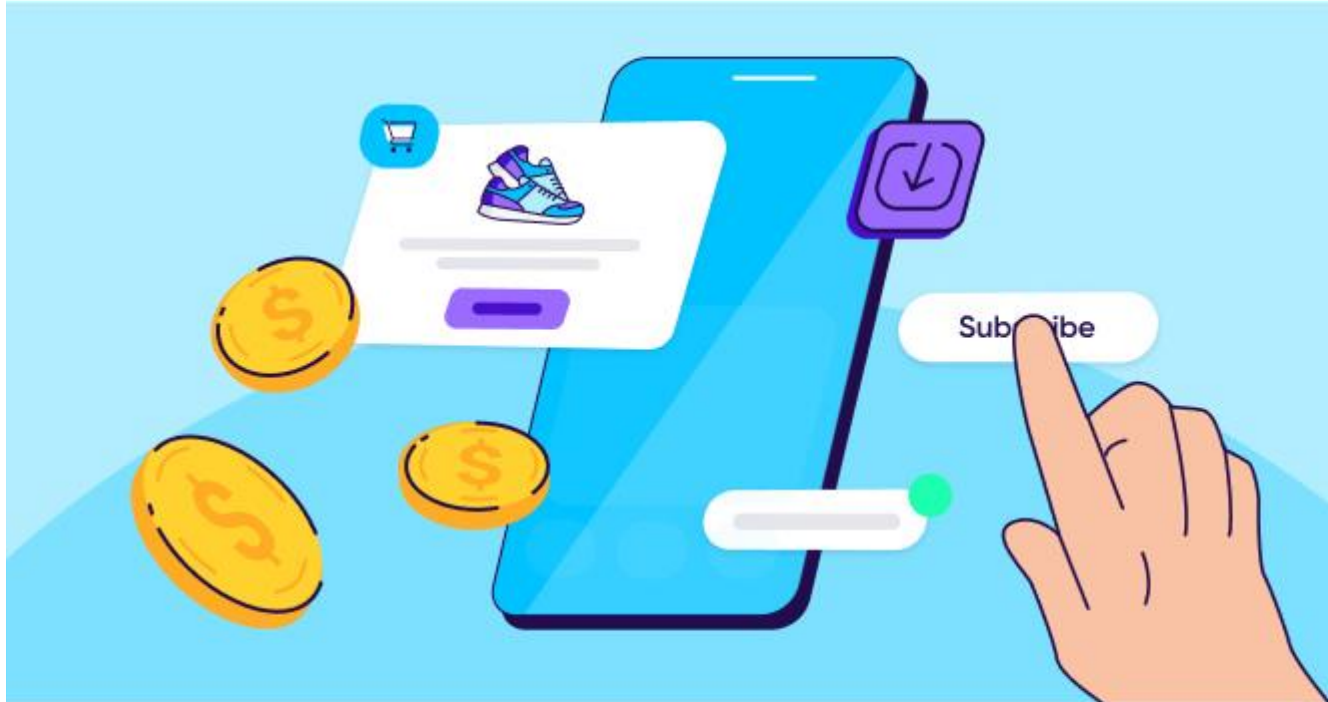
Apps with a unique and clearly differentiated feature or a household brand name with a unique offering should be paid.

Best practices

Before you adopt this method, ensure that you're positioned as the differentiated market leader in what you do. The market will pay a fair value if you're serving a unique need.

Also, verify that there are no free alternatives to your app and that you provide value to your users immediately. Lastly, dedicate your resources to improve [App Store Optimization] to improve your app's ranking, trust, and visibility.

Hybrid monetization strategy



There's no reason to limit your app to one monetization strategy. Many successful apps utilize a hybrid monetization strategy, which is a combination of two or more strategies to maximize revenue. Typically, apps using a hybrid model utilize a combination of in-app ads and in-app purchases.

Benefits of the hybrid model

The hybrid monetization model is extremely flexible and allows developers to generate diverse revenue streams from both users who are likely to make purchases and the less inclined. It entices more profitable users to reap the benefits of in-app purchases while providing an opportunity for all other [users to improve their app experience] by watching and engaging with ads.

What kind of apps should use the hybrid model?

While all apps can benefit from the hybrid monetization model, gaming apps have found great success utilizing a combination of in-app purchases with in-app advertising. For example, users may see a video ad upon completion of a new level while having the option to purchase a new life as well.

Best practices

Apps using hybrid monetization strategies need to be extra wary about showing too many ads. Prioritizing user experience is especially key, to ensure your app is worth spending money on. It's integral to moderate the number of pop-ups, ad displays, and even the pricing of your in-app purchases as well.

Taking this a step further, segment your users appropriately. Encourage your "whales" to make more valuable purchases. You may also consider showing more in-app ads or rewarded video ads to players less inclined to spend.

Get the latest marketing news and expert insights delivered to your inbox

Subscribe

CHAPTER 3

App store platform fees

Getting your app listed on the app store only takes a few clicks. The listing process is easy, but getting your app shown is challenging. With so much competition growing by the hour, it's important to ensure your app is listed on multiple marketplaces and optimized for. On the flipside, you'll have to pay a fee for listing your apps on each respective marketplace.

Here's a rundown of all the fees.

Apple App Store: 30% commission for every paid app download and in-app purchases of digital goods and services. Developers generating less than $1 million per year in App Store sales may qualify for Apple's App Store Small Business Program that charges a 15% commission instead. No fees are charged for physical products. Subscription commissions are lowered to 15% after one year.

Google Play Store: 15% fee for every paid app download and in-app purchases of digital goods and services up to $1 million in sales. The rate is increased to 30% once the threshold is met.

Galaxy Store: 30% commission (negotiable) on purchases made on the app store.

Amazon App Store: 30% commision for every paid app download and in-app purchases of digital goods and services. Video apps have a 20% commision. Developers that generate less than $1

million per year in app store revenue can qualify for a 20% commission rate and 10% promotional credits for Amazon Web Services.

Microsoft Store: 15% commision for every paid app download and in-app purchases of digital goods and services. 12% commission on PC games. 30% commission on all apps, games, and in-app purchases on Xbox consoles. Non-gaming apps may also utilize their own payment systems to avoid commissions.

CHAPTER 4

Key metrics to measure the effectiveness of your app monetization model

Now that you understand the difference between all the app monetization models, let's talk about how you can measure the success of the one you've chosen. Here are the most important metrics to keep an eye on.

Retention rate: The percentage of users who use your app over a certain period of time. While there are nuances in independent user behavior, retention rate is a great high-level metric to identify whether or not your app provides a valuable user experience which translates into loyalty. When implementing in-app advertising strategies, it can also help identify whether or not it's affecting user behavior.

Retention is key as it is the basis of monetization: the more loyal your users, the more ads they will engage with and / or the higher the likelihood that they will make an in-app purchase.

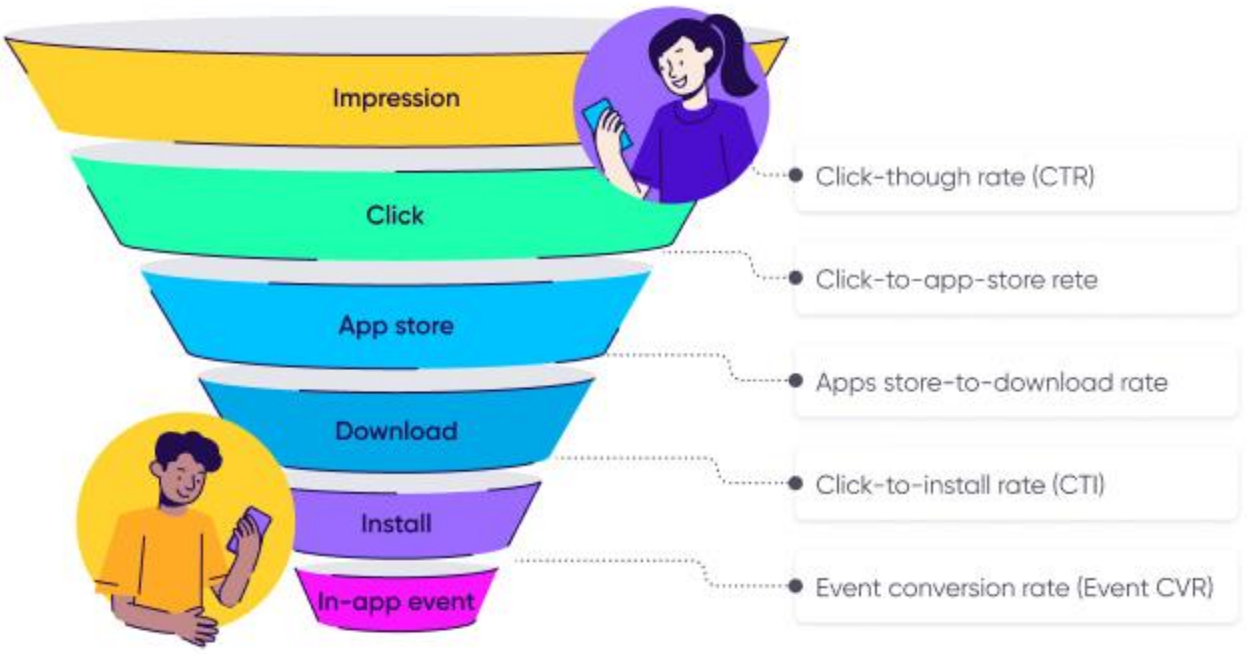
App Stickiness: The extent to which users engage with your app on an ongoing and regular basis. The higher user engagement is, the stickier the app is. This is measured by dividing your Daily Active Users (DAU) by your Monthly Active Users (MAU).

$$\frac{\#\text{ of DAU}}{\#\text{ of MAU}} = \text{DAU MAU Ratio}$$

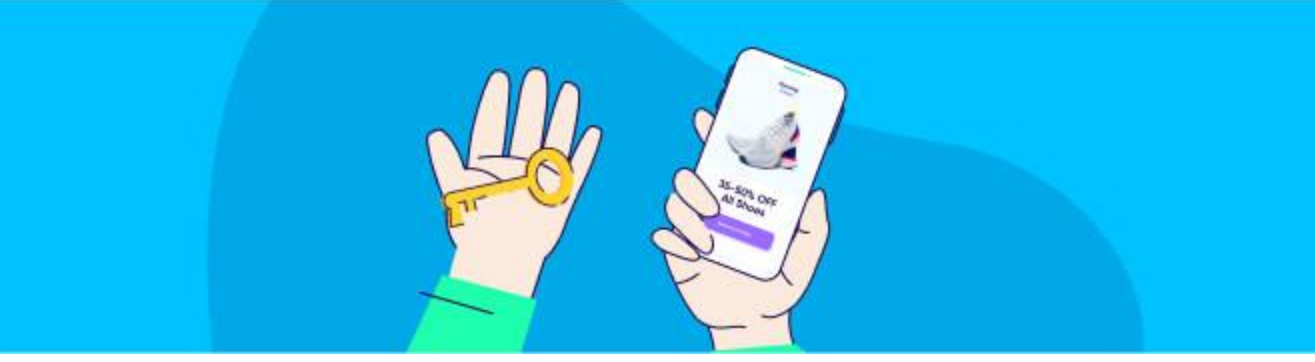
ARPU (average revenue per user): Helps measure how much you're earning, on average, from each user. It's calculated by dividing total revenue of a business by the average number of users within the same time period.

Lifetime Value (LTV): Determines how valuable a user is over the span of time they're using an app. The information helps marketers determine how much money they can invest in advertising to acquire customers and remain profitable at the same time.

Funnel conversion rates: Help determine how the quality of users going down the marketing funnel — from the moment they see an ad to completing a meaningful, revenue-generating in-app event after installing an app. This data allows apps to optimize monetization strategies by identifying where users are falling off in the advertising/purchasing process.



Once you understand these key metrics, you can test multiple models that you think would work best for your app more effectively. Ultimately, you need to measure how each ad experience affects your bottom line.



KEY TAKEAWAYS

Key takeaways

- Although 95% of apps are free to install, the entire industry generated a whopping $400 billion in that year from a variety of monetization sources
- There are 5 app monetization models: in-app advertising, in-app purchases, paid apps, subscriptions, and hybrid
- Creating a user-friendly app monetization strategy will ensure you retain and grow your users profitably
- In-app advertising and in-app purchases are by far the most popular mobile app monetization models
- To best understand what advertising model to use, start by differentiating whether you want your app to be paid or free. Test multiple models while calculating KPIs like retention rates and ARPU to effectively measure each model's performance.

What is monetization?

The word monetization sounds dirty, doesn't it? It sounds like you're planning to exploit your product's users in some way to dupe them into paying for something they don't want in order to generate cold, hard, evil cash.

The same thoughts appear when you ask someone to think of salespeople, selling – or the opportunity to pursue a career in sales. The thoughts that spring to mind are of somehow being tricked into buying something you don't want.

Sales is dirty.

Money is dirty.

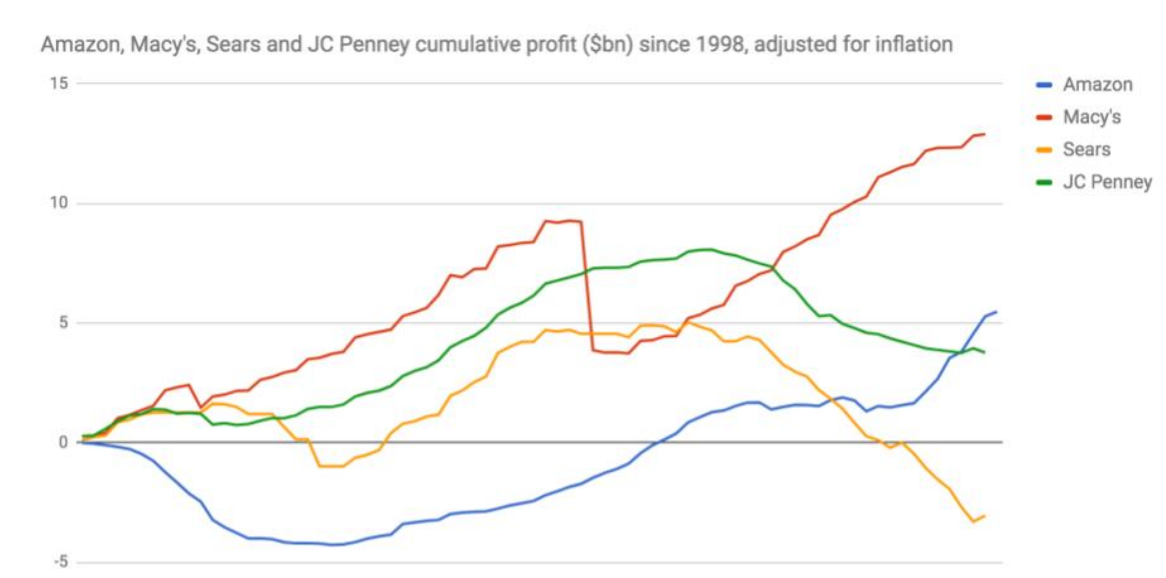
'Monetization' is very dirty.

Sure, there are plenty of examples of where companies have gone too far in monetizing their product, invading users privacy and destroying the user experience, but that's monetization and sales done badly.

Monetization is the process of deriving revenue from the value you offer to your users.

Your product – if it's a product worth using – is delivering meaningful value to its users in some way. It's only natural therefore that you can expect to receive something in return – including revenue. This revenue may or may not come from your users but it's fair to suggest that in exchange for the value you offer and deliver to your customers, you can expect to be able to derive some form of remuneration from somewhere in return.

The difference between revenue and profit



For over a decade, Amazon didn't make a profit. It's primary focus is – and arguably always will be – to provide extreme value for its customers. How? By offering the cheapest possible price on the goods it sells by cutting costs and ploughing money back into the business to drive efficiencies where possible.

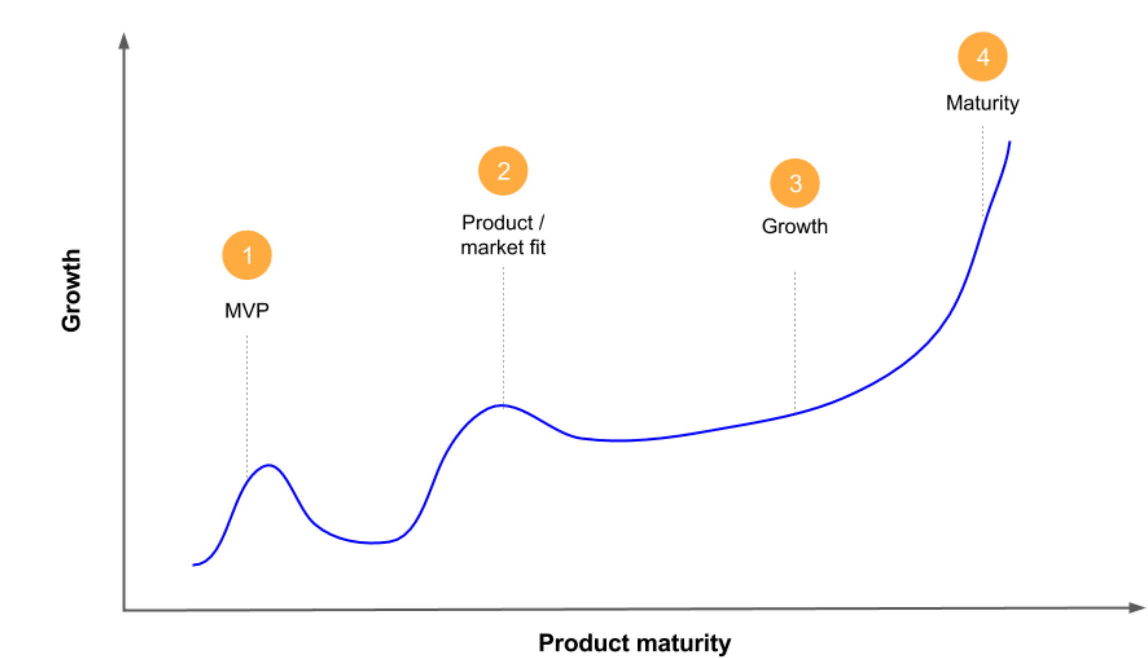
Amazon could sit pretty on top of a huge cash pile but instead chooses to enhance its value proposition by offering cheaper prices through razor thin margins and loss leaders.

It's large enough to experiment with new loss-making product ideas (Kindle Fire Phone, anyone?), put competitors out of business and branch out into web services because its commitment to delivering value through its core business and not chasing profits is what continues to underpin its success.

Monetization, startups and the product lifecycle

Some products will have monetization mechanisms baked into their model from day 1. Others will need to achieve scale before they have a solid platform which can be monetized later.

Indeed, it's arguable that too much focus on monetization in the early stages of your startup could *inhibit* growth. Would you have an Instagram account if you had to pay £5 a month for it? Would you ever have tried Spotify if the only option was to pay $9.99 a month?



Your monetization strategy is linked to the stage of your product's lifecycle / business.
Some startups won't focus on monetization at all. Instead, they may be acquired purely on the basis of the technology they've built or the talent they've recruited.
If you're a startup and you know upfront that you're not seeking to generate meaningful revenues until you have reached X number of users, make this decision clear and set your investment / growth goals based upon this.
Even if you don't have revenue generation baked into your product from day 1, it's still worth considering what your revenue streams might be in the future. If there is a no potential path to monetization in the future this can be problem for your product and for your business.

The five rules of data monetization

How do we value our data? That's the question so many in business are asking at a time when the volume of data available to companies is growing exponentially. What companies are realizing is there is no finite answer. Unlike tangible assets, the value of data can grow the more ways it is utilized and the more insights that are drawn from it. No wonder then that data monetization – literally defined as the process of increasing the economic value of data – is becoming such an important and growing area for all businesses. Through data monetization your business can start treating data as an asset and gain the benefits from maximizing its value.

The potential for data to deliver value for many parts of the business is enormous.

How then can your business succeed at data monetization? From our* experience these are the five rules you need to follow.

1. **Understand the role and value of data in your business**
   Good data management is about making sure you have the proper data to support your business and improve performance. Smart data utilization also helps in managing risk and

provides assurance that the business is compliant with laws and regulations. But it can only serve this purpose effectively if you know where your data resides, how relevant it is and how valuable it could be. Often companies fail to accurately value their data because it is not strictly accounted for as an asset even though it has real worth in external markets.

2. **Get your data house in order**
   Too many companies lack metadata – i.e. data about data – such as the data quality, where it's stored and what it means. In fact, many companies are more likely to have a more detailed inventory of their office furniture than their own data. Before thinking about monetizing data, companies need to discover what kind of data they hold about their partners, customers, products, assets or transactions and what publicly available data can be called on to increase the value of their proprietary data. They must also work out whether that data is of value internally to cut costs, streamline operations or improve sales processes, or as an external revenue stream such as customer intelligence as a service, or both.

3. **Embed data monetization into business strategy and get the right structures in place**
   Too often corporate strategy is not supported by related data management initiatives and vice versa. Executives should evaluate their key business goals and strategic initiatives through the lens of how data can support them. Consider the energy sector where a company that applies service-oriented asset management can gain a lot of insight into individual materials and parts or the equipment. This information can be used to barter better terms and conditions with the material and parts vendor.

   Once you understand the quality of data and have tied it to business strategy then you can put the right structures in place to monetize it. Often this involves assembling a multi-disciplinary, cross-functional team – including information product leads, data management experts and executives from sales, marketing and operations – to create a platform for business innovation powered by the data. Together they can determine ownership structures for different data sets while making sure that sense of ownership doesn't result in data bottlenecks or siloes.

4. **Be open to new opportunities**
   The potential for data to deliver value for many parts of the business is enormous. Sometimes, though, it's hard for companies to imagine quite what the opportunities could be because they are so used to pursuing growth through established strategies and revenue streams. That's why all companies should be open to learning from other businesses and partnering in ways that make sense from a data point of view.

   Take the example of one telecom company in the Czech Republic. It was able to package the geographical data it had about mobile phone users to help a national park better understand where visitors were coming from, how long they stayed in the park, and, by analyzing traffic flow data, was able to model which special events prompted increased visit footfall. Armed with this information, the park optimized its services and marketing and saw a 10 percent visitor increase.

5. **Communicate data's value internally and externally to foster growth**
   Monetizing data is still a relatively new experience for many organizations, and even when successful initiatives are in place they aren't always known to the business as a whole. As data becomes more and more important, companies will need both to communicate and educate internal and external stakeholders so they fully grasp the value data can deliver.

   One approach for internal awareness building is to brainstorm with those professionals in charge of data or with business function leads to show them how data can improve their business processes. What types of data could really help change their work, offer a competitive advantage or open up new revenue streams? Then there are the success stories of companies monetizing their data – these tend to win over even the most skeptical of executives.

   Gathering direct customer input is also important. What benefit do they see in the data you can give them access to? However, external data monetization initiatives do need to pay heed to issues concerning legal, privacy, intellectual property, logistical and technological delivery.

   Ultimately, the sooner companies educate themselves and settle on a strategy to monetize their data the better. With the amount of data increasing at an exponential rate, those companies that understand its potential and can monetize it shold enjoy a real advantage over competitors that still struggle to understand its true business potential.

What are in-app purchases (IAPs)?

An in-app purchase (IAP) is the purchase of virtual goods, services, or additional content within a mobile app. IAPs are made by users while they are using the app and are typically facilitated through various payment methods, such as credit cards, digital wallets, or app store accounts.

IAPs offer app developers and publishers a way to generate revenue beyond traditional methods like selling the app itself or displaying ads. In-app purchases allow developers to design their apps in such a way that users have the option to buy various in-app items, features, or subscriptions.

Common examples of IAPs include:

- Virtual goods: Virtual goods are items within gaming apps that enhance the user experience, such as character outfits, power-ups, virtual currency, weapons, or decorative items.
- Unlockable content: Developers can offer users the ability to access premium content or features after making a purchase. This could include levels, chapters, advanced tools, or special modes.
- Subscriptions: Some apps offer subscription models where users pay a recurring fee to access premium content, services, or features on an ongoing basis. This is common in apps providing streaming, fitness, productivity, or entertainment services.

- Consumables: Consumables are items that can be used up or consumed, often within gaming or gamefied learning, fitness, or health apps. Examples include extra lives, energy, tokens, or points needed to advance to higher levels, as well as other resources that can help users progress faster.
- DLC (downloadable content): Additional content packs, expansions, or updates that users can purchase to extend the gameplay or functionality of the app.

Why are IAPs important?

IAPs can significantly boost an app's revenue potential, especially if the app has a dedicated user base and offers compelling and valuable content through the purchases. However, it's important for developers to strike a balance between providing value to users and avoiding an overly aggressive monetization strategy that could negatively impact user experience or perception.

In addition, the design and implementation of IAPs should adhere to the guidelines and policies set by platform-specific app stores (such as Apple's App Store or Google Play Store) to ensure a fair and transparent user experience

What is an advertising network?

An advertising network, or ad network, connects businesses that want to run advertisements with websites that wish to host them. The principle attribute of an ad network is the gathering of ad space and matching it with the advertiser's needs.

The term ad network is media neutral, but is often used to imply "online ad network" since the marketplace of aggregated publisher ad space and advertisers is increasingly found on the Internet. The crucial difference between traditional and online ad networks is that online ones deliver advertisements to the public through an ad server. Delivering ads through one central hub allows the business owner to use various methods of targeting, tracking and reporting that don't exist with traditional media alternatives.

How do they work?

Ad networks work with publishers all over the Web, helping anyone who has unsold inventory, or ad space, and wishes to monetize their offerings. The networks then aggregate this inventory, package it and sell it to advertisers (1).

Pros and cons

The benefits of using ad networks are numerous for both content providers and advertisers. Content providers find them an easy and reliable way to sell inventory, although the revenue is typically less than what they could earn selling the space themselves. Advertisers also like the ease of use. With minimal effort, they can purchase a campaign that targets a specific group of consumers on websites throughout the world. Ad networks are also known for flexible payment models and cost efficiencies.

The downside to advertisers is limited control over ad placements. The possibility exists that ads could appear next to inappropriate content. Many advertisers have also complained that their own

campaign analytics often do not match up with the metrics provided by the networks (2). Despite these downfalls, many businesses find ad networks to be an affordable and efficient way of reaching consumers.

Pricing structure

Ad networks offer many different pricing models to suit the needs of different businesses.

- CPM - cost-per-mille (Latin for "thousand") is the simplest of ad delivery options. Advertisers pay a price for every 1,000 impressions delivered. An ad served on someone's browser is an impression. Whether or not the user clicks on the ad has no bearing on the price.
- CPC - cost-per-click is a performance-based model. Impressions served do not factor in cost. Instead, advertisers pay for every click they get on an ad campaign. This method of pricing often depends upon advertisers bidding on the maximum amount they will pay for a single click, a model that's become very popular on Google AdSense and Adwords. The downside of CPC is the uncertainty of how often the ad will be served, however, the low risk and focus on performance still make it an attractive option to most marketers.
- CPA - cost-per-acquisition, or cost-per-action, takes the performance-based model even further by guaranteeing advertisers will only pay for a specific action or conversion by the user. This ad delivery metric is often a key performance indicator for marketers when evaluating return on investment (ROI). A campaign that cost $250 and resulted in 25 sales means the CPA was $10. If the per-sale-profit exceeds $10 then the campaign was a success.

Ad targeting capabilities

Targeting capabilities for advertisers have increased dramatically over the years due to massive quantities of consumer data that have become available to the marketing community. Targeting capabilities may include:

- Age
- Gender
- Income
- Geography
- Behavior - displays relevant ads to users based on content they've clicked on while visiting several sites.

Google Display Network (GDN)

Google has a vast network of websites advertisers can display ads on, from the New York Times down to the smallest blogs on the Web. They offer several options businesses can use to target audiences. They may even combine two or more targeting methods in the same campaign (3).

- Contextual Targeting - Google delivers advertisements to users based on the content they consume. The content aggregator - Google, in this case - assigns labels to specific websites and matches it with keywords associated with a specific ad. GDN can also serve an ad that contextually fits websites where a user has previously been. For instance, a user might visit an outdoor sporting goods website and then click away to look at political news site.

Contextual targeting could be used to show an ad for family-sized camping tents to that user on the political site.

- Placement Targeting - marketers can choose which websites or webpages the ad serves on.
- Remarketing - users who visit a website are then shown advertisements for that website as they browse other areas of the display network. Remarketing can be a highly effective tactic to market to people who may have abandoned their shopping carts on your website. To increase the chance of a sale, you should combine this with an abandoned cart email.
- Interest Categories - Marketers can target people based on interests they reveal in the Web content they visit.
- Topic Targeting - Similar to interest categories, topic targeting allows marketers to pick a specific topic and Google will display the ads on quality sites related to that subject.
- Geographic and Language - Marketers can distribute their ads within a specified region or postal code and define the native language of the audience.
- Demographic Targeting - Ads are distributed to an audience based on age and gender.

Guide to Add Ads to your Apps

Google AdMob: A common way to add mobile advertising to your mobile app

How do you integrate ads into your mobile app? Let's explore a popular way to do this, which is Google AdMob.

AdMob, which stands for "Advertising on mobile", is a company originally founded in 2006. The company is based in Mountain View, California, USA, and Google acquired it in 2009. It is one of the largest mobile ad networks.

With the AdMob ad network, you can integrate ads into your mobile apps relatively easily. Apart from incorporating in-app ads, you get actionable insights. You can also access powerful tools to use AdMob with your mobile apps, and these tools are easy to use.

At the time of writing this, the AdMob website claims that 81% of the top 1,000 Android apps use it. The company claims that more than 1 million apps use AdMob, and over 1 million Google advertisers are on AdMob.

Hire expert developers for your next project

Trusted by

GET STARTED NOW

Why should you as an app developer explore AdMob? It offers SDKs for both Android and iOS, which makes it considerably easy for you to integrate ads.

How you can integrate ads into your Android app using AdMob

How can you use AdMob to integrate ads into your Android app? We will now explore this.

You can do this in two different ways, and the difference is whether you are using Google Firebase, i.e., the "Mobile-Backend-as-a-Service" (MBaaS) from Google. If you aren't using Firebase, then you need to do the following:

- Create a Google AdMob account and register your app.
- Import the Mobile Ads Android SDK.
- Update your AndroidManifest.xml file with the AdMob "App ID" for your app.
- Initialize the Mobile Ads SDK.
- Selec ad formats, e.g., interstitial ads, banner ads, native ads, etc.

Read the Google AdMobs Mobile Ads SDK (Android) "Get started" guide for more details.

If you are using Google Firebase for your Android app development, then you need to do the following:

- Sign-up for a Google AdMob account, and register your app.
- Connect your app to a Google Firebase project, which includes configuring it for Firebase. This requires you to add Google's Maven repository and download the Firebase SDK for Android.

The subsequent steps are similar to the option without Firebase, which includes the following:

- Importing the Android Mobile Ads SDK;
- Updating the AndroidManifest.xml file with the AdMob "App ID";
- Initializing the SDK and choosing an appropriate ad format.

Read the Google Firebase AdMob Android guide named "Get started in Android Studio" for more information.

Integrating ads into your iOS app using AdMob

Are you offering an iOS app too? You would likely integrate ads into that too, and AdMob can help. Once again, there are two options. You can use Google Firebase to host the mobile backend, alternatively, you can use another MBaaS platform like AWS Amplify.

If you are using Google Firebase, then you need to take the following steps:

- Install the Firebase SDK.
- If you don't have an AdMob account already, then create one.
- You need to register your app with Firebase.
- Link the app to a Firebase project.
- Import the Mobile Ads iOS SDK.
- Update your "Info.plist" file adding your AdMob "App ID".
- Initialize "Mobile Ads".
- Choose an ad format that works for you from options like banner, native, etc.

Read the Firebase AdMob iOS “Get started” guide for more insights.

Hire expert developers for your next project

62Expert dev teams,
1,200 top developers

300+Businesses trusted
us since 2016

GET STARTED NOW

Are you using an MBaaS platform other than Firebase? You can still use the AdMob iOS SDK with Xcode 10 or higher, and you need to target iOS 8.0 or higher. Create an AdMob account if you don’t have one and register your app.

The remaining steps are similar to the option with Firebase, i.e., you need to install the AdMob iOS SDK, update your “Info.plist” file, and initialize the AdMob iOS SDK. Choose the ad format you want, and you are ready!

Read the AdMob iOS SDK “Get started” guide for more information.

Integrate ads into your website

You might want to integrate ads into your website, but, how do you do this? Well, you can use Google AdSense, which is a program run by Google. Website owners can use it to serve text, image, video, or interactive media ads.

Google administers, sorts, and maintains the ads. The company uses powerful analytics to serve ads that are relevant to the audience of your website, moreover, Google ensures that only high-quality ads are served.

You as a website owner can block ads that you don’t want, customize where ads will appear, and choose the kind of ads you want to display on your website.

With AdSense, you are connected to relevant advertisers. You get paid using standard methods, e.g.: CPC (cost per click), CPM (cost per thousand impressions), etc. Integrating AdSense is easy, and you need to take the following steps:

- Ensure that your website complies with the AdSense standards.
- Apply to AdSense.
- Configure your ads.
- Copy-paste the AdSense code onto your site.
- Update your privacy policy, and verify your address.

Read “How to add Adsense to your website” for more insights.

Offering excellent user experience even while integrating ads into your app

Now that you know how to integrate ads into your app, I want to remind you about the need to offer a great user experience. Many users don't receive ads well, therefore, you need to offer an excellent user experience so that they continue to use your app. How do you do this?

If you are integrating ads into your website, then ensure that you design a great user interface (UI). Do the following:

Hire expert developers for your next project

Trusted by

GET STARTED NOW

- Design the UI in a way that mirrors the real world.
- Provide control and freedom to users.
- Maintain consistency in your UI design and follow applicable standards.
- Enable users to recognize task-related relevant information instead of making them recollect it.
- Provide flexibility and efficiency to users.
- Minimize clutter, and design an aesthetically pleasing UI.
- Minimize errors and deliver error messages that are easy to understand.
- Display help and documentation prominently.

Read "User interface design guidelines: 10 rules of thumb" for more insights.

Integrating ads into your mobile app? You need to follow the appropriate best practices for UI design, e.g.:

- Design your UI by following platform-specific guidelines. You need to follow the "Material Design" guidelines when you design the UI of an Android app, whereas, you need to consult the "Human Interface Guidelines" while designing an iOS app UI.
- Choose an appropriate mobile navigation menu pattern that suits your app. Our guide "Mobile navigation menu examples" can help you.
- Select an appropriate color scheme for your mobile app UI. You can consult "8 trends in mobile app color scheme" for more insights.
- Design icons that help your users to access the features of your app easily. We have a useful guide for this, which you can access in "How to design the perfect icon for your mobile app?".

Planning to integrate ads into your app?

The latest technologies like AR and VR are contributing significantly to the growth of mobile advertising industry. The mobile ad spending globally is expected to reach 413 billion US dollars by 2024. However, as you can see, integrating ads into your app while delivering excellent user experience can be hard.

I recommend that you work with a reputed and trusted development partner, and read our guide "How to find the best software development company?" to find such a partner.

DevTeam.Space has a field-expert software developer community experienced in developing mobile applications delivering excellent user experience using the latest cutting-edge technologies.

You can partner with these mobile app developers by filling out this quick form. One of our account managers will contact you to assist with the further process of onboarding software developers, project planning, etc.

Frequently Asked Questions

How do I add ads to my app?

The most straightforward approach is to integrate Google Mobile Ads Lite SDK into your application. Your developer will know how to do this and understand the guidelines that your project needs to conform with in order to display ads.

How do I integrate ads into my website?

Most small to medium-sized websites choose to use Google Ads to do this. Simply instruct your developer to do this. Alternatively, you can go the route of finding another ad platform. Keep in mind to read all the documentation of whatever platform you choose to ensure it fits your requirements.

How much do apps make from video ads?

Total ad revenue depends on the number of ad impressions. However, the average amount made per click through mobile ad monetization in the United States currently stands at $2 cents.

A Publishers' Guide to Ad Networks

Introduction

Ad networks, or advertising networks, are platforms or intermediaries connecting advertisers with publishers or website owners. Their main purpose is to facilitate the buying and selling of advertising space or inventory across a network of websites or apps.

Without ad networks, publishers would be required to negotiate deals with each advertiser individually, which would be time-consuming and inefficient.

In this article, we'll cover what ad networks are, their role in the programmatic ecosystem, and how to choose the best ad network for your necessities!

What are Ad Networks?

The primary role of ad networks is to gather ad inventory that publishers haven't sold (often referred to as remnant inventory) and connect it with advertisers seeking ad placements.

The ad network acts as a trusted intermediary, handling the logistics of ad delivery, targeting, and payment collection on behalf of both parties.

By partnering with ad networks, publishers can access a network of advertisers interested in purchasing ad space. Ad networks streamline the process by consolidating multiple advertisers into a single platform, making it easier for publishers to connect with potential buyers.

This eliminates the need for publishers to individually approach and negotiate deals with each advertiser, saving them significant time and effort.

Ad network types

There are 6 main ad network types, such as:

- **Premium Ad Network.** Ad networks that offer inventory from premium publishers.
- **Vertical Ad Networks**. Topic-specific ad networks, such as travel, technology, or business.
- **Horizontal Ad Networks.** Ad networks that deliver ads to a large inventory base available to the advertiser and offer an extensive reach. Horizontal ads offer scale, reach, and multiple targeting opportunities for advertisers.
- **Inventory-Specific Ad Network**. Ad networks that provide a specific type of ad inventory, such as video or mobile.
- **Targeted Ad networks.** Ad networks offer specific targeting capabilities built into the ad server.
- **Affiliate Advertising Network.** These ad networks act as intermediaries between publishers and advertisers, providing a platform for publishers to discover and join different affiliate programs. They offer various advertisers and products across different industries, enabling publishers to choose the ones that align with their audience and content.

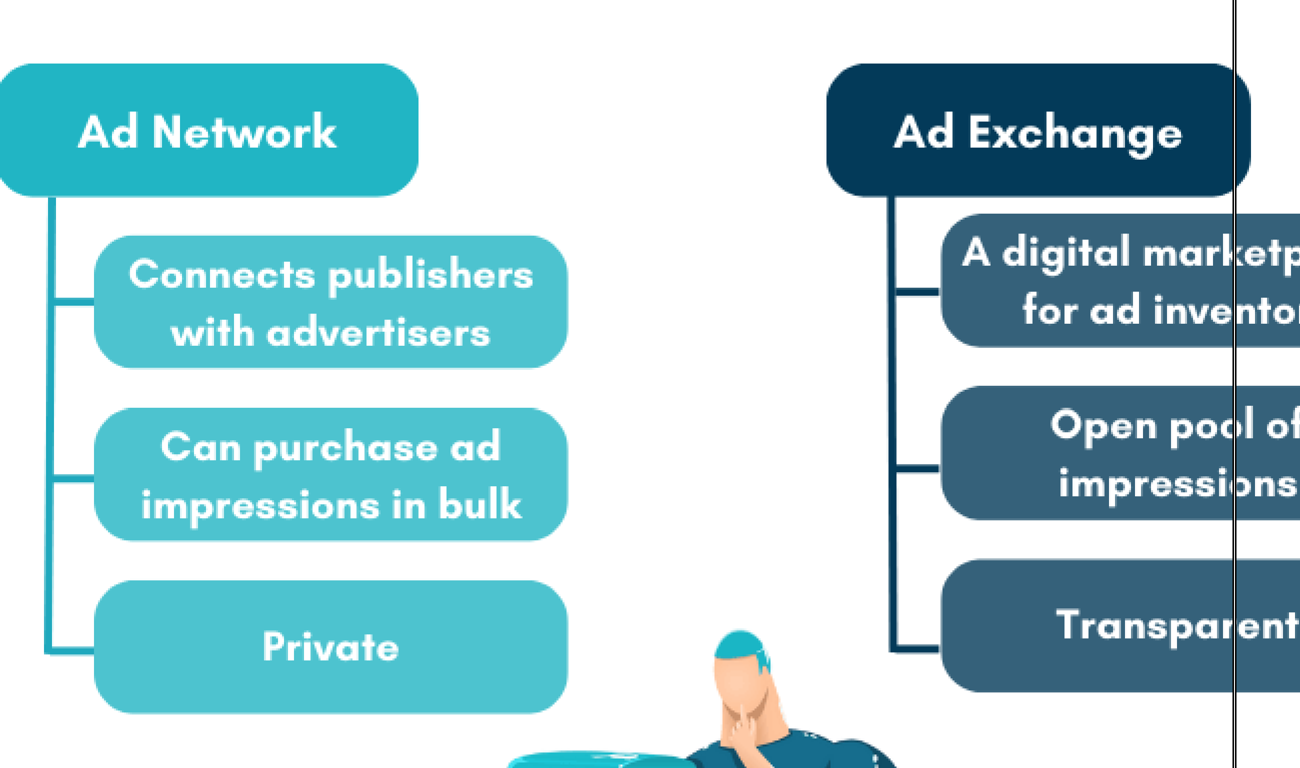
What's the difference between an ad network and an ad exchange?

An ad exchange is an online marketplace where programmatic advertising takes place. It enables advertisers to purchase ads across a wide range of mobile apps, mobile websites, and regular websites simultaneously.

Ad networks manage digital inventory from various publishers, purchase large quantities of ad impressions from ad exchanges, and then sell them to advertisers.

Ad exchanges provide a more transparent and efficient method for buying and selling digital ads. They use algorithms that enable publishers to receive the best prices for their ad impressions.

Simultaneously, advertisers can target specific audiences at the right time and in the appropriate context, eliminating the need to negotiate purchases directly with publishers.



What's the difference between an ad network and an ad server?

To recognize the difference between an ad network and an ad server, it's important to understand that ad networks (as well as publishers and advertisers) use an ad server as a part of their operations.

Ad servers are responsible for ad management, tracking impressions, and providing analytics. They store and serve the ads, ensuring they are displayed correctly on websites or mobile apps.

Ad servers also collect data on ad performance, such as impressions, clicks, and conversions, allowing advertisers and publishers to measure the effectiveness of their campaigns.

In contrast to ad servers, ad networks have a different role within the programmatic advertising ecosystem. Ad networks don't focus on ad management or storage.

Instead, they primarily facilitate transactions between advertisers and publishers. As mentioned, ad networks serve as intermediaries, connecting advertisers with available ad inventory from publishers.

The ad network relies on the ad server's capabilities to serve and track ads, ensuring they are displayed in the appropriate locations and optimized.

Simply put, the ad server serves as the underlying technology that enables advertisers to place their ads and assists publishers in efficiently handling these ads.

How Do Ad Networks Work?

Here are the 4 main steps that show how ad networks work:

1. Publishers install one or more ad networks on their websites or mobile apps.

This allows them to tap into the network's pool of advertisers and access additional opportunities to monetize their ad inventory.

2. Conversely, advertisers select the ad network(s) they prefer to work with to set up their ad campaigns. They create their campaigns and define specific parameters for each one.

These parameters include budget allocation, audience targeting criteria, and frequency caps to control how often an ad is shown to individual users.

3. When advertisers have their campaigns ready, the ad network considers their requirements.
4. The ad network then uses its algorithms and targeting capabilities to determine which publishers are the most suitable for displaying the advertiser's ads.

In summary, the ad network acts as a matchmaker, connecting the right advertisers with the appropriate publishers based on their needs and preferences.

How do ad networks operate in the digital advertising ecosystem?

Ad networks help publishers manage and monetize their ad inventory while providing advertisers access to a wide range of targeted ad inventory and optimization services. They play a vital role in the digital advertising ecosystem by facilitating efficient transactions and offering valuable services to both sides.

On the supply side

From the supply-side perspective, ad networks are crucial in helping publishers manage their digital ad inventory and maximize their revenue potential. Publishers have valuable ad space on their websites or mobile apps, and ad networks act as intermediaries between publishers and advertisers.

Ad networks assist publishers in organizing and categorizing the available ad space to make it more attractive to potential advertisers.

Ad networks also have a wide network of advertisers, allowing publishers to fill their ad inventory with relevant and high-paying ads.

By partnering with an ad network, publishers simplify monetizing their ad space. Instead of dealing with multiple advertisers individually, publishers can rely on the ad network to handle negotiations, ad delivery, and payment collection on their behalf.

This saves publishers time and resources while ensuring a steady revenue stream from their ad inventory.

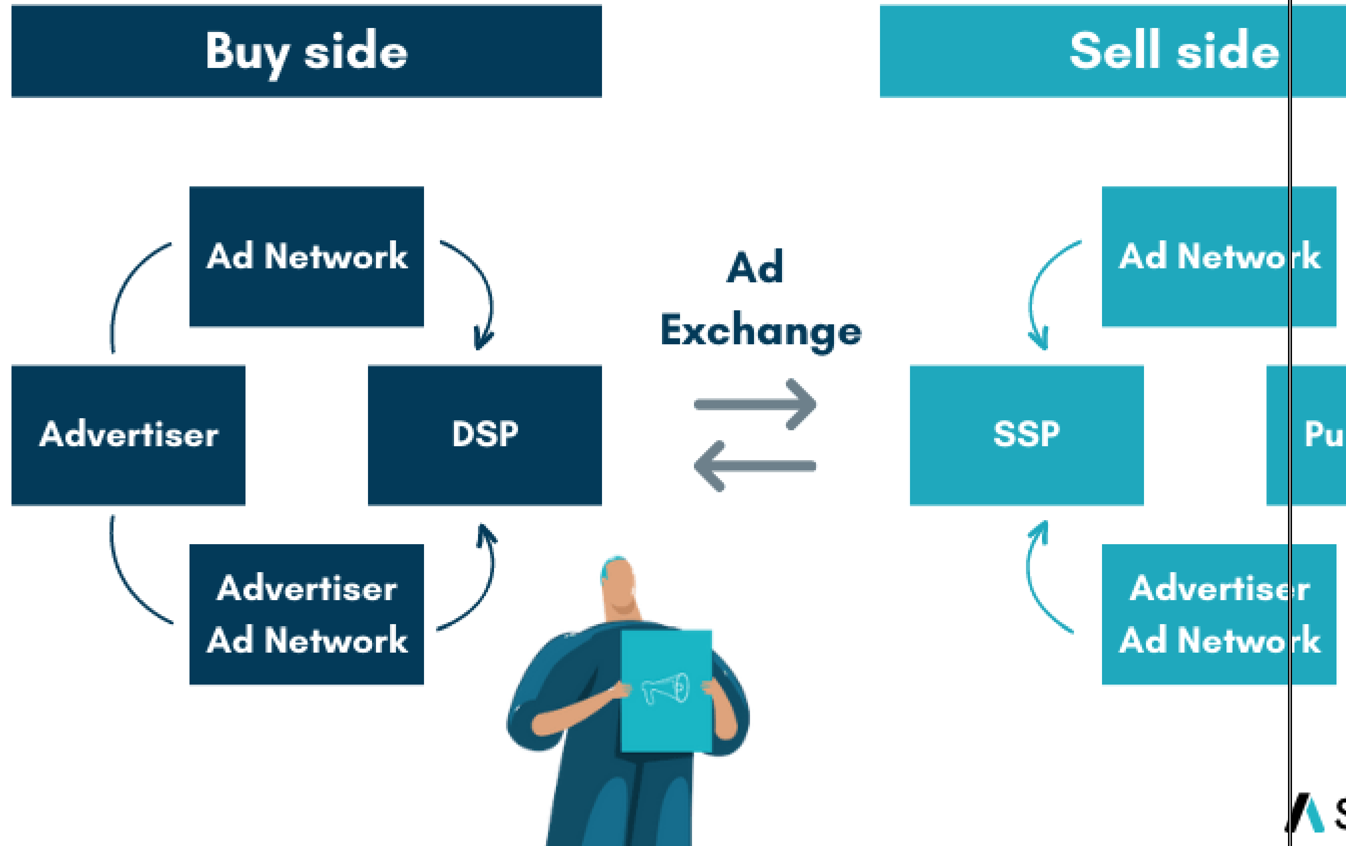
On the demand side

Regarding the demand side, ad networks provide advertisers access to a broad range of ad inventory across different publishers, websites, and mobile apps. Advertisers can leverage ad networks to reach their target audience effectively.

Ad networks offer targeted advertising options to advertisers, such as demographic targeting, interest-based targeting, and contextual targeting. These targeting options enable advertisers to display their ads to the most relevant audience, increasing the likelihood of engagement and conversions.

Furthermore, ad networks optimize campaigns on behalf of advertisers.

Through algorithms and data analysis, they monitor the performance of ads in real time and make necessary adjustments to maximize campaign effectiveness. Ad networks provide insights and analytics to advertisers, helping them refine their strategies and achieve a better return on investment (ROI).



Benefits of Using Ad Networks for Publishers and Advertisers

The advantages of using an ad network depend on the user's perspective.

For publishers: ad networks provide means to secure buyers for their unsold ad space, effectively monetizing inventory that would otherwise go unused.

However, the revenue generated through ad networks is generally lower than what publishers could earn through direct sales.

For advertisers: the benefit from ad networks comes from gaining access to a wide range of inventory options that match their target audience and budget considerations.

Ad networks help advertisers find suitable ad placements that align with their desired audience demographics and fit within their allocated advertising budget.

How to Choose the Right Ad Network for Your Website?

When selecting the best ad network, there are several factors to consider. The following 7 points highlight the key considerations:

1. Size of advertiser network.

The ad network's advertiser base size is vital to evaluate the potential reach and ad fill rates. A more extensive network may offer more opportunities for advertisers, increasing the likelihood of better revenue for publishers.

2. Quality of ads in the network.

Assessing the quality of ads within the network is crucial. High-quality ads enhance user experience, leading to higher engagement and conversion rates. Ensuring that the ads align with your website's content and target audience is essential.

3. Variety of available ad formats.

Look for an ad network that offers a diverse range of ad formats. This lets you choose formats that best suit your website's layout and user experience. Having various options gives you the flexibility to experiment and optimize ad performance.

4. Compensation and payment terms.

Evaluate the compensation and payment terms offered by the ad network. Factors such as revenue share, payment frequency, and minimum payout thresholds should be considered to ensure they align with your financial goals and requirements.

5. Stability and User-Friendliness.

Select an ad network with a stable, reliable platform and minimal service disruptions. Additionally, prioritize a user-friendly interface and intuitive and easy-to-navigate tools. This will save you time and effort in managing your ad campaigns.

6. Additional support.

A reliable and responsive support team can be priceless in resolving any issues or concerns that may arise during your partnership with the ad network. They can provide guidance, answer questions, and address technical difficulties promptly, ensuring smooth operations and a positive experience for publishers.

A dedicated and knowledgeable support team can significantly contribute to the success of your ad campaigns and overall satisfaction with the ad network.

7. Underlying technology.

Analyze the underlying technology the ad network uses, as it is crucial in optimizing ad performance and revenue potential.

A robust and advanced technology infrastructure can enhance ad targeting capabilities, improve ad delivery speed, and provide comprehensive reporting and analytics tools.

How to Integrate Ad Networks?

Here are the 6 main steps that show how you can integrate ad networks with your website or app:

1. **Choose the right ad network.** Select the ad network that aligns with your requirements, such as target audience, ad formats, and revenue models. Research and compare different ad networks to find the ones that fit your needs.

2. **Sign up and create an account.** Register with the chosen ad network and create an account. This typically involves providing your website or app information and contact details and agreeing to the network's terms and conditions.

Maximize Your Ad Revenue With Header Bidding

- Reliable 60 day Payments
- No Minimal Commitments
- Ensured Ad Quality

1. Sign Up
2. Receive the following steps in your email
3. Implement Setupad tags
4. Receive the payment in 60 days

Fill in the form below
steps.

Name

John Doe...

Email Address

Email...

Your Website Url

You Website Url

Website's Google Analytics

Drag and

3. **Obtain ad network code/tags.** The ad network will provide you with specific code snippets or tags after signing up. These are usually JavaScript or HTML codes that you must implement on your website or app to display the ad placements.

4. **Implement code/tags.** Place the provided code or tags in the appropriate sections of your website or app. This may involve modifying your website's HTML or integrating the code within your app's source code. An ad network may do it for you if this feature is available.

5. **Test and verify.** Once the code/tags are implemented, thoroughly test the integration to ensure that ads are displayed correctly and that they align with your website or app's design and user experience. Verify that the ads are loading properly and tracking the necessary metrics.

6. **Monitor and optimize.** Keep track of the ad performance and metrics through the ad network's reporting tools or dashboard. Analyze the data to optimize the placement, formats, and targeting for better revenue generation and user engagement. This may involve experimenting with different ad formats, positions, and targeting settings.

It's important to note that the specific steps and implementation process may vary depending on the ad network and your platform (website, mobile app, etc.).

Ad network documentation, guides, and support teams are valuable resources to consult throughout the integration process, as they can provide specific instructions and address any technical challenges you may encounter.

Ad Network Monetization Models

Online ad networks commonly employ 5 revenue models to generate income. These revenue models are CPM, CPC, CPA, CPI, and CPV. It's also possible to employ a revenue sharing or sponsorship model.

Let's take a closer look at each of these models:

Cost per Mille (CPM)

CPM is based on the number of ad impressions per thousand views.

Advertisers are charged a fixed rate for every thousand times their ad is shown, regardless of the number of clicks or actions taken. CPM is often used to enhance brand exposure and awareness.

Cost per Click (CPC)

Advertisers pay for every click their ad receives on the publisher's platform in this model.

The cost is determined by the number of clicks the ad generates, regardless of the number of impressions it receives.

Cost per Action (CPA)

With this model, advertisers are charged based on user actions.

The advertiser defines chargeable actions like clicks, sales, downloads, or other desired user engagements. Advertisers pay when users complete these predefined actions.

Cost Per Install (CPI)

CPI is primarily used in the mobile app advertising space. Advertisers pay for each installation of their app on a user's device.

This model is commonly employed by app developers or businesses looking to drive app downloads and installations.

Cost per View (CPV)

This model involves advertisers paying for each ad view. It's commonly used in video campaigns, where advertisers are charged for each time a user views their video ad.

Revenue Sharing

Publishers can receive a percentage of the revenue generated from the ads displayed on their platform. Most of the time, the revenue share is 80:20.

Sponsorships

Publishers can partner with brands for sponsored content or specific advertising campaigns, earning a fee for promoting their products or services.

Ad Network Optimization Strategies

Ad network optimization strategies enable to maximize the revenue generated from ad placements on publisher websites or apps. These strategies involve different tactics to improve the performance and effectiveness of ad networks, such as:

- Experimenting with different ad formats, sizes, and placements to find the most engaging and effective combinations.
- Segmenting audience based on demographics and interests to deliver more targeted and relevant ads.
- Conducting A/B tests to compare different versions of ads and identify the most effective elements.
- Analyzing performance data to make data-driven decisions and optimize ad inventory.
- Implementing smart ad refresh and frequency capping to strike a balance between maximizing ad impressions and providing a positive user experience.

Ad Fraud and Ad Network Security

Ad fraud refers to deceptive activities in the digital advertising industry that aim to exploit publishers and advertisers by generating fraudulent impressions, clicks, or conversions.

By staying proactive and responsive to emerging threats, ad networks work diligently to maintain a secure environment where advertisers can confidently reach their target audience and publishers can monetize their ad inventory without falling victim to fraudulent practices.

- **How do ad networks combat ad fraud?**

To tackle this issue, ad networks can employ different strategies, such as traffic verification and filtering systems to detect and eliminate fraudulent or low-quality traffic. Advanced fraud detection tools, powered by machine learning algorithms, identify abnormal patterns and fraudulent activities in real-time.

Ad networks also rely on manual review processes to assess publishers and their inventory for legitimacy. They employ viewability metrics, verification technologies, and anti-fraud partnerships to ensure real users see ads and prevent fraudsters from manipulating ad view counts.

Ad Network Performance Tracking and Reporting

Ad network performance tracking and reporting are crucial to evaluate the effectiveness and success of ad campaigns running through ad networks.

Publishers can track essential metrics in real-time by using analytics tools and reporting dashboards provided by the ad network or third-party solutions.

Below is an example of Setupad's reporting client dashboard.

By using advanced reporting dashboard, publishers can identify the best-performing ad units, optimize their ad inventory, and make informed decisions about ad placement and formats that resonate well with their audience.

It also allows publishers to track revenue streams, calculate earnings, and make data-driven decisions about their ad network partnerships.

How to Assess Ad Network Performance and ROI?

There are 4 key performance indicators (KPIs), which help to maintain and analyze ad network performance:

- **Click-through rate (CTR)** measures the percentage of users who click on an ad, indicating its effectiveness in generating interest.
- **The conversion rate** calculates the number of users who completed a desired action, such as purchasing or signing up.
- **Cost per acquisition (CPA)** evaluates the cost of acquiring each customer, helping determine campaign profitability.
- **Return on ad spend (ROAS)** quantifies the revenue generated relative to ad spend, offering insights into campaign profitability.

Additionally, tracking overall campaign performance, impressions, reach, and engagement can also help to evaluate ad network performance and ROI.

Types of Ad Networks

Native ad networks

Native ad networks (e.g., Adsterra) connect publishers with advertisers who want to deliver native ads. Native ads are designed to blend seamlessly with the surrounding content, providing a non-disruptive and engaging user experience.

These ad networks provide tools and resources to help publishers optimize their native ad placements, ensuring that the ads are relevant, contextual, and visually appealing.

Video ad networks

Video ad networks (e.g., Publift) specialize in delivering video ads to publishers. They connect publishers with advertisers who want to display video ads on their platforms.

Video ads can be displayed within the video content (instream), as standalone video ads (outstream), or within standard display ad units (in-banner). Video ad networks provide access to a wide range of video ad formats and offer advanced targeting and optimization features.

Mobile ad networks

Mobile ad networks (e.g., AdMob) focus on delivering ads on mobile devices such as smartphones and tablets. They connect publishers with advertisers who want to reach mobile users through mobile apps (in-app advertising) or mobile websites.

Mobile ad networks offer different ad formats, including display ads, interstitials, native ads, and rewarded ads. These networks often offer advanced targeting options based on user behavior, location, and demographic information.

Connected TV (CTV) ad networks

Connected TV (CTV) ad networks (e.g., Magnite) deliver internet-connected television ads. They connect publishers with advertisers who want to reach viewers through smart TVs, streaming devices, and other CTV platforms.

As more viewers shift towards streaming services and CTV platforms, this type of ad network allows publishers to monetize their CTV inventory. CTV ad networks can offer advanced targeting capabilities and help publishers optimize their ad placements

The ultimate guide: Ad placements, formats & specs in social media

Facebook

Facebook for business recommends that you include advertising on Instagram if you are planning on advertising on Facebook. In this way, you optimize delivery. If you are choosing automatic placements in Ads Manager it is preferable as Facebook itself optimizes your budget on the placements that work best for maximum results. In the survey "Swedes and the Internet 2020" – Facebook is the most used social channel in all age groups 26 years and up in Sweden.

Note: Apple has announced new iOS 14 policy requirements that may affect where your ad will be displayed. Learn more.

Facebook News Feed: Your ads appear in the news feed on your computer when people access the Facebook site on their computers. Your ads appear in the news feed on mobile devices when people use the Facebook app on mobile devices or open the Facebook website in a mobile browser.

Instagram feed: Your ads appear in the mobile feed when people use the Instagram app on mobile devices. Instagram flow ads are only shown to people who use the Instagram app.

Facebook Marketplace: Your ads appear on the Marketplace homepage or when someone browses the Marketplace on the Facebook app. Learn more about ads in Marketplace.

Facebook video feeds: Your video is displayed between organic videos in video-only environments on Facebook Watch and in Facebook's news feed.

Facebook right column: Your ads appear in the right column of Facebook. Ads in the right column are only displayed to people who connect to Facebook on their computer.

Instagram Explore: Your ads appear when someone clicks on a photo or video. Learn more about ads in Instagram Explore.

Messenger Inbox: Your ads appear on the Messenger Home tab. Learn more about ads in Messenger.

Messenger sponsored messages: Your ads appear as messages to people who have an existing conversation with you in Messenger. Learn more about sponsored messages. Learn more about sponsored messages.

Stories: Stories on Facebook, Instagram and Messenger allow you to target personalized ads to your target audience and optimize your reach to maximize the number of people you can reach. Even in placements below, there are opportunities to reach your target group in a targeted way.

Your full-screen vertical ad appears in people's stories like:

- Facebook Stories: Your ads appear in people's stories on Facebook. Learn more about ads in Facebook Stories.
- Instagram Stories: Your ads appear in people's stories on Instagram. Learn more about ads in Instagram Stories.
- Messenger Stories: Your ads appear in people's stories on Messenger. Learn more about ads in Messenger Stories.

Audience Network: Audience Network extends Facebook's people-based advertising beyond the Facebook platform. With Audience Network, publishers can make money by showing ads from Facebook advertisers in their apps. This is an effective way to reach more people who use other apps and get an increased reach at a lower cost than if you had only chosen Instagram and Facebook. In other words, the advantage is that you can reach more people without having to spend more budget. There are two types of categories that your ads through Audience Network covers:

- **Audience Network Native, Banner and Interstitial:** Your ads appear in Audience Network apps. Find out more about banner ads, interstitial ads and native ads.
- **Audience Network Rewarded Video:** Your ads appear as videos that people can watch in exchange for rewards in an app (such as currency or in-app items). Find out more about rewarded video.

Facebook in-stream videos: Your ads appear in Video on Demand and in a select group of approved partner live streams on Facebook. Learn more about in-stream video.

Facebook search results: Your ads appear next to relevant Facebook and Marketplace search results.

Facebook Instant Articles: Your ads appear in Instant Articles within the Facebook mobile app. Learn more about Instant Articles.

One conclusion of Facebook's ad placements is that there are enormous opportunities to reach your target audience in the exactly right mood, time and type of digital behavior. Therefore, it's important that you make sure to adapt your content to the different placements and include it in your campaign setup to ensure an optimal customer experience. To cover maximum exposure within each placement, you need to optimize your ads creative into the formats 1:1, 4:5, 16:9 and 9:16.

Snapchat

Depending on whether your goal with the campaign is to drive traffic, increase the number of conversions or create engagement, there are placements in Snapchat that are made to match your goal. According to "Swedes and the Internet 2020", Snapchat continues to be used mainly by generation Z in Sweden, where 9 of 10 users are between 16 and 25 years old.

A Snapchat ad can be up to ten seconds long and is available in these formats: Video, GIF, Picture & Cinemagraphs.

Now, let's have a look at the ads formats.

Single Image or Video Ads: A Single Image or Video Ad is a full screen ad that can be used for many objectives. Simply add an attachment and enable Snapchatters to swipe up and take action. Learn more about Single Image or Video Ads

LinkedIn

If you are a B2B company, LinkedIn is a great channel for advertisement since you can be extremely granular with your audience targeting and reach businesses and decision makers that are important to you.

LinkedIn has over 740 million members and 55 million companies listed in 200 countries. They also have more than 260 million monthly active users and of those LinkedIn users who are engaging with the platform monthly, 40% access it on a daily basis. However, LinkedIn is used sparingly, so you only have a few minutes to make an impact. Users only spend about 17 minutes on LinkedIn per month.

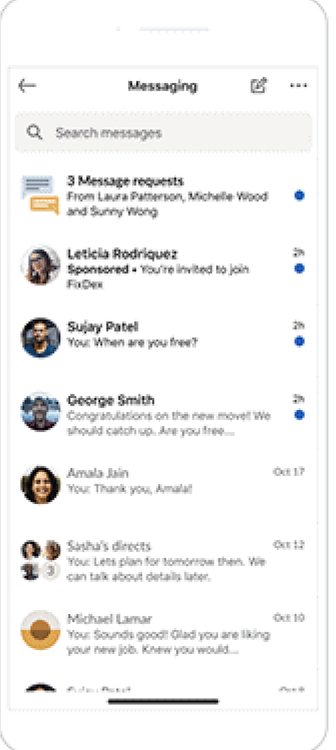
Let's have a look on how to do it.

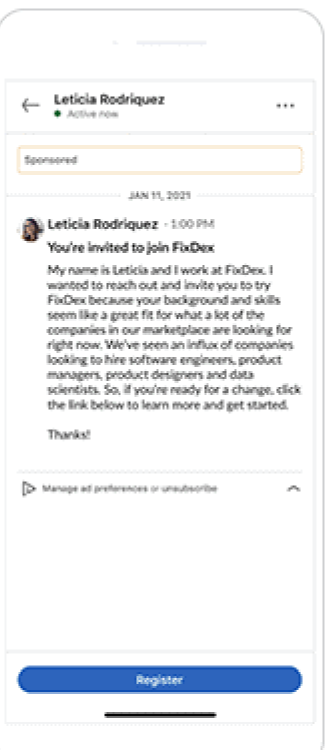
Sponsored Content: Reach a highly engaged audience in the LinkedIn news feed.

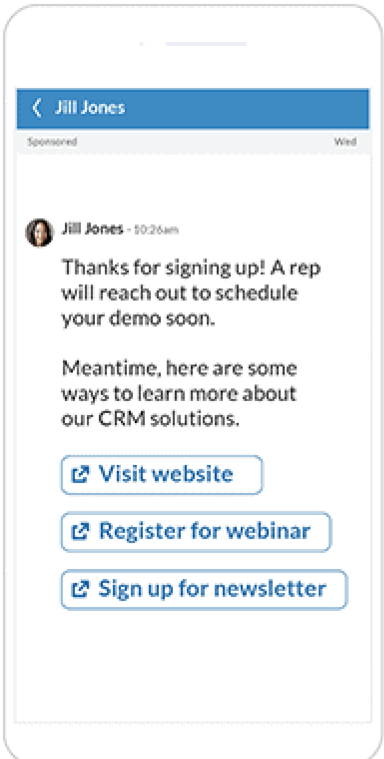
- **Single Image Ads:** Include one image and appear directly in the LinkedIn feed of members in your campaign's target audience. Learn more about Single Image Ads here.
- **Video Ads:** Sponsored Content ad format that appear in the LinkedIn feed. Learn more about Video Ads here.
- **Carousel Ads:** Display multiple images in succession in a single, carousel-style ad. They appear in the LinkedIn feed of members you want to reach. Learn more about Carousel Ads here.

Sponsored Messaging: Engage your audience in LinkedIn Messaging, where professional conversations happen. There are two types of Sponsored messaging on LinkedIn:

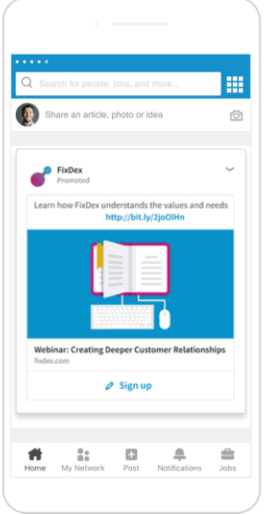
- **Conversation Ads:** Engage your prospects in LinkedIn Messaging. Learn more about Conversation Ads here.
- **Message Ads:** Send direct messages to your prospects to spark immediate action. Learn more about Message Ads here.

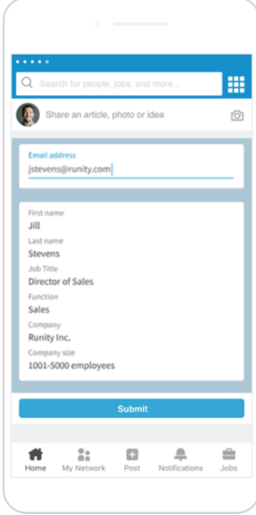






Lead Gen Forms: Collect even more quality leads from your ads on LinkedIn with seamless pre-filled forms. Learn more about Lead Gen Forms here.

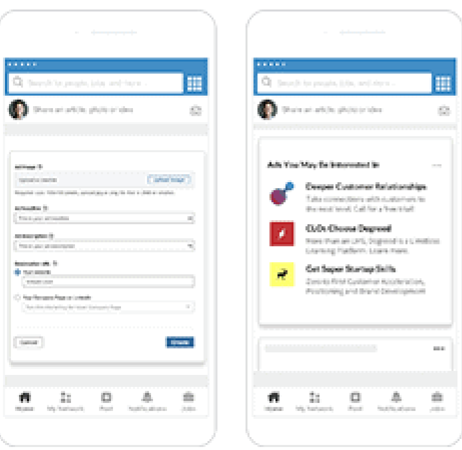


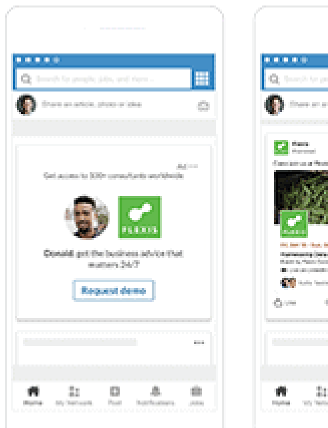


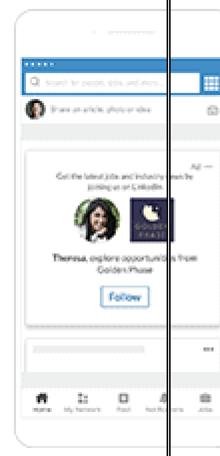
Text and Dynamic Ads: Run ads in the LinkedIn right rail. There are three types of Text and Dynamic Ads on LinkedIn:

- **Text Ads:** Drive new customer to your business – on a budget that works for you through LinkedIns self-service pay per click (PPC) advertising platform. Learn more about Text Ads here.
- **Spotlight Ads:** To drive conversions, use spotlight ads to showcase your product, service, event and more to increase traffic to your website or landing page. Learn more about Spotlight Ads here.

- Follower Ads: To build brand awareness, use follower ads to promote your LinkedIn Page to seamlessly acquire followers. [Learn more about Follower Ads here.]







Pinterest

Ads on Pinterest are called “promoted pins” and look the same as regular pins, but it says “sponsored at the bottom. The advantage of advertising on Pinterest is that it is a low competition advertising opportunity opened up in 2019 in Sweden. The channel has a buying-oriented target group, so there are good opportunities to drive conversion at the perfect mood!

Pinterest Business account holders can choose to promote existing best-performing pins, create a new image or video, or even promote an image that’s been pinned from a website.

Once an ad is set up in Pinterest Ads Manager, it will appear in users’ Home feeds and relevant search results. Let’s deep dive into the different formats.

Static Pins and ads: Static Pins and ads feature only one image. These specs apply to both organic Pins and ads.

Max. width video ads: Max. width video ads are videos that expand across people’s entire feed on mobile. Max. width video ads are only available as a paid format. These specs only apply to max. width video ads.

Standard width video Pins and ads: Standard width video Pins and ads are videos that are the same size as a regular Pin. These specs apply to both organic standard width video Pins and ads.

App install ads: App install ads feature one image and allow people to download apps without having to leave Pinterest. They’re only available as a paid format. App install ads follow the same specs as static Pins and ads.

Shopping ads: Shopping ads feature one image or a video at a time and allow people to purchase products they find on Pinterest. Shopping ads follow the same specs as static Pins and ads.

Collections Pins and ads: Collections Pins and ads appear as one main image above three smaller images, in feeds on mobile devices. These specs apply to both organic collections Pins and collections ads.

Story Pins: Story Pins appear as a set of multiple videos, images, lists and custom text in a single Pin. Story Pins are only available as an organic format.

Explore Pinterest's creative specs and requirements, from file types to character counts here.

TikTok

In recent years, TikTok has grown enormously since it was launched in 2016. The app merged with Musical.ly in 2018, and has been on the top lists of the most downloaded apps in both the App Store and Play Store. Today, the app has over 500 million users per month.

Advertising on TikTok has only just begun, because competition is lower among your target group and there are great chances that your company will gain ground on the platform in this way. Therefore, by hooking on to trends, you can easily grow and take a place with the target group.

Let's have a look at what formats and placements TikTok has to offer.

Brand Takeover: Is format adapted for brand building advertising- effective towards the target group. Immediately hook user attention with full-screen static or dynamic display, delivering strong visual impact for your brand through a full-screen visual experience.

Twitter

On Twitter, you can easily reach a relevant target group in such a way that you can identify it by using specific keywords, hashtags, interests, engagement in topics that are trending and following specific accounts. Some formats and placements can be perceived as more flexible
than Facebook's. So which ad formats do we have here?

Promoted Tweets: Two variants. The first option is to further increase engagement for a tweet you posted that garnered high engagement. The second option is to boost a post so that only the target audience you are targeting is seen. Run a poll, add a GIF, or promote your account. These formats work best with engagements, reach, and followers campaigns.A great way to get potential followers to discover you. Learn more here.

Youtube

YouTube has more than 2 billion logged-in monthly users and is the second largest search engine after Google. With YouTube ads, you can reach potential customers and encourage them to take action while watching or searching for videos on YouTube. The best part? You only pay when they show interest.

Although your ad will appear on YouTube, you'll manage your campaign using Google Ads, an advertising platform used by businesses running ads on Google and its advertising network – which includes YouTube.

Let's deep dive into YouTube's advertising formats.

Display Ads: Appears to the right of the feature video and above the video suggestions list. For larger players, this ad may appear below the player.

Overlay Ads: Semi-transparent overlay ads that appear on the lower 20% portion of your video.

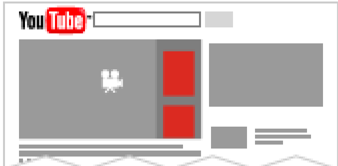
Skippable Video Ads: Allow viewers to skip ads after 5 seconds. Inserted before, during, or after the main video.

Non-skippable Video Ads: Must be watched before your video can be viewed. These ads can appear before, during, or after the main video.

Bumper Ads: Non-skippable video ads of up to 6 seconds that must be watched before your video can be viewed.

Sponsored Cards: Displays content that may be relevant to your video, such as products featured in the video.

Viewers will see a teaser for the card for a few seconds. They can also click the icon in the top-right corner of the video to browse the cards.



Explore YouTube's advertising formats further here.

That was "all" for today! We hope you enjoyed the reading and gathered some valuable insights about how to maximize your marketing mix and social media marketing strategy by taking advantage of all these amazing opportunities out there. Good luck and as we usually say: Try, learn and above all develop to achieve the best results.

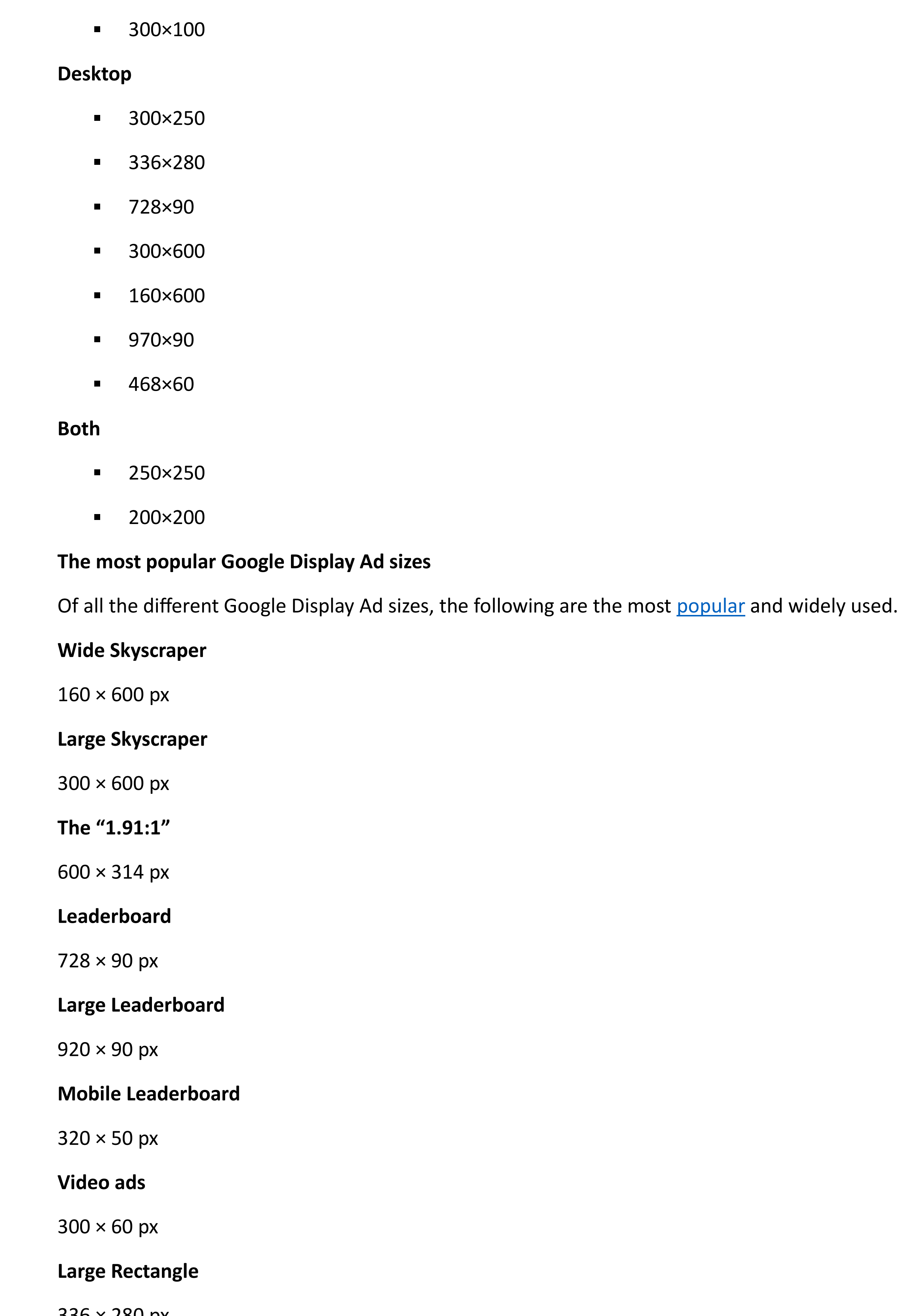
Don't forget to stay updated by subscribing to our newsletter!

Google Display Ad sizes

If you plan to use uploaded display ads, here is a list of some of the most commonly used display ad sizes you'll need to know:

Mobile

- 300×200
- 300×50



- 300×100

Desktop

- 300×250
- 336×280
- 728×90
- 300×600
- 160×600
- 970×90
- 468×60

Both

- 250×250
- 200×200

The most popular Google Display Ad sizes

Of all the different Google Display Ad sizes, the following are the most popular and widely used.

Wide Skyscraper

160 × 600 px

Large Skyscraper

300 × 600 px

The “1.91:1”

600 × 314 px

Leaderboard

728 × 90 px

Large Leaderboard

920 × 90 px

Mobile Leaderboard

320 × 50 px

Video ads

300 × 60 px

Large Rectangle

336 × 280 px

Medium Rectangle

300 × 250 px

Large Mobile Banner

320 × 100 px

Setting up a Google Display Ad campaign

If you're ready to get started, just follow this step-by-step checklist:

1. First, log in to your Google account.
2. Next, click **New Campaign**.
3. From there, select **Display**.
4. Then, Google will ask you to provide your website.
5. Be sure to give your campaign a name that's descriptive and memorable.
6. Select your language and your location targeting.
7. Determine your budget and bidding strategy.
8. Develop your targeting parameters for this campaign.
9. Provide your copy and upload creatives.
10. Then, review all of your desired settings.
11. Lastly, go ahead and publish your new campaign.

Video Source

5 best practices for Google Display Ads

To maximize your return on Google display ads, consider implementing these top five best practices:

Bid adjustments

Once you've collected enough data from your display campaigns, you can make informed decisions about the performance of your keywords, affinity audiences, and so on. Bid adjustments, which can

be set at either the ad group or campaign level, allow you to turn these performance insights into action.

A positive bid adjustment for an ad group instructs Google Ads to raise your maximum cost-per-click (CPC) bid when one of the ads in that ad group is eligible to be displayed. On the other hand, a negative bid adjustment tells Google Ads to decrease your maximum CPC bid for that ad group.

Simply put, bid adjustments are an effective way to increase the gains from your top performers and reduce your losses from poor performers.

Focus on top-performing search keywords

If you're interested in keyword targeting, start by using your most successful search keywords. If you have groups of keywords that have driven few conversions or clicks, you may want to test them on the display network as well.

Keep in mind that consumer intent between the two Google Ads networks can be different, but if your search ads are performing well in terms of clicks and conversions, then the keywords behind them have likely proven to be effective.

Analyze referral traffic

Google Analytics provides a wealth of information, and for display advertisers, the referral traffic report is extremely valuable. It shows which websites are linking to your website the most, essentially indicating which websites attract audiences who would benefit from your product or service.

These websites are ideal for displaying your ads since you know that the audience is relevant and therefore, you can be confident that you'll see returns from the impressions and clicks made on those ads.

Make your value proposition clear

Consumers have become highly familiar with display ads, so it's easy for prospects to overlook them while scrolling. To avoid missing out on opportunities and wasting resources, it's crucial to make sure your display ads stand out and grab the attention of your prospects.

The visual design of your ads, including the color scheme and typography, contribute to this, but the real key is having a clear and impactful value proposition that is prominently displayed on your ads.

Create strong headlines

Most responsive display ads will prompt you to write four required pieces of copy:

- A short headline (25 characters)
- A long headline (90 characters)
- A description (90 characters)
- Your business name (25 characters)

When it comes to headlines, keep in mind that Google Ads will not display both headlines simultaneously, and that Google Ads may exclude your description. This means that when your Responsive Display Ad (RDA) is displayed, there is no guarantee that your description will

accompany either headline. With this in mind, it's essential to ensure that both headlines effectively convey the unique value of your business or offer on their own.

Get your Google Display Ads noticed

Using Google Display Ads as part of your campaign can be a powerful tool for increasing traffic to your site and helping customers make their way through the conversion pipeline.

When integrated into a comprehensive marketing strategy, Google Display ads can amplify your campaign's success. It plays a crucial role in elevating brand awareness at the beginning of the buyer's journey, as well as re-engaging repeat customers with personalized promotions and offers. Clearly, display advertising is a valuable tactic and can be leveraged at various stages of the buyer's journey to maximize marketing outcomes.

To make sure your Google Display ads are as effective as possible, reach out to the experts at MarinOne today. We can help align your marketing efforts to take your campaign to the next level.

1) Define your business goals and KPIs

The first thing to do is define what you want to achieve using analytics.

What are your company's **business goals**, and how do you want your website or app to help you meet those goals?

For example, if you work for a healthcare provider, your main goal might be to sell more medical packages via online channels. And if you're a bank, you might be aiming to increase online mortgage sales.

After you have your goals in place, it's time to assign your **key performance indicators (KPIs)**.

Your KPIs should help you connect the dots between stages of the customer journey and between every touchpoint where your clients interact with your brand, like your app or website. The KPIs you set should reflect the complexity of this issue. In determining suitable KPIs, consider the industry you operate in and your business goals.

Here are some examples of KPIs:

- Conversion rate
- Cost of client acquisition (CPA / CPS)
- Customer retention rate
- Number of active users
- Revenue / revenue per user
- Percent of new / returning users

Choose both lagging and leading indicators. Lagging indicators are aspects such as revenue and cost that inform you about the results of past actions. Meanwhile, leading indicators tell you how likely it is that your product will meet a goal in the future.

Stay clear of vanity metrics that make the product or your company look good but provide little business value, such as the number of followers or newsletter subscribers. It's also better to choose a few metrics rather than measure as much data as possible.

If you operate in the banking industry, learn about KPIs to track that we describe in our ebook on How web and product analytics improve the customer journey in banking

At the beginning of your analytics journey, your KPIs and goals might lack accuracy. You may overestimate them since you don't have a lot of data that would allow you to develop more in-depth ideas. But you will be able to adjust them after collecting more data, and the conclusions you draw will be more precise and relevant to your business.

2) Analyze your website/product/app structure

Next, it's time to investigate the structure of your website, app, or product and list all its features.

Here is an example. A website for a medical provider could include the following features:

- Information about services offered and pricing
- An option for clients to log in to their accounts
- Contact form and chat to help clients reach out with questions
- Blog about medical topics

After you have listed the features, break them down into every action your users perform while using them.

Treat each feature as a separate funnel and list every action your user has to take in order to complete a particular task. This might be downloading a report, contacting your sales representative, logging in to their account, reading a specific article, learning about the offer, etc. Start from the essential features or sections of your website, then move to the ones you consider less important.

Also, when it comes to web analytics, you should use this step to analyze every section and diligently examine its environment. Seek out any potential hurdles like multi-domain structures, the usage of Ajax, frames, or any other technology that requires the creation of custom analytics scripts.

Write it all down – this list will come in handy in the next steps of the implementation.

3) Operationalize your goals and KPIs

Now it's time to translate your business objectives and KPIs into things you can do and measure using analytics software.

In the case of product analytics, you'll need to translate every user action (for instance, uploading a picture, playing a song, or completing a meditation session) into an event and decide what is going to trigger each of them.

Then you'll have to choose which events are more important than others and designate them as your goals, which are the vital actions that define your macro conversions. Your goals could be getting new users to create an account, filling out a contact form or upgrading their subscription.

Your product or site can have several goals because different sections may have different objectives.

Every micro conversion will also be a valuable source of information. Micro conversions are actions taken by users that lead toward macro conversions or are highly correlated with them. A micro conversion could be signing up for a newsletter, writing a comment on an article or downloading an ebook.

One of the most popular frameworks for defining Product Analytics metrics and goals is the HEART framework. It's a user-centric approach that allows you to measure user experience and evaluate advancements towards your primary objectives. The framework consists of five different categories of metrics:

- **Happiness:** measures attitude or satisfaction (typically through some sort of user survey)
- **Engagement:** measures the frequency, intensity, and depth of user interactions with a product
- **Adoption:** measures how easily, often, and quickly users adapt to the new features of the product
- **Retention:** measures how many existing users retain in a certain amount of time
- **Task success:** measures the effectiveness and efficiency of the tasks users complete within your application

With web analytics, you'll have to decide which metrics best align with your business goals and KPIs.

There is no need to measure everything that happens on your website. Take another close look at your business objectives and ensure you're going to track only things that are aligned with your goals and KPIs.

To collect data about your goals in web analytics, you should keep track of such data categories as:

- **URLs** – each time someone visits a particular URL, they trigger the goal. These are ideal for thank you pages, confirmation pages, and PDFs.
- **Time (visit duration)** – how long people stay on your site.
- **Pages per visit** – the number of pages each visitor sees before they leave.
- **Events** – event tracking lets you record clicks on both clickable and non-clickable elements of your website, usage of video and audio players, scrolling down, leaving the page, filling out a form without submitting it, downloads from your website and more. Even though these actions don't generate changes in the website's URL, they can still be recorded.

4) Create a tracking plan

Now it's time to write down everything you want to measure with your analytics and turn it into a tracking plan.

A tracking plan clarifies what events to track and why those events are necessary from a business perspective. Don't skip this step – you can use this document whenever you want to refresh what your main goals are, what features you want to measure, or make any changes to your analytics plan.

The most convenient way is to maintain a tracking plan in a spreadsheet.

When creating a tracking plan, consistency is essential. Develop an intuitive naming convention and stick to it when drafting your document. How you organize your documentation will depend on what you want to measure.

For implementing product analytics, one of the best solutions would be to divide the spreadsheet into the following categories:

- Product section
- Funnel name
- Step in the funnel name
- Event name
- KPI
- Trigger
- Event properties
- Event values
- Additional notes

Download a product analytics tracking plan template

In product analytics, it's convenient to separate people's activities in the onboarding phase from user actions taken at other stages. That way, you'll be able to distinguish between the actions typical for new users and those performed by users who have been with you for a while.

As for web analytics, apart from events, you'll also include every other analytics goal type you want to use to measure website performance, together with the properties and values attached.

5) Choose the right analytics platform

When your analytics tracking plan is ready, it's time to find a platform that will meet your expectations.

Of course, there are many factors you should consider, including:

- Compliance with data privacy regulations (like GDPR or HIPAA)
- Integration with other systems (like CRM, CDP or consent manager)
- The level of data ownership you'll have
- Accuracy of data, including sampling and access to raw data
- Hosting and data location options
- Customization possibilities and available features

You may be skeptical about the quantity and quality of the data you gather once you employ mechanisms that protect user privacy, such as consent forms. Some users may consent to the processing of their personal data, but many of them will ignore or reject your consent banner.

But it's still possible to draw insights without gathering personal data. Some platforms offer advanced anonymization options.

For instance, Piwik PRO Analytics Suite gives you a few options for utilizing anonymous data tracking:

- With cookies and session data

- Without cookies and with session data
- Without cookies or session data

One way to benefit from anonymous tracking is to display consent forms and enable data anonymization that will collect data about users that don't give their consent. This way, you collect personal data in a compliant way while also being able to access anonymous data. For example, you can have a user who initially ignores the consent form, performs actions and grants consent to personalization after 10 minutes. In this scenario, the initially anonymous data can be used for personalization every time the user returns to your website.

You may want to investigate other features when deciding on the most suitable analytics software based on the functionalities of your site or app.

For example, you might benefit from analyzing user behavior in post-login areas of websites or apps. Such data is valuable yet contains sensitive details, so finding a secure platform is fundamental.

Make your decision easier by checking out our comparisons of analytics platforms:

- Free comparison of 7 enterprise-ready customer data platforms
- Google Analytics alternatives – free and paid
- Compare 7 free web analytics platforms (product analytics included)

6) Implement the analytics platform

After choosing the right platform, it's time to implement it for your website or product. This step involves installing the tracking code that collects data about visitors to your site or app.

Piwik PRO'S Migration tool (GA3 and GTM)
The migration tool lets you quickly transfer your settings from Google Analytics 3 (Universal Analytics) and Google Tag Manager. It enables you to import GA3 properties, settings, goals, custom dimensions, and Google Tag Manager containers, including tags, triggers, and variables.

You should also add any modifications necessary – for example, set up tags to gather more detailed data or connect third-party tools.

On top of that, you can combine your selected platform with other tools to complement your analytics stack.

You could choose from a range of integrations to combine with your platform. For example, when you use Piwik PRO Analytics, you can integrate it with other modules and external tools like:

- Tag manager, allowing you to create and publish tags
- Consent manager, helping you collect, store and manage users' consents for different purposes of data processing
- Google products, like Google Ads, Google Search Console and Google Sheets
- SharePoint

Don't forget to create instances for all your sites you want to track, including subdomains or international versions.

7) Add events

Adding and tracking events lets you monitor what users do on your website. There are basically no limits to what you can track – clicks on specific buttons, how far people scroll on a page, and which fields users fill out in a form, to name a few.

Use meta setup in Tag Manager

Tag Manager's meta setups allow you to create setups for groups of sites or apps. It is also a way to apply it to all of them simultaneously. Variables, tags, and triggers are all part of it. You can use this tool if you want to run an advertising campaign across multiple websites or apps at once.

You need a meta site/app to use a meta setup. If you want to create a setup for a group of sites, you first need to create a meta site.

Let's look at use cases for meta setup in Tag Manager:

- Meta setup allows displaying a popup (or another form of promotion) across multiple sites based on one meta site (with the ability to customize it per site via variables).
- You can easily install marketing tools for all sites that are a part of the meta site.
- You can quickly create one shareable setup for multiple SharePoint sites (in many cases, most of the setup is identical).

By using a meta setup for your meta sites, you can simplify tag management. Tag management can be streamlined by importing and using configurations across multiple sites.

8) Set up goals and funnels

At this stage, you need to design and structure the goals and funnels outlined in the tracking plan.

Setting up goals will help you track the most important events and measure conversions.

Funnels should consist of the individual actions people take on your site or app to complete a given journey, for example, download an ebook or view a specific page with your offer. They let you examine how users navigate through pages and at which point they drop off. Using funnel reports will help you determine how to improve pages in the user journeys, leading to more of your goals being met.

9) Configure the UI of your analytics instance

In this step, you can enrich your analytics interface with supplementary information about user behavior.

Dashboards consist of widgets that visualize data so you can quickly glance at key aspects of your site's performance and share them with appropriate people. When making a new dashboard, work out the specifics first – who is going to view it, what is the context of the presented data and what insights can be drawn from it.

Only depict relevant metrics and make sure the dashboard is easy to read and visually coherent. For example, you might want to have a dashboard dedicated to marketing metrics, displaying the performance of various marketing campaigns, channels and other related initiatives.

Find out how to best present data in our guide on How to use data visualization in web analytics.

Custom reports give you an opportunity to customize the data by using your selected metrics and dimensions. They give you flexibility in displaying specifically the information you need.

At this stage, it's also good to manage other settings for your analytics platform, deciding what data you want to collect and how it appears in reports.

For example, in Piwik PRO Analytics Suite, you can select the conditions for collecting users' consent, decide if you want to remove URL parameters and see page URLs with anchors in reports. You can also specify IPs that should be excluded from tracking – like internal IPs, so you don't track your employees' behaviors and actions – and turn off collecting data from crawlers.

10) Create different segments

Next, you can create segments to filter relevant information and break down the data into smaller chunks. You can group data in a segment to see groups of people that match specified conditions, for example, users from a specific country or returning visitors.

From our experience, it's essential to create segments for the following data:

- Mobile users
- Desktop users
- Countries
- Paid campaigns

Lastly, be sure to maintain a test segment to monitor the results of your new ideas without losing existing data.

That way, you'll have all the relevant data at your fingertips.

Think about what other segments will help you clearly see the most important data that determines how well your website or product goals are being met. For example, if you aim to get users to create their accounts, consider monitoring how many users reached a certain stage of signing up. You could solve this by setting up a few segments: for people who visited different pages with your offer, started the signup process and finished it.

11) Grant proper permissions to people in your organization

In this step, you assign permissions to your stakeholders so they can view specific reports and dashboards.

The choice of who has access to particular information should be dictated by several factors, including security considerations and what data is needed in the day-to-day work of a given employee. For instance, marketers will likely need a completely different data set than people in the UX department or customer success team.

Pay attention to who is able to make important decisions, such as giving permissions to others or editing and publishing changes.

It may be a better call to apply restrictive rules when deciding on employees' permissions. Many employees might get all the information they need by just viewing the data.

You can schedule reports to be sent to any stakeholder without needing to log in to the analytics system.

12) Constantly monitor changes to your website/product structure, goals and KPIs

Your analytics strategy should never be static. Instead, it's a constantly evolving process that changes with every new insight, business goal, website or product feature, or new idea produced by experiments and tests.

It is worth reviewing your strategy every once in a while, refreshing goals, metrics, or reports if necessary. This way, you will always be sure that your analytics system delivers only fresh and valuable data relevant to your business objectives.

Getting started with analytics – what's next?

We hope the above steps will make the analytics implementation process much easier for you.

Remember to download our tracking plan to keep the data on your events, goals and funnels organized and easy to follow!

We can also recommend further reading, so you can expand your analytics skills:

- Customer journey analytics 101: what it is and why it's important for your business
- Mobile analytics: A complete guide to optimizing the user journey inside your app
- 10 things to consider for improving user experience with web analytics

Even with all the tips and tricks presented above, you might still have some unanswered questions. Our team will be happy to resolve any of your doubts:

What can GA4 event tracking be used for?

Before we go into detail about how to set everything up. Let's look at some of the different ways that event tracking can be used to measure visitor engagement on your site. Some of the typical uses for event tracking are:

- Tracking outbound link clicks to other websites.
- Understanding how many users clicked on mailto email addresses or click-to-call phone numbers – this can help you better understand the number of enquiries you're getting through your site.
- Tracking PDF and other media downloads.
- Measuring interactions with video content, such as time spent watching a video.
- Tracking exactly where users drop off when filling in fields on your forms or checkout.
- Monitoring the clicks on unique elements of a page, such as the "contact us" call to action on your about page.
- Collecting data about how many users filled in and submitted a form, although we'd always recommend sending users to thank you pages whenever possible.

Event tracking allows you to count interactions that don't necessarily involve loading another page on your website. GA4 conversions can also be set up based on your events.

Search... Rows per page:

	Event name	Event count	Total users	Event count per user
		182,424 100% of total	23,451 100% of total	7.81 Avg 0%
1	page_view	45,108	17,534	2.57
2	user_engagement	40,183	17,728	2.28
3	blog_post_viewed	30,982	18,816	1.65
4	session_start	23,911	19,005	1.27
5	first_visit	17,030	16,907	1.01
6	audience_used_organic	14,990	14,917	1.02
7	scroll	6,175	3,871	1.61
8	click	1,119	820	1.37
9	contact_page_view	882	341	2.59

What to consider before you start

To use event tracking you'll need to have GA4 installed on your website. You can do this by adding the code to all pages on your site or by adding Google Tag Manager code to your site and then configuring GA4 tags and triggers and variables.

Before you jump into setting up event tracking, it's important to consider the following points:

- Decide on **which elements of your site you want to track**, whether it's PDF downloads or clicks on outbound links.
- **Adopt a consistent and clear naming convention** for the different event names you're going to use when you're setting up event tracking. Every name you give to an event is the only means you'll have to understand what it does. If your naming isn't sensible then your report won't make a lot of sense later on.
- **Decide whether you want to set up auto-event tagging or manually tag links** on your site. If you have a lot of documents and page elements to track it may be worth setting up Auto Event Tagging and using Google Tag Manager Events.

How does Google Analytics event tracking work?

A snippet of custom code is added to the link code on the items you want to track on your website and when the item is clicked, the element is tracked and displayed as an event in Google Analytics. The event tracking code consists of four elements that you can define to describe a user's interaction on your website:

- **Event Name** (Required) is the name you give to the event, it's the main thing you'll see in your reports within GA4.
- **Event Parameters** (Optional) are the extra bits of data you can put into your events, e.g. the page it occurred on, or how far down the page they scrolled

The event tracking code for an event tracked link in GA4 looks like this:

```
gtag('event', '<event_name>', {
  <event_parameters>
});
```

The code is placed after the href link text as illustrated in the example below:

<a href=”www.examplewebsite.co.uk/company_brochure.pdf” *onclick=”gtag(‘event’, ‘<event_name>’, {<event_parameters>});”>*

The event name and any parameter labels are replaced by the values that you decide to enter. An example of an event-tracked link with entered values is further down the page.

How to set up GA4 event tracking

Depending on the number of events you’d like to track, or the level of control you’d like to have on the tracking parameters for your events, you can setup up auto event tracking, or manually tag links on your website.

If you have lots of documents and page elements to track it’s worth using auto event tagging, which you can do using Google Tag Manager. Auto event tagging will fire on the following:

- When users click on links.
- Clicks on any type of page element.
- After a certain visit duration or at timed intervals.
- On submission of a form.

If there are other actions that you want to track then you can use Google Tag Manager to set this up.

Manually tagging links to track events

You can manually customise links on your site to add the event attributes to a link as we discussed earlier. To give you a practical example of this, below is a link that includes event code for tracking a PDF download.

The italicised text shows an example of event tracking parameters configured to record the download of a company brochure PDF document.

Download Our Brochure

Auto-event tracking options

Event tracking using Google Tag Manager

You can measure interactions on your site by setting up [Google Tag Manager’s Auto-Event Tracking](#).

This might sound complicated but once you understand the basic principles of tags, triggers and variables it becomes quite straight forward.

Event tracking using Google Tag Manager is initiated via clicks on event triggers that can be setup on specific variables on your webpage.

Google Tag Manager events are user actions with web page elements ("DOM elements") that are triggered by your browser and sent into the Tag Manager data layer so they can be used to set up triggers.

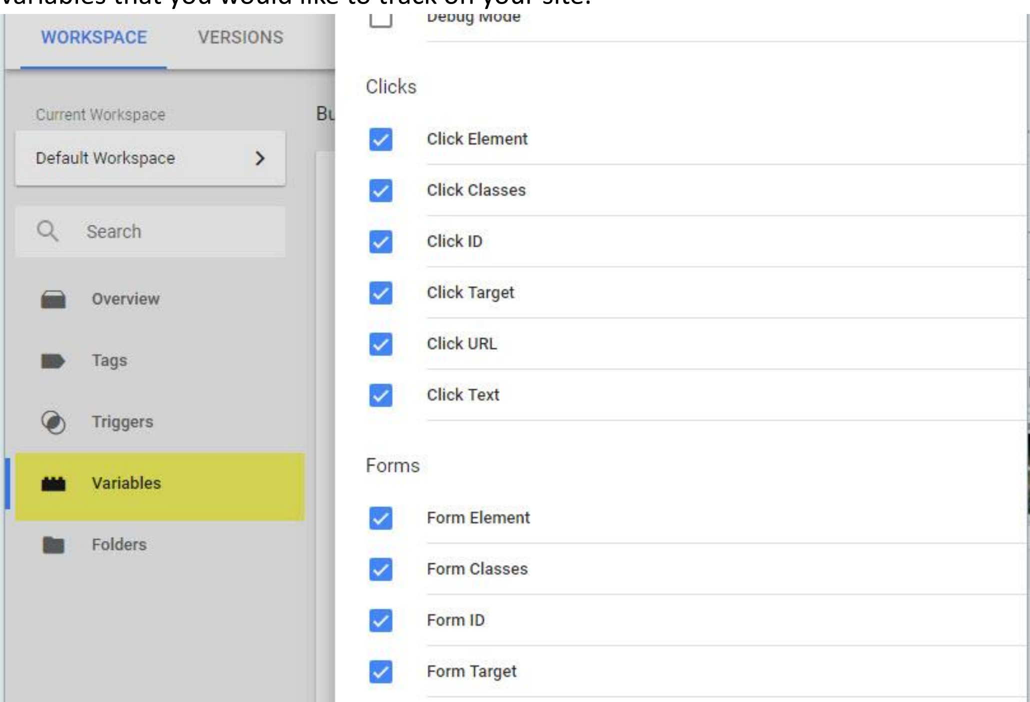
The steps for setting up an event in Google Tag Manager are as follows:

1. Log into Google Tag Manager
2. Select "Tags" from the left-hand side
3. Create a new tag and select Universal Analytics as the Tag Type
4. Set your GA4 Tracking ID
5. Choose "Event" for the track type
6. Set your Event Category, Action, Label and Values. You can use Google Tag Manager variable names such as {{click url}}
7. Set your triggers as required

An example of how to set up an event within Google Tag Manager is outlined below:

Step 1 – check you have the right enabled variables selected for your event

Head over to the Variables section in Google Tag Manager and make sure that you have ticked the variables that you would like to track on your site.



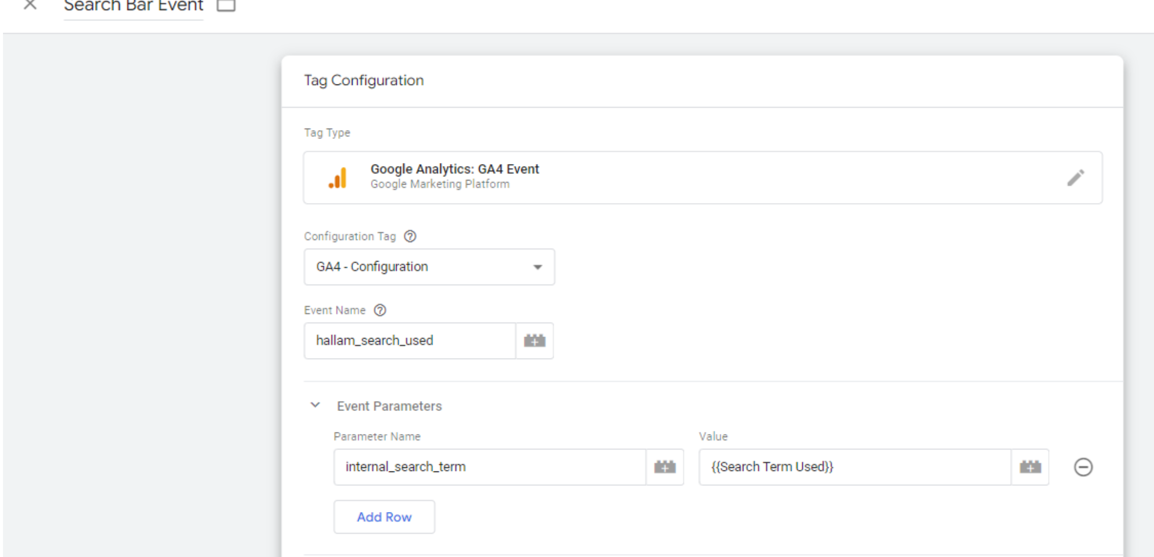
You have the option to set up your triggers based on click classes, click elements, click text and other variables.

Step 2 – create a new tag in Google Tag Manager

Create a new tag in Google Tag Manager making sure to choose the "GA4 Event" tag and not the Configuration tag. The configuration options will now include tracking parameter fields for Event Name and any Parameters you want to add as shown in the screenshot below.

As we discussed earlier, the Event Names form the basis of the tracking within GA4. The event parameters are only there if you want to do further narrowing down within GA4 or through Big Query. There are Google Tag Manager specific code variables that you can use to automatically populate fields with values. For example, the {{Click}} code will automatically pull in the web URL into the field it's entered into.

Here, we've built a GTM variable to populate the search term used in our search bar.



Step 3 – configure the tag

Now you want to enter values for the Event Name and any parameters.

The traditional naming convention is to use all lower-case and _'s instead of spaces. However, if you want to put capitals or spaces into your events they will work, it just often ends up looking untidy. Outside of that convention, the real key is making it something that's identifiable at a glance. It should be immediately obvious what this event is and how it fires as you're going to be coming back to it in 6 months time and don't want to have to play archaeologist.

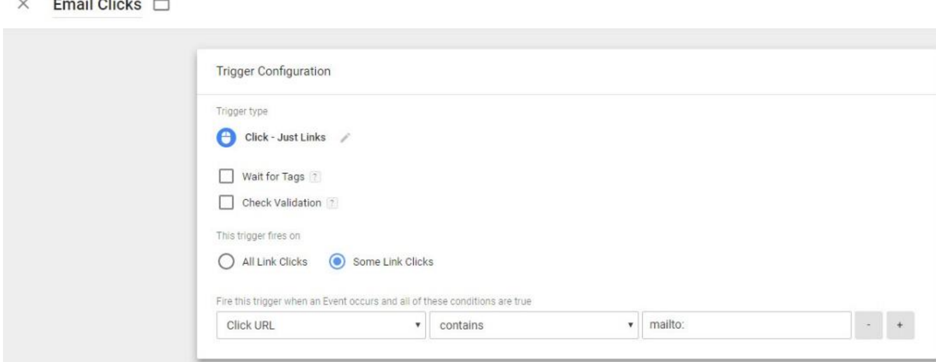
The Parameters box for your event can be populated if you want to attribute any other data to your event. You may want to specify which form was submitted or if a user was logged in or not. All the basic Parameters from a page_view event will populate automatically (e.g. page_location) so don't feel like you need to duplicate that work.

Step 4 – select what the event tag will fire on

The next step is to select or create a trigger for the tag to fire on. You'll need to set the fire on conditions for your tag. Below are a few examples of different types of triggers for different events on your site.

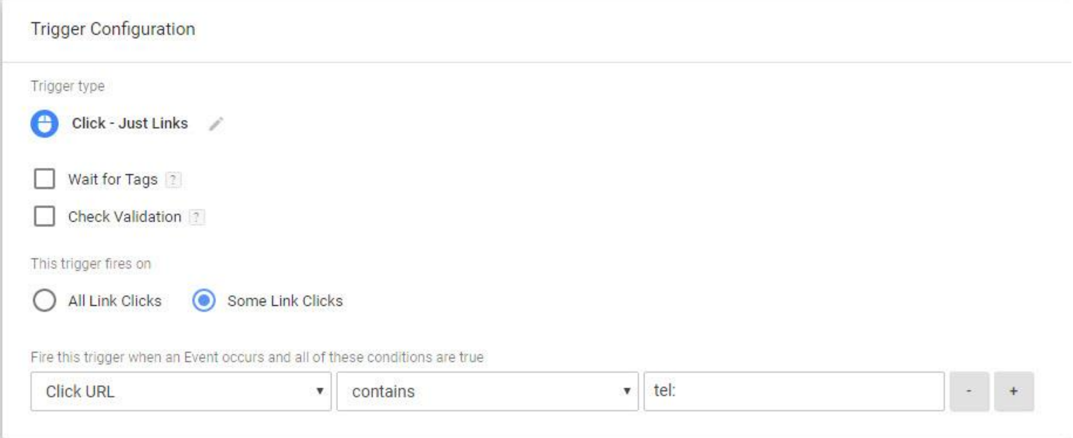
Trigger for measuring clicks on an email address link

An example of a completed trigger for an email address clicked link is below.



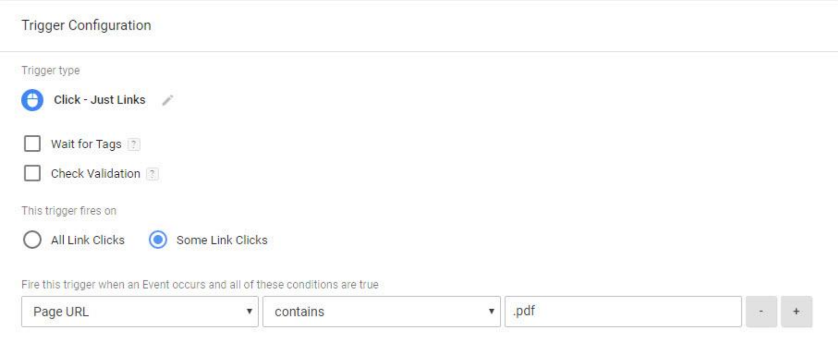
1. On the choose Trigger Type screen under the Click heading choose Just Links
2. Select the Some Link Clicks under this trigger fires on
3. Set the variable to Click URL within the first drop-down box
4. Specify that the URL ‘Contains’ in the second drop-down option
5. Enter mailto: within the third field

Trigger for measuring clicks on a phone number



1. On the choose Trigger Type screen under the Click heading choose Just Links
2. Select the Some Link Clicks under this trigger fires on
3. Set the variable to Click URL within the first drop-down box
4. Specify that the URL ‘Contains’ in the second drop-down option
5. Enter tel: within the third field

Trigger for measuring PDF downloads



Using GTM to set up a PDF download event

1. On the choose Trigger Type screen under the Click heading choose Just Links
2. Select the Some Link Clicks under this trigger fires on
3. Set the variable to Click URL within the first drop-down box
4. Specify that the URL 'Contains' in the second drop-down option
5. Enter .pdf within the third field

Testing that your events work as planned

It's a good idea to check that your events work as you intend. To view results immediately you can preview your Google Tag Manager events using the preview feature and carry out your event action to see whether the tag fires successfully. You should then follow up by going to "Debug View" within GA4 to check that it's being received by Google Analytics as well.
Within GA4 you can also see events fired within the last 30 minutes by going to the Real-Time section and then Events section.

If you're not in a rush or want to view historic data, go to the Behaviour > Events section of GA4.

Setting up event-based goals

Conversions can also be set up in GA4 based on your events. To do this login to your GA4 account and follow these instructions:

1. Go to the property where you want to set up the goal
2. Click on Admin and then go to Events
3. Find your Event in the list (Note: If you've only just set up your event, you'll need to give it 24 hours for the data to make it through to GA4.
4. Toggle the slider that says to mark it as a conversion.
5. That's it! It's one of the few things that's actually easier post-GA4.

What is user behavior?

User behavior describes all the actions a user performs on a website or mobile app. It includes factors like time on page, the number of pages visited, how people interact with different features, and also friction they encountered while using the product, etc.

What is user behavior analytics?

User behavior analytics (UBA) is a technique for recording user activity and then combining and analyzing that data to understand how and why users interact with a product or website.

In essence, analyzing user behavior helps you understand users the same way you would in an offline setting. For instance, if you run a brick-and-mortar store, you can set up cameras to record how customers enter the mall and interact with your products.

Your on-site staffers will also notice some of these customer behaviors by just observing.

The information collected over time will allow you to decide on improvements you must make to better customer experience and increase sales. This is something user analytics helps you to replicate for your SaaS product and website visitors.

A traditional tracking tool like Google Analytics helps you know what's going on with your website's visitors, but it isn't built to explain why site visitors behave the way they do.

User behavior analytics is different because it provides both quantitative and qualitative data—the what and the why—to facilitate better decision-making.

As we proceed in this article, you will see the different kinds of user behavior tracking, and tools to enable you to execute a proper user behavior analysis. But before we move on, it's worth noting that there's a difference between behavior analytics and behavioral analytics.

Both terms sound similar, but they are not.

Behavioral analytics is more concerned with predicting user behavior, while behavior analytics focuses more on optimizing the user experience.

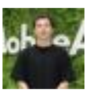
What are user behavior metrics?

User behavior metrics are analytics data that tell you how visitors engage with your website or app.

Key metrics to look out for as you track user behavior include:

- Conversion (turning website visitors into trial users, then into paid users etc).
- Leading trial users to the "aha" moment and the activation point—the point they instantly see how your tool makes life easier for them.
- User activation.
- Complete adoption.

Mobile App Monetization: Top 13 Ways to Maximize Your Revenue



Yarkın Tepe

August 1, 2023

MobileAction BLOG

Mobile App Monetization:
Top 13 Ways to Maximize Your Revenue

Today, mobile apps make the world go around. We do everything via mobile apps, calling a taxi, going on a date, ordering food, playing games, learning languages, catching up with the most viral videos, and many more. On top of that, according to statistics, 95% of the apps are free of charge, which means we can download approximately 95 apps out of 100 for free.

But it is also worth remembering: every app is a business, and they need to make money to survive. So how do these apps with free access generate revenue? How can mobile app developers make money and invest in the further development of their apps?

The answer is mobile app monetization.

Whether you have a free app or a paid app, regardless of your expertise, we have collected 13 ways to start or optimize your mobile app monetization processes. At the end of this article, you are going

to discover what mobile monetization is, why it is so important, what the best practices are to follow in 2023, how to choose the right pricing model by finding a balance between value and revenue, and some frequently asked questions about mobile app monetization.

Just a quick reminder: staying strong in the competition is at least as important as having a well-designed monetization model. For this purpose, Ad Intelligence by MobileAction was developed to help your app stay a few steps ahead of the competition.

Using Ad Intelligence helps you analyze various ad networks and provide insights into the creative sets of your competitors. In this way, Ad Intelligence is helpful in the long run as a good mobile app monetization model. By signing up for free, you can discover more about our game-changing solution!

So get ready and fasten your seatbelt to jump into the world of mobile app monetization.

What is Mobile App Monetization?

Mobile app monetization is the process of following a set of strategies to generate the necessary revenue to support your business. It is basically the concept of converting your app's users into revenue without applying any download fee. So, although it is a way of generating revenue, charging some download fees for your mobile app is not part of the mobile app monetization process.

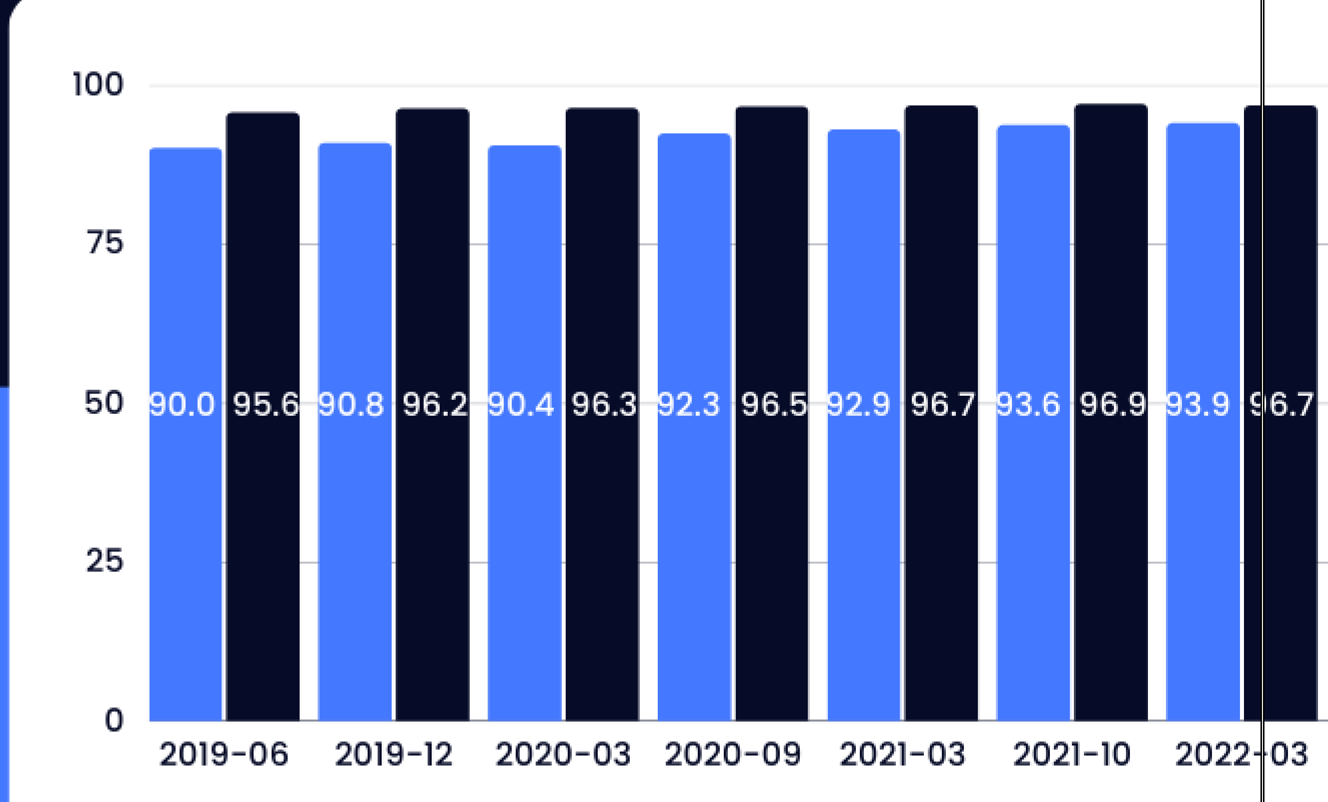
Due to the severe competition among them, free mobile apps are becoming more prevalent across the entire digital environment. Considering that there are more than 5 million apps available in the two biggest application stores, it is not difficult to predict that this competition will continue. As a result, users are becoming more and more reluctant to pay for any application.

As a result, app developers must take into account a variety of app monetization strategies. There are a number of monetization methods that enable mobile apps to convert their users' actions into revenue. However, since there is no such thing as the best way, it is necessary to analyze which method is suitable for which app.

So let's first continue with the prominence of mobile app monetization briefly, then dive deep into the best mobile app monetization strategies available in 2023.

Why is Mobile App Monetization Important?

The trend of free apps spreading continues to grow with a snowball effect. This means that the pay-per-download method is now an old-fashioned monetization strategy to generate revenue. For this reason, mobile app developers have to find alternative routes.



App developers can utilize one of the following app monetization models to make sure an app's revenue keeps increasing while maintaining an excellent user experience. As an alternative, they can use a hybrid app monetization strategy that combines two or more ways to make money.

After choosing the right strategy, the first thing to do should be to move the app at least one step ahead of the competition. You can use the Creative Analysis feature of the Ad Intelligence tool to get ahead of your opponents' creative sets. With Creative Analysis, you can filter time intervals, ad networks, and countries to get an overview of your competitors' paid user acquisition efforts; see

your competitors' creatives and analyze impressions and clicks to understand what works for them; and get to the best-performing creatives faster by analyzing your competitors' current and past creative tests.

If everything is crystal clear so far, we can start reviewing the best 13 mobile app monetization strategies in 2023.

Top 13 Ways to Monetize Your App in 2023

1. In-app Advertising (IAA)

This method offers a valuable source of revenue for apps that want to remain free in the app store. The most common way to monetize mobile apps is by running in-app ads. Providing content and ad formats relevant to your users can create a great user experience. Bu sebeple, mobile app userlarinin dikkatlerini farkli sekillerde cekebilmek ve onlara en iyi deneyimi sunabilmek icin birden fazla in-app advertising metodu bulunuyor.

There are five main types of IAAs:

Banners: Advertisers can post static or animated ads across a banner within your app using this simplistic and easy ad style. With banner ads, having eye-catching images and a compelling call to action are essential.

Interstitial Ads: This style presents advertisements in full-screen. Because of this, interstitial ads are more likely to prevent creative ad fatigue, a condition in which viewers are so accustomed to seeing banner ads that they fail to notice them. It's crucial that these advertisements are inserted at natural pauses, such as the transitions between levels of a gaming game, as full-screen interstitials can be annoying.

Native Ads: Native ads work best on news websites and social media because they are made to blend in smoothly with their host app, appearing as another post in the feed. When used properly, these advertisements cause only little disturbance to the user experience.

Video Ads: Another common option is video advertising, which has tremendous potential for engagement and some of the highest CTRs of any media. Additionally, by rewarding viewers for watching an entire video, rewarded video ads allow you to increase the attraction of video ads.

Playables: These advertisements expose viewers to interactive gameplay as a try-before-you-buy strategy. Before receiving a CTA, users see a brief preview of the advertiser's app. Because customers who choose to install will already be interested in the gameplay, this is an excellent approach for advertisers to lower uninstall rates.

2. In-App Purchases (IAP)

IAPs (in-app purchases) provide a unique revenue strategy that may even improve the user experience. Think about how you can improve the user experience with IAPs.

There are two types of IAPs:

Consumables: This type of IAP is for temporary or limited purchases. Extra lives in Candy Crush Saga, keys in Subway Surfers, and hearts in Score Hero can be good examples of consumables. What they have in common is that all of them give you extra chances to continue playing the game.

Non-consumables: This type, on the other hand, only needs to be purchased just for once. For instance, unlocking a new character in Subway Surfers and removing in-app ads in New Star Soccer are non-consumables.

While the ability to make an in-app purchase can be added at any time while playing the game, the wisest approach to doing so is to strategically place in-app purchase options where customers are most likely to need them.

3. Subscriptions

An additional popular way of monetizing your mobile app is through subscription services. You can give paying users of your app a different experience than those who just use the free version thanks to subscriptions.

You aren't forced to take an all-or-nothing stance when setting up a subscription bundle for your mobile app. It might be wiser to build up multiple subscription packages with differing levels depending on what your app has to offer. For various user categories, you could want to provide a basic subscription, a pro subscription, or a premium subscription. Alternatively, maybe you can name it differently, such as Silver, Gold, and Platinum.

This way of monetizing mobile apps is very popular among dating apps such as Tinder and Bumble. In both of these apps, after you pay a monthly subscription fee, you can get unlimited likes, rewind your last swipe, and match people who are not in your range.

4. Freemium

Today, because of their countless advantages, freemiums are one of the most popular mobile app monetization strategies. Freemiums offer the basic features of a mobile app for free to everyone while offering exclusive access to users who only make in-app purchases. These in-app purchases enable them to use some premium features and/or access additional content.

Two of the most popular examples of freemium services are Spotify and YouTube. Both of them offer their basic services for free, but Spotify Premium and YouTube Premium users can access these services without having to watch or listen to ads.

Freemium subscriptions are advantageous for two reasons:

First, they enable users to try out your app before they need to pay some amount of money for an app they are not familiar with. In this way, the app becomes more persuasive. Second, people who prefer to use the free version of the app and never consider upgrading to the premium version help the app reach a wider audience by just downloading it.

5. Affiliate Marketing

By promoting other apps, goods, and services through your app and putting affiliate tracking in place, you can monetize your app by using affiliate ads.

People enjoy recommending things to each other, which is why affiliate ads are effective. This strategy can be quite helpful in converting if the audience has faith in the source.

Throughout your app, these affiliate schemes will advertise other mobile applications. The fact that app developers can choose which apps to advertise, even though the commissions paid in this app monetization approach are typically not high, can attract consumers to associate their app with similar companies.

Therefore, you should link the advertisement to display at relevant points in the user journey. The advertisement can recommend an app that is related to the user's preferences.

6. SMS Marketing

Although SMS marketing is an old-school mobile app monetization method, it can still be effective. When the language in the SMS sent is retouched with today's slang, humor, and speaking style, you can avoid the SMS being perceived as spam.

We can say that mobile applications can easily access our phone numbers since almost all of the apps we download require a membership, and our phone number is usually requested in this membership form. This makes the implementation of SMS marketing very easy.

However, there is a point to be noted here because knowing a user's phone number does not mean that mobile apps can easily send messages to those apps. Mobile apps must obtain consent to send SMS to their members during the membership process.

Since we have seen almost half of the 13 ways, it is useful to mention the Advertiser Analysis feature of Ad Intelligence here. Advertiser Analysis lets you see how your competitors split up their ad creatives throughout various ad networks in various storefronts and time frames. Find the most profitable ad networks to increase your ROAS.

Find out which storefronts have the most app downloads in your genre and which languages your competitors are using in their marketing. With our ad analytics, you can discover which media types are most effective for your particular app category and outperform your rivals.

7. Email Marketing

Although it has many similarities with SMS marketing, another old-school marketing method, email marketing also has many unique advantages and disadvantages.

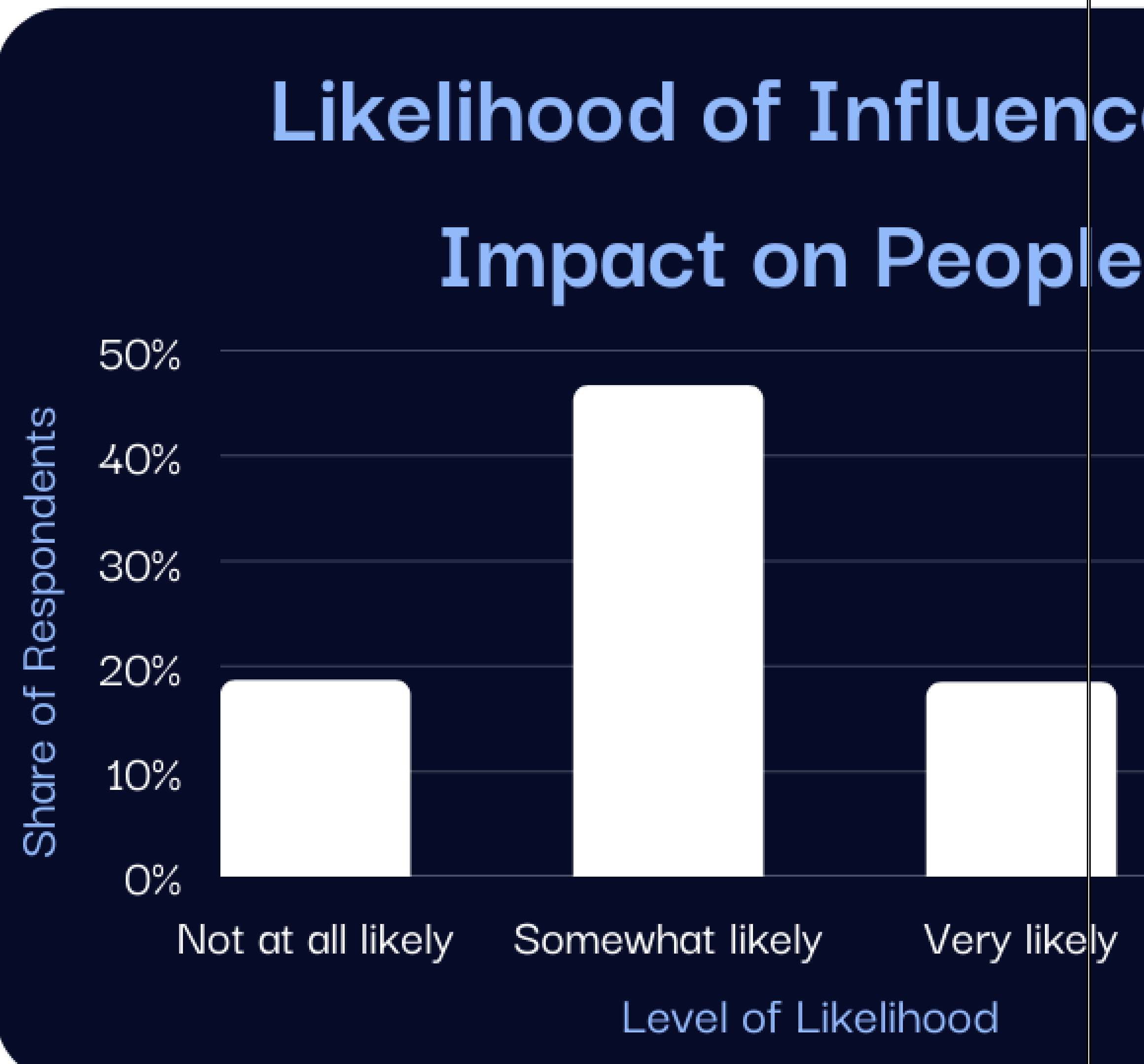
As in SMS marketing, mobile applications can easily access our email addresses, but again, mobile apps must obtain consent to send emails to their members. Luckily, an advantage of email marketing for an app developer is that it is easier and cheaper than any other mobile app monetization method.

Since this method will be weaker than many other mobile app monetization methods, it would be beneficial for mobile app developers to combine this method with another method and establish a hybrid model instead of choosing this method alone.

8. Influencer Marketing

People are more likely to believe recommendations from others than advertisements. Therefore, recommendations from influencers provide potential users of your app with social proof.

Mobile app influencer marketing helps you build a robust brand image, establish long-lasting relationships, gain access to your intended audience, establish authenticity, increase social media visibility, and attract a large number of new users.



However, there are two points that mobile app developers should consider. First, there should be a psychological contract between the influencer and the target audience, which means the users of the mobile app must love and trust the selected influencer. Secondly, since influencer marketing is a costly marketing method, especially for indie developers who are in the very early stages of the mobile app development industry, the potential financial burdens it may create should be carefully analyzed.

9. Sponsorships/Partnerships

If an app is very popular in a particular market, companies operating in that space may offer sponsorship to the app developer to show their brand to app users.

Therefore, during the ideation phase of your next app, you may also think about the target user base the app can appeal to and the brands that are likely to be of interest to this audience. Mobile app monetization through sponsorship, partnership, or app purchase can also be good option if the user base is large enough.

Sponsorships and partnerships can also create a cross-promotion opportunity in which you can promote other products and services to increase user acquisition and revenue. However, especially in the early stages of mobile app development, finding a sponsor is not very easy since most businesses prefer to partner with mobile apps that their target audience is familiar with.

10. Licensing

Licensing can also be a way of monetizing a mobile app for a developer. It basically means selling your app's codes and/or user data. If you really think you have a strong code that other mobile app developers can benefit from or you can collect valuable user insights from your app's audience, you can consider selling them to generate revenue.

For instance, if you have developed a well-built mobile game that has been approved by other developers and has successfully reached large audiences of players, other developers may approach you and offer to re-skin your game. Imagine licensing Flappy Bird as its developer 10 years ago. Undoubtedly, the possible revenue you could generate would be enough to develop several new mobile apps.

Unfortunately, especially for indie mobile app developers, licensing codes and data can be very expensive. In addition, there is no possibility of selling all the user app data and insights you collect because, in most countries, some data from users fall within the scope of data protection laws and regulations.

11. Crowdfunding

Crowdfunding is the practice of raising money from a large number of people on the Internet. Developers who prefer to fund their businesses with crowdfunding promote some or all features of their apps on some websites such as Kickstarter, Patreon, GoFundMe, etc.

This way of monetizing is very effective, especially if you are in the early stages of developing your app. With a well-designed, engaging, and persuasive crowdfunding campaign, any mobile app can raise the desired amount of money in a very short period of time.

Crowdfunding does not only give you a helping hand in terms of financial support but also creates a built-in community of users who can promote your app by word-of-mouth, provide valuable reviews and comments to help you optimize your app, and support you during your app development process. However, when there is no application in sight and it is very difficult for a new business to find a correct and strong brand voice, crowdfunding campaigns may not look convincing enough.

12. Donations

Although it looks similar to Crowdfunding at first glance, donations are a unique way of monetizing mobile apps. Even if most of their advantages and disadvantages are common with crowdfunding, donations can be leveraged for any mobile app at any stage.

In mobile apps monetized through donations, the app's users serve as "patrons" and raise some money. With their help, developers can offer the entire platform for free to everyone.

Suppose mobile app developers want to get the most out of this monetization strategy. In that case, they can offer suggestions about how much a user should donate in order to make a significant contribution. In this way, patrons will see that their money serves a good cause and that these donations are really useful in the development or optimization of the mobile application.

13. Offerwalls

Offewalls offer rewards to app users in exchange for taking some desired actions and completing certain tasks. These tasks can include completing surveys, playing games, signing up for some websites, downloading apps, or reaching a certain level or rank in a game.

As offerwalls are a mobile app monetization method that makes it easy to integrate with other apps and websites, they facilitate user migration between two businesses. For this reason, mobile app developers can employ offerwalls and affiliate marketing at the same time as a hybrid monetization model.

If you've read all these monetizing methods and learned how to get one step ahead in competition with Ad Intelligence's Creative Analysis and Advertiser Analysis features, you might be interested in the Top Charts in Ad Intelligence as a final step.

In Top Charts, you can see Top Advertisers, Top Ad Publishers, Top Creatives, and Top Developers, examine the millions of creatives in apps, discover who is dominating the ad ecosystem across different ad networks, and identify the most active app publishers to get a better understanding of the competition.

Key Takeaways

- Mobile app monetization involves converting users' actions into revenue without charging a download fee.
- Mobile app monetization is crucial for app developers to generate revenue and invest in app development.
- With over 5 million apps available in the two biggest app stores, competition is increasing, making users more reluctant to pay for apps. As a result, 95% of the available apps do not charge a download fee.
- To maximize revenue, you should consider 13 ways to optimize their app monetization strategies, whether they are free or paid. These methods are In-app Advertising (IAA), In-App Purchases (IAP), Subscriptions, Freemium, Affiliate Marketing, SMS Marketing, Email Marketing, Influencer Marketing, Sponsorships/Partnerships, Licensing, Crowdfunding, Donations, and Offerwalls.
- By analyzing which method is suitable for each app, you can pick one or multiple strategies to employ and ensure your mobile app's revenue continues to grow while maintaining an excellent user experience.
- If just having a well-designed mobile app monetization model doesn't satisfy you and you always aim for better and further, why don't you schedule a demo today and discover more about our ASO solutions?

Page **578** of **578**