

CCS332 App Development Unit - 2 native App

Native App

Native App: A native app is built specifically for a particular platform (e.g., Android, iOS, Windows) using the platform's native programming language (Java/Kotlin for Android, Swift/Objective-C for iOS). It's installed directly on a user's device and has full access to the device's capabilities

Web App

Web App: A web app, on the other hand, is accessed through a web browser and built using web technologies like HTML, CSS, and JavaScript. It is not installed on the device but rather runs within the browser, making it platform-independent.

Native Web App

A native web app refers to a web application that is built using web technologies (HTML, CSS, and JavaScript) but behaves and feels like a native app when used on a user's device. It is designed to run on various platforms and devices with a single codebase, rather than being developed separately for each platform. This is typically achieved by using frameworks like React Native, Flutter, or other technologies that enable crossplatform development. Native web apps are becoming increasingly popular due to their ability to offer a native-like experience while still being accessible through web browsers. **Benefits**

of Native App

Native apps offer several benefits, which contribute to their popularity and effectiveness in delivering a superior user experience. Some of the key advantages of native apps include:

1. **Performance:** Native apps are optimized to run directly on a specific platform and use the device's hardware and software to their fullest potential. This results in faster loading times, smoother animations, and overall better performance compared to web apps.
2. **User Experience:** Native apps can provide a seamless and intuitive user experience, as they are designed with the platform's guidelines and user interface components in mind. This leads to a more familiar and user-friendly interface for the app's target audience.
3. **Access to Device Features:** Native apps have full access to the device's features and capabilities, such as camera, GPS, accelerometer, and more. This enables developers to create feature-rich and interactive applications.
4. **Offline Functionality:** Native apps can often work offline or in areas with poor internet connectivity by storing data locally. This feature is particularly useful for certain types of apps like productivity tools or games.

5. **App Store Distribution:** Native apps can be distributed through app stores like Google Play Store and Apple App Store. This makes them easily discoverable to a broad user base and allows for seamless updates and app management.
6. **Security:** Native apps have the advantage of following the security guidelines of the platform they are built for, reducing the risk of security vulnerabilities.
7. **Better Integration:** Native apps can be integrated more effectively with other apps on the device, allowing for a more cohesive user experience when switching between applications.
8. **Performance Analytics:** App stores and developer tools provide detailed analytics on how users interact with native apps. This data can help developers make data-driven decisions to improve app performance and user engagement.

Drawbacks of Native App:

1. Higher development costs (especially if targeting multiple platforms),
2. Longer development timelines

Scenarios to create Native App

Creating a native app can be a suitable choice for various scenarios, depending on the specific requirements and goals of the project. Here are some scenarios where developing a native app is often preferred:

1. **Performance-Critical Applications:** When the app requires high performance, responsiveness, and smooth animations, a native app can fully leverage the device's hardware and software capabilities to deliver the best user experience. Examples include graphics-intensive games, real-time communication apps, and video editing applications.
2. **Utilizing Device Features:** If the app needs to access and utilize device-specific features extensively, such as GPS for location-based services, camera for photo/video processing, or Bluetooth for IoT applications, a native app is the way to go.
3. **Offline Functionality:** For apps that need to function offline or in areas with limited connectivity, native apps can store data locally and allow users to interact with the app even when not connected to the internet.
4. **Platform-Specific User Experience:** If you want to provide a seamless and platform-specific user experience to cater to the preferences and habits of users on different platforms (iOS and Android), native app development is the ideal choice.

5. **App Store Distribution:** If your app's success depends on widespread distribution and easy discoverability, publishing it on app stores like Google Play Store and Apple App Store as a native app can greatly improve its visibility and accessibility.
6. **Security and Data Protection:** Native apps can implement security measures more effectively, ensuring the protection of sensitive user data and complying with platform-specific security guidelines.
7. **Complex Business Applications:** Enterprise-grade applications often require integration with complex backend systems, and native apps can offer robust APIs and libraries to facilitate this integration.
8. **High Interactivity and Rich User Interface:** If your app demands a high level of interactivity and a rich user interface with custom animations and transitions, native development provides greater control over these aspects.

It's important to note that while native apps offer many benefits, they come with higher development costs and longer timelines, especially if you plan to support multiple platforms. In some cases, cross-platform frameworks like React Native or Flutter may also be considered as alternatives, striking a balance between native performance and development efficiency. The choice of development approach should be based on a careful evaluation of the project's specific needs and resources.

Tools for creating Native App

There are several tools and frameworks available to create native apps for various platforms. These tools aim to streamline the development process and enable developers to build apps using familiar web technologies. Here are some popular tools for creating native apps: 1. **Android Studio (Java/Kotlin):** Android Studio is the official Integrated Development Environment (IDE) for Android app development. It supports Java and Kotlin as programming languages and offers a rich set of tools to build native Android apps.

2. **Xcode (Swift/Objective-C):** Xcode is the official IDE for iOS and macOS app development. It supports Swift and Objective-C as programming languages and provides tools to create native apps for iPhones, iPads, and Macs.
3. **React Native:** Developed by Facebook, React Native is a popular cross-platform framework that allows developers to build native apps using JavaScript and React. It enables code reuse across iOS and Android platforms while offering near-native performance.

4. Flutter: Developed by Google, Flutter is another cross-platform framework that enables developers to build native apps using Dart programming language. It offers a fast and expressive way to create beautiful UIs and has a rich set of widgets.
5. Xamarin: Owned by Microsoft, Xamarin is a cross-platform framework that allows developers to build native apps for Android, iOS, and Windows using C# and .NET. It provides access to platform-specific APIs and features.
6. Ionic: Ionic is a popular open-source framework for building hybrid apps using web technologies like HTML, CSS, and JavaScript. While not fully native, it uses WebView to run inside a native container, allowing for cross-platform compatibility.
7. NativeScript: NativeScript is an open-source framework that allows developers to build native apps using JavaScript, TypeScript, or Angular. It provides direct access to native APIs and delivers native performance.
8. Appcelerator Titanium: Appcelerator Titanium is a platform that enables developers to build native apps using JavaScript. It provides a single codebase for multiple platforms, including Android and iOS.

Each of these tools comes with its own set of features, advantages, and limitations. The choice of the tool depends on factors such as the development team's expertise, project requirements, desired platform support, and development efficiency. Before selecting a tool, it's essential to evaluate its capabilities and suitability for the specific native app project.

Cons of Native App

While native apps offer many advantages, they also come with certain drawbacks and challenges. Here are some of the cons of native app development:

1. Development Cost and Time: Native apps require separate development for each platform (iOS and Android), which can increase development costs and timelines significantly. Maintaining two codebases can also lead to higher ongoing development and maintenance expenses.
2. Platform Dependence: Native apps are specific to a particular platform (iOS or Android). This means that separate codebases are needed to target different platforms, which may limit the app's reach to a broader audience.
3. App Store Approval Process: Publishing native apps on app stores like Google Play Store and Apple App Store requires adherence to strict guidelines and approval processes, which can introduce delays and rejections.

4. **Skill Set Requirements:** Developing native apps requires developers with platform-specific expertise. For example, iOS apps may require knowledge of Swift or ObjectiveC, while Android apps may need expertise in Java or Kotlin. Finding developers with diverse skill sets can be challenging.
5. **Updates and Maintenance:** Native apps need to be regularly updated and maintained to stay compatible with the latest OS versions and devices. This continuous effort may increase operational costs.
6. **Distribution Limitations:** Native apps can only be distributed through official app stores or sideloaded through specific procedures. This can be a disadvantage if the app needs to be distributed outside app stores or to a limited audience.
7. **Fragmentation:** Both iOS and Android platforms have multiple device models and OS versions in use, leading to device fragmentation. Ensuring consistent performance across all devices can be challenging.
8. **User Adoption Barrier:** Users need to download and install native apps on their devices, which may act as a barrier to adoption compared to web apps that can be accessed directly through a browser.
9. **Cross-Platform Challenges:** If the goal is to support multiple platforms, developers may face challenges in achieving full consistency and leveraging all native capabilities across different operating systems.

To mitigate some of these drawbacks, businesses and developers often consider alternative approaches, such as cross-platform development frameworks (e.g., React Native, Flutter, Xamarin) or hybrid app development, which combine web technologies with native components. The choice between native, cross-platform, or hybrid development depends on the specific project requirements, budget, and long-term goals.

Popular Native App Development Frameworks

Some of the popular native app development frameworks include:

1. **React Native:** Developed by Facebook, React Native is a widely used crossplatform framework that allows developers to build native apps using JavaScript and React. It offers a vast community and extensive third-party libraries, enabling code sharing between iOS and Android platforms.
2. **Flutter:** Developed by Google, Flutter is another popular cross-platform framework that uses the Dart programming language. It allows developers to create native apps with a

single codebase for iOS, Android, and other platforms, offering a rich set of widgets and high-performance rendering.

3. Xamarin: Owned by Microsoft, Xamarin is a cross-platform framework that enables developers to build native apps for Android, iOS, and Windows using C# and .NET. It provides access to platform-specific APIs and features and allows code sharing among different platforms.

4. NativeScript: NativeScript is an open-source framework that enables developers to build native apps using JavaScript, TypeScript, or Angular. It provides direct access to native APIs, enabling the creation of high-performance apps.

5. Kotlin Multiplatform Mobile (KMM): Kotlin Multiplatform Mobile, introduced by JetBrains, allows developers to share business logic and code across Android and iOS platforms using the Kotlin programming language. It is designed to foster better interoperability and code sharing.

6. Ionic: Ionic is a popular open-source framework for building hybrid apps using web technologies like HTML, CSS, and JavaScript. While not fully native, it uses WebView to run inside a native container, allowing for cross-platform compatibility.

7. Appcelerator Titanium: Appcelerator Titanium is a platform that enables developers to build native apps using JavaScript. It provides a single codebase for multiple platforms, including Android and iOS.

8. PhoneGap/Cordova: PhoneGap (also known as Cordova) is an open-source framework that allows developers to build hybrid mobile apps using web technologies. It wraps web app code within a native container for distribution on various platforms.

9. Swift UI: For iOS app development, SwiftUI is Apple's declarative framework that allows developers to build user interfaces across all Apple platforms using Swift.

Java & Kotlin for Android

Java and Kotlin are popular programming languages used for Android app development.

1. Java: Java has been the traditional and official programming language for Android app development for many years. It is widely used and supported by the Android platform. Many existing Android apps are written in Java.
2. Kotlin: Kotlin, introduced by JetBrains, is a modern and officially supported programming language for Android development. It is fully interoperable with Java, which means

developers can use Kotlin alongside Java in the same project without any issues. Kotlin offers concise syntax, null safety, and improved readability, making it increasingly popular among Android developers.

Google announced Kotlin as an official language for Android development at Google I/O 2017. Since then, Kotlin has gained significant adoption in the Android development community due to its many advantages over Java. Many developers have transitioned to using Kotlin for new projects or migrating existing Java codebases to Kotlin.

Using Kotlin or Java for Android development ultimately depends on the developer's preference, project requirements, and the team's familiarity with the language. Both languages are fully supported by Android Studio, the official Integrated Development Environment (IDE) for Android app development, making it seamless to use either language for building high-quality Android apps.

Swift & Objective-C for iOS

Swift and Objective-C are the primary programming languages used for iOS app development.

1. Objective-C: Objective-C was the original programming language used for iOS app development. It has been around for many years and has a mature ecosystem of libraries and tools. Many of the early iOS apps were written in Objective-C.
2. Swift: Swift is a modern and powerful programming language introduced by Apple in 2014. It is designed to be safe, fast, and expressive, with a syntax that is more user-friendly and easier to read than Objective-C. Swift has rapidly gained popularity among iOS developers and has become the preferred language for new iOS app development.

Both Objective-C and Swift can be used to build iOS apps, and they are fully interoperable. This means developers can use Objective-C code alongside Swift in the same project without any issues. Apple's Xcode, the official Integrated Development Environment (IDE) for iOS app development, supports both languages.

Over the years, Swift has seen widespread adoption, and many developers have transitioned from Objective-C to Swift for new projects or when updating existing apps. Swift's features such as type safety, optional chaining, and memory management improvements make it attractive for iOS app development.

The choice between Swift and Objective-C depends on the developer's preference, the requirements of the project, and the team's expertise. While Objective-C may still be

relevant for maintaining older projects or working with legacy codebases, Swift is becoming the go-to language for modern iOS app development.

Basics of React Native

React Native is a popular open-source framework for building cross-platform mobile apps. It allows developers to use the same codebase written in JavaScript and React to create native apps for both iOS and Android platforms. Here are the basics of React Native:

1. **JavaScript and React:** React Native apps are built using JavaScript, a widely used programming language for web development. React Native uses the React library, which allows developers to create reusable user interface components. React follows a declarative approach, where developers define how the app should look and behave based on the state of the application.
2. **Components:** In React Native, the user interface is composed of reusable components. Components are small, self-contained building blocks that encapsulate UI elements and logic. Developers can create their custom components or use pre-built components provided by the React Native library or community.
3. **Cross-Platform:** One of the key advantages of React Native is its ability to develop crossplatform apps. This means you can write a single codebase and deploy it on both iOS and Android platforms, reducing development time and effort.
4. **Native Modules:** Although the majority of the app is written in JavaScript, there are cases where developers may need access to native features and functionalities that are not available in React Native. In such cases, React Native allows developers to write native modules in Swift, Objective-C, Java, or Kotlin and expose them to the JavaScript layer.
5. **Hot Reloading:** React Native supports hot reloading, allowing developers to see the changes they make in the code immediately reflected in the running app, without the need to recompile the entire project. This makes the development process faster and more efficient.
6. **Styling:** React Native uses a style system similar to CSS for styling components. Developers can use Flexbox layout to create responsive and flexible UIs that adapt to different screen sizes and orientations.
7. **Community and Libraries:** React Native has a vibrant and active community, which has contributed to an extensive collection of third-party libraries and plugins. These libraries

help extend the functionality of React Native apps and make it easier to integrate with various services and features.

8. **Performance:** While React Native provides a near-native experience, it might not achieve the same level of performance as fully native apps, especially for complex and graphicsintensive applications. However, efforts have been made to optimize the framework for better performance.

Overall, React Native is a powerful tool for building cross-platform mobile apps, especially for projects that prioritize code reusability and quick development cycles. Its strong community support and active development make it an attractive choice for many developers and businesses.

Native Components

Native components in the context of React Native refer to UI elements that are rendered using native platform-specific views. These components allow developers to build user interfaces that look and feel like native apps on iOS and Android. React Native bridges JavaScript code with the native platform's components to provide a seamless and performant user experience.

Some examples of native components in React Native include:

1. **View:** The **<View>** component is similar to a **<div>** in web development. It is a container used to group and layout other components and does not render any visual output itself.
2. **Text:** The **<Text>** component is used for displaying text content. It renders text in a platform-specific manner, taking into account the platform's fonts and text rendering features.
3. **Image:** The **<Image>** component is used for displaying images. It loads and displays images efficiently using the native platform's image handling capabilities.
4. **ScrollView:** The **<ScrollView>** component provides a scrollable view, enabling users to scroll through a list or content that exceeds the screen's height.
5. **TextInput:** The **<TextInput>** component is used for capturing user input. It provides a native input field where users can enter text.
6. **Button:** The **<Button>** component is used for creating buttons that trigger actions when pressed. It is styled as a platform-specific button.

7. **StatusBar**: The `<StatusBar>` component allows developers to customize the status bar (the area displaying time, battery, and other indicators) on the device's screen.
8. **Touchable Components**: React Native provides various touchable components like `<TouchableOpacity>`, `<TouchableHighlight>`, and `<TouchableWithoutFeedback>`, allowing developers to add touch interaction to elements.

The key advantage of using native components is that they provide a more authentic native look and feel, as they leverage the native platform's UI elements. React Native bridges the gap between JavaScript and the native platform, ensuring that the app's UI components are rendered natively, resulting in better performance and user experience.

By using these native components, React Native developers can build apps that are visually consistent with native apps on both iOS and Android, making the development process efficient and enabling code reuse across different platforms.

JSX

JSX (JavaScript XML) is a syntax extension for JavaScript that allows developers to write HTML-like code within JavaScript. It is commonly used with libraries and frameworks like React and React Native to define the structure of user interfaces. JSX provides a more concise and readable way to create UI components compared to manually manipulating the DOM in JavaScript.

In JSX, you can write HTML-like elements directly in your JavaScript code. For example:

```
const element = <div>Hello, JSX!</div>;
```

In the above example,

We are defining a JSX element with a `<div>` tag containing the text "Hello, JSX!". This JSX code will be transformed into equivalent JavaScript code by a transpiler (like Babel) before it's executed in the browser or on a mobile device.

JSX elements can also include attributes and support JavaScript expressions within curly braces `{}`. For instance:

```
const name = "John"; const element = <div>Hello, {name}!</div>;
```

In this example, the **name** variable is used within the JSX expression to display dynamic content.

JSX makes it easier for developers to express the UI components in a more declarative manner, similar to how they would describe the desired UI structure in HTML. Under the hood, JSX gets transpiled to JavaScript function calls that create React elements, which are then rendered to the DOM or native components by React or React Native respectively.

JSX is not mandatory when using React or React Native, but it's widely adopted due to its readability and ease of use. Developers can write the equivalent JavaScript code without JSX, but using JSX makes the code more expressive and easier to understand, especially for UI-intensive applications.

JSX States

In React (and React Native), states refer to the dynamic data that can change during the lifetime of a component. In JSX, you can use states to manage and update the content of your components, allowing them to respond to user interactions or changes in the application's data.

To use states in a React or React Native component, you typically follow these steps:

1. **Initialize State:** You define the initial state of your component within its constructor or using the **useState** hook (in functional components). The state is usually defined as an object containing key-value pairs representing different data elements.

Example using a class component:

```
import React, { Component } from 'react';

class MyComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {
```

```
    counter: 0,  
    message: 'Hello, World!'  
  };  
}  
  
// ...  
}
```

Example using a functional component with the **useState** hook:

```
function MyComponent() {  
  const [counter, setCounter] = useState(0);  
  const [message, setMessage] = useState('Hello, World!');  
  
  // ...  
}
```

import React, { useState } from 'react';

2. Update State: To update the state, you use the **setState** method (in class components) or the state setter function (in functional components).

Example using a class component:

```
class MyComponent extends Component {  
  constructor(props) {  
    // ...  
  }  
  
  handleIncrement = () => {  
    this.setState({ counter: this.state.counter + 1 });  
  };  
  
  render() {  
    return (  
      <div>  
        <p>Counter: {this.state.counter}</p>  
        <button onClick={this.handleIncrement}>Increment</button>  
      </div>  
    );  
  }  
}
```

Example using a functional component with the **useState** hook:

```
function MyComponent() {  
  const [counter, setCounter] = useState(0);  
  
  const handleIncrement = () => {  
    setCounter(counter + 1);  
  };  
  
  return (  
    <div>  
      <p>Counter: {counter}</p>  
      <button onClick={handleIncrement}>Increment</button>  
    </div>  
  );  
}
```

When the state is updated, React will automatically re-render the component and update parts of the JSX that depend on the changed state. This allows you to build interactive and dynamic UI components that respond to user actions or data changes. States are an essential concept in React and React Native development, enabling developers to create powerful and reactive user interfaces.

Props of JSX

JSX, "props" (short for "properties") are a way to pass data from a parent component to a child component. They allow you to customize and configure child components by providing them with specific values or functions.

When you use JSX, you can pass props to a component by adding attributes to the JSX element. These attributes are then accessible within the child component as properties.

Here's how you can pass props to a child component in JSX:

ParentComponent.jsx:

```
import React from 'react';
import ChildComponent from './ChildComponent';

function ParentComponent() {
  // Define the props
  const name = "John";
  const age = 30;

  return (
    <div>
      { /* Pass the props to the ChildComponent */ }
      <ChildComponent name={name} age={age} />
    </div>
  );
}

export default ParentComponent;
```

ChildComponent.jsx:

```
import React from 'react';
function ChildComponent(props) {
  // Access the props inside the ChildComponent
  return (
    <div>
      <p>Name: {props.name}</p>
      <p>Age: {props.age}</p>
    </div>
  );
}
export default ChildComponent;
```

In this example, the **ParentComponent** passes the **name** and **age** props to the **ChildComponent**. Inside the **ChildComponent**, the props are accessed through the **props** object.

Props are read-only, meaning that a child component cannot modify its own props. They are used for providing data and communication between parent and child components. If a child component needs to manage its own data that can change, it typically uses states (in class components) or the **useState** hook (in functional components).

props in JSX enables you to create reusable and configurable components, allowing you to pass different data or behavior to the same component based on the requirements of the parent component. This composability is one of the key features that make React and JSX powerful for building complex UI hierarchies.

Develop an app using Java - step by step explanation

Developing an Android app using Java involves several steps. Here's a step-by-step explanation of the process:

Step 1: Setup Development Environment

- Install Android Studio, the official Integrated Development Environment (IDE) for Android app development, from the official website.
- Open Android Studio, set up the Android SDK, and create a new Android project.

Step 2: Design the User Interface (UI)

- Use XML files (located in the "res/layout" directory) to design the app's user interface using views like TextView, EditText, Button, etc.
- You can use Android's visual layout editor in Android Studio or manually edit the XML files to create the UI.

Step 3: Handle User Interaction

- Implement event handling in Java code to respond to user interactions with UI elements.
- Use listeners like OnClickListener to handle button clicks and other user actions.

Step 4: Handle Activity Lifecycle

- Understand and handle the lifecycle of the Android activity (screen) where the UI components are displayed.

- Override methods like onCreate, onStart, onResume, onPause, etc., to manage the activity's state during different lifecycle events.

Step 5: Implement Business Logic

- Write Java code to implement the app's business logic, such as data processing, calculations, and data manipulation.

Step 6: Data Storage (Optional)

- If your app requires data storage, you can use Android's built-in SQLite database or other data storage options like SharedPreferences or external storage.

Step 7: Testing

- Test your app thoroughly to ensure it works as expected on different devices and screen sizes.
- Use Android Virtual Device (AVD) in Android Studio to test on various virtual devices or test on physical Android devices.

Step 8: Debugging

- Use Android Studio's debugger to identify and fix any issues in your app's code.

Step 9: Optimizing Performance

- Optimize your app for better performance by managing memory usage, using background threads for time-consuming tasks, and following best practices.

Step 10: Publish the App

- Once your app is tested and ready, generate a signed APK (Android Package) using Android Studio.
- Upload the APK to Google Play Store or other distribution platforms to make it available for users to download and install.

Step 11: Maintenance and Updates

- Monitor user feedback and app performance after publishing.
- Regularly update your app to fix bugs, add new features, and stay compatible with the latest Android versions.

Remember that this step-by-step explanation provides an overview of the Android app development process using Java. The actual implementation may vary based on your app's specific requirements and complexity. As you progress, you can explore more

advanced topics like networking, push notifications, user authentication, and integrating third-party libraries to enhance your app's functionality.

Install Android studio

Note: This version of the codelab requires Android Studio 3.6 or higher.

We can download Android Studio 3.6 from the Android Studio page.

Android Studio provides a complete IDE, including an advanced code editor and app templates. It also contains tools for development, debugging, testing, and performance that make it faster and easier to develop apps. We can use Android Studio to test your apps with a large range of preconfigured emulators, or on your own mobile device. We can also build production apps and publish apps on the Google Play store.

Note: Android Studio is continually being improved. For the latest information on system requirements and installation instructions, see the Android Studio download page.

Android Studio is available for computers running Windows or Linux, and for Macs running macOS. The OpenJDK (Java Development Kit) is bundled with Android Studio.

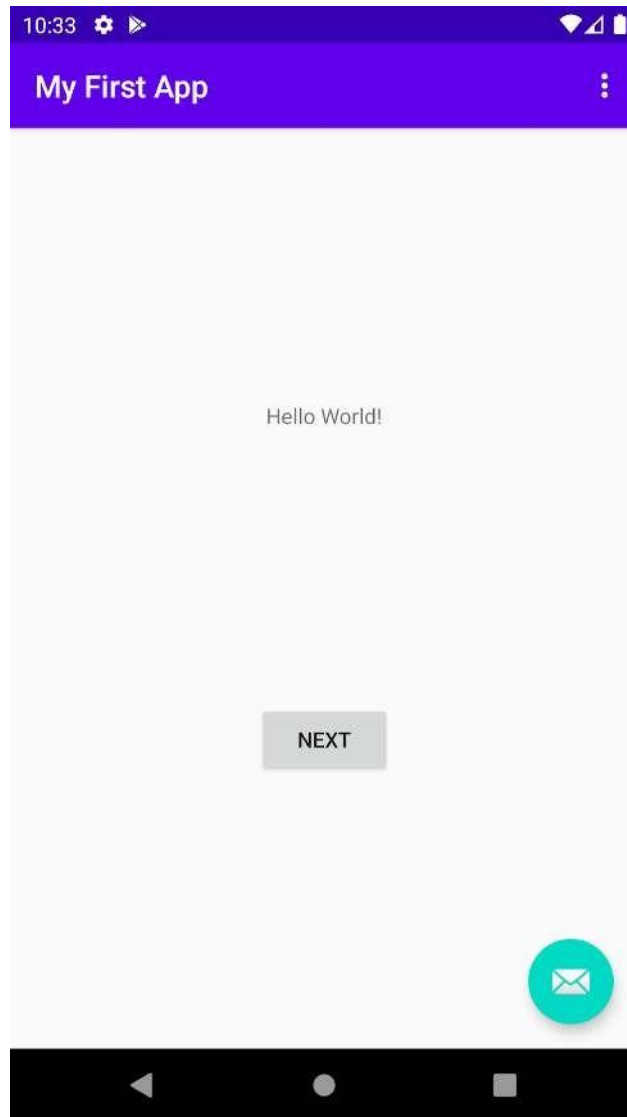
The installation is similar for all platforms. Any differences are noted below.

1. Navigate to the Android Studio download page and follow the instructions to download and install Android Studio.
2. Accept the default configurations for all steps, and ensure that all components are selected for installation.
3. After the install is complete, the setup wizard downloads and installs additional components, including the Android SDK. Be patient, because this process might take some time, depending on Our internet speed.
4. When the installation completes, Android Studio starts, and We are ready to create Our first project.

Create a new Android project for our first app:

In this step, you will create a new Android project for your first app. This simple app displays the string "Hello World" on the screen of an Android virtual or physical device.

Here's what finished app will look like:

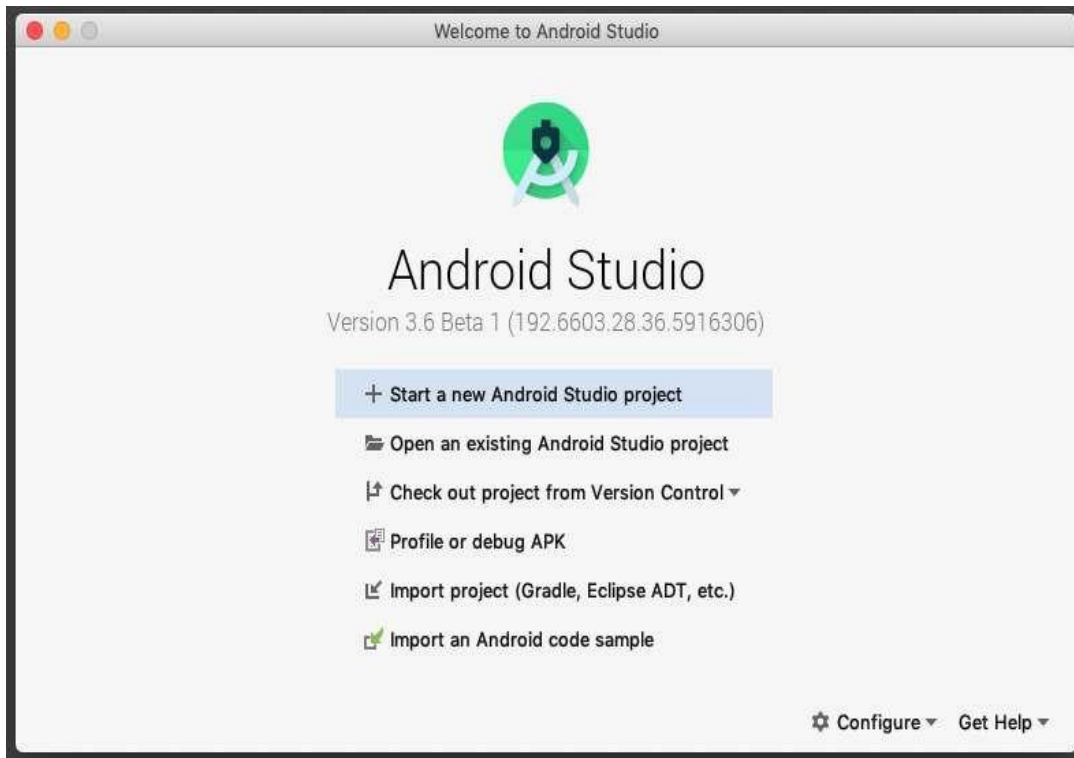


What we'll learn

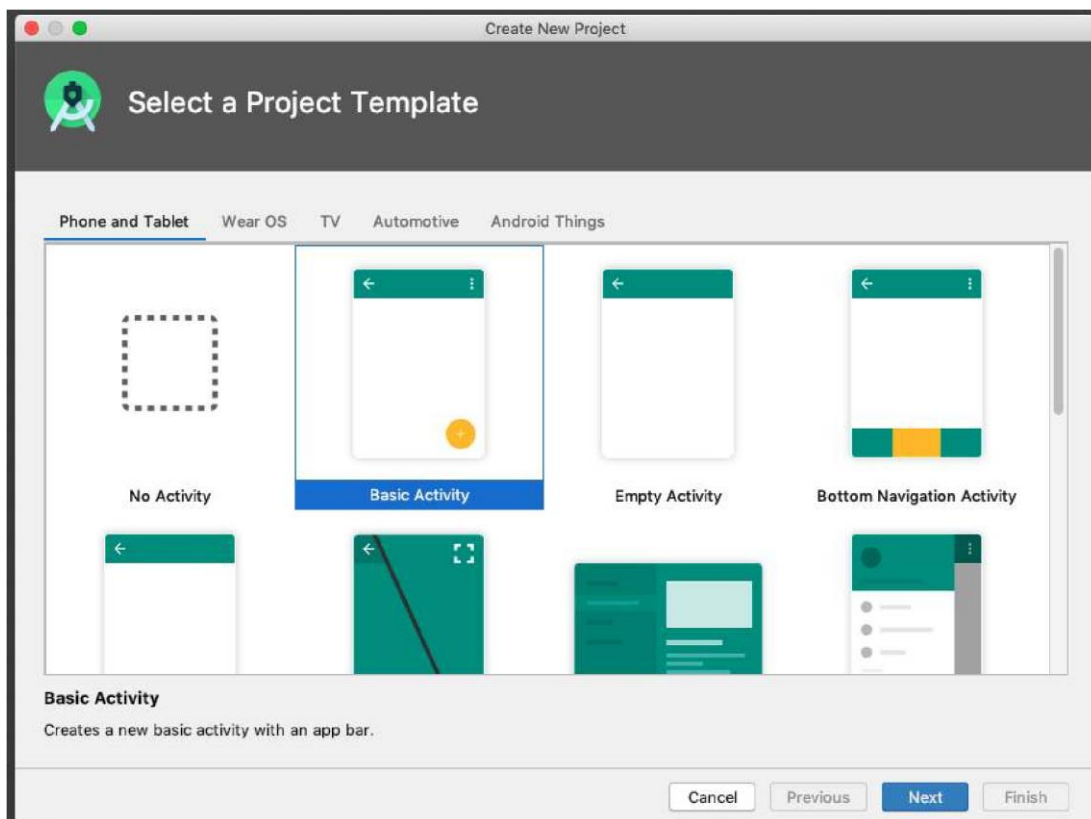
- How to create a project in Android Studio.
- How to create an emulated Android device.
- How to run our app on the emulator.
- How to run our app on your own physical device, if we have one.

Step 1: Create a new project

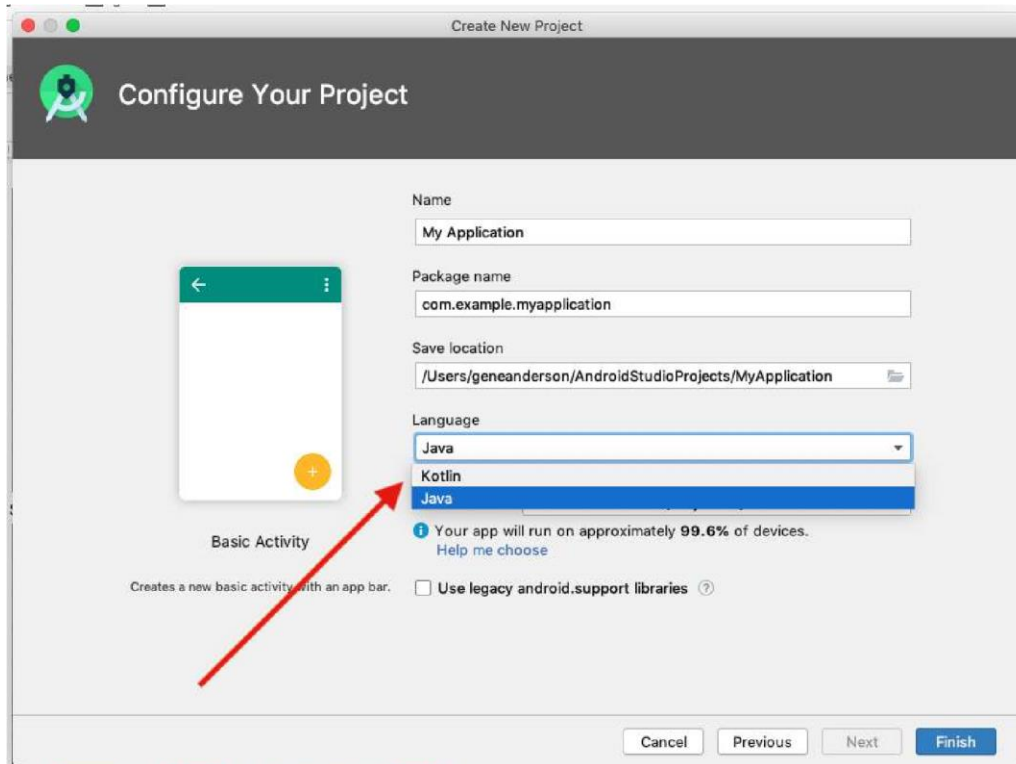
1. Open Android Studio.
2. In the **Welcome to Android Studio** dialog, click **Start a new Android Studio project**



3. Select **Basic Activity** (not the default). Click **Next**.



4. Give our application a name such as **My First App**.



5. Make sure the **Language** is set to **Java**.
6. Leave the defaults for the other fields.
7. Click **Finish**.

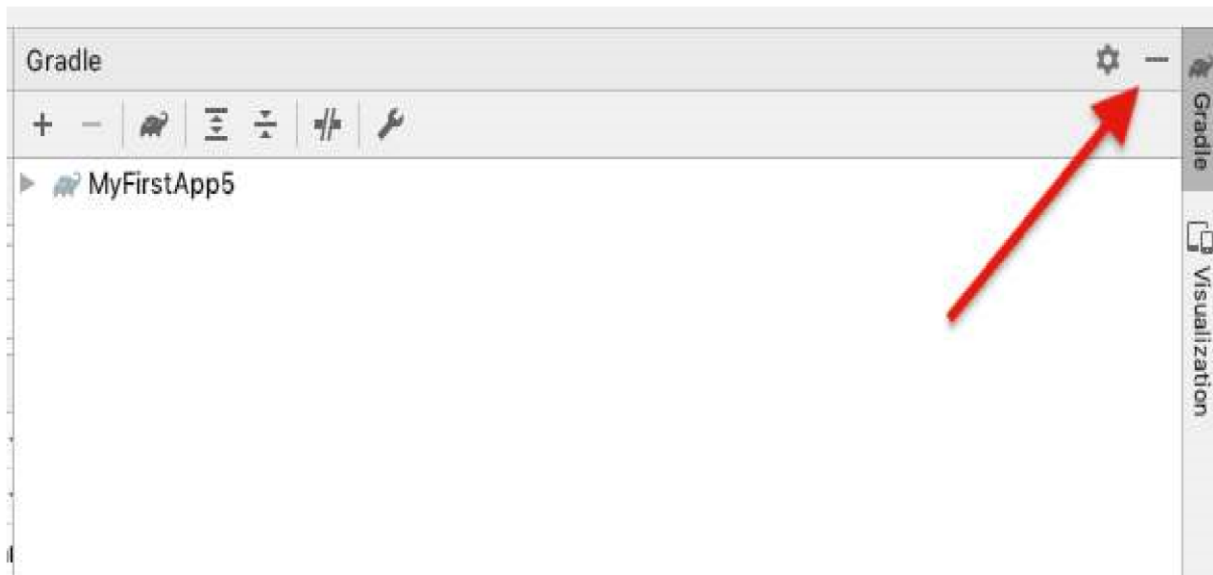
After these steps, Android Studio:

- Creates a folder for our Android Studio project called **MyFirstApp**. This is usually in a folder called **AndroidStudioProjects** below our home directory.
- Builds our project (this may take a few moments). Android Studio uses [Gradle](#) as its build system. We can follow the build progress at the bottom of the Android Studio window.
- Opens the code editor showing our project.

Step 2: Get our screen set up

When our project first opens in Android Studio, there may be a lot of windows and panes open. To make it easier to get to know Android Studio, here are some suggestions on how to customize the layout.

1. If there's a **Gradle** window open on the right side, click on the minimize button (—) in the upper right corner to hide it.



2.D

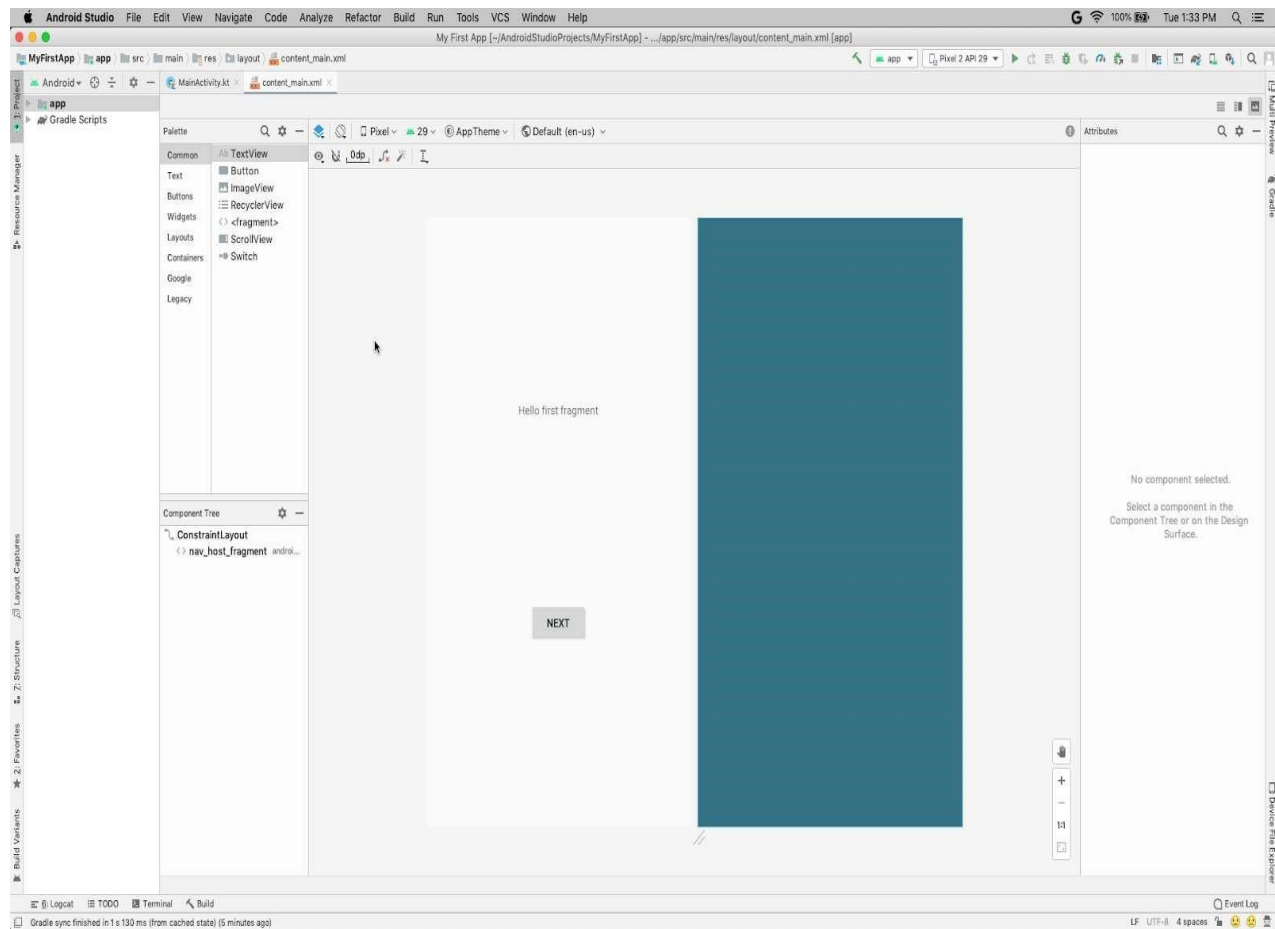
e p e n d

n g

i

on the size of our screen, consider resizing the pane on the left showing the project folders to take up less space.

At this point, our screen should look a bit less cluttered, similar to the screenshot shown below.



Step 3: Explore the project structure and layout

The upper left of the Android Studio window should look similar to the following diagram:

Based on We selecting the **Basic Activity** template for our project, Android Studio has set up a number of files for you. We can look at the hierarchy of the files for our app in multiple ways, one is in **Project** view. **Project** view shows our files and folders structured in a way that is convenient for working with an Android project. (This does not always match the file hierarchy! To see the file hierarchy, choose the **Project files** view by clicking (3).)

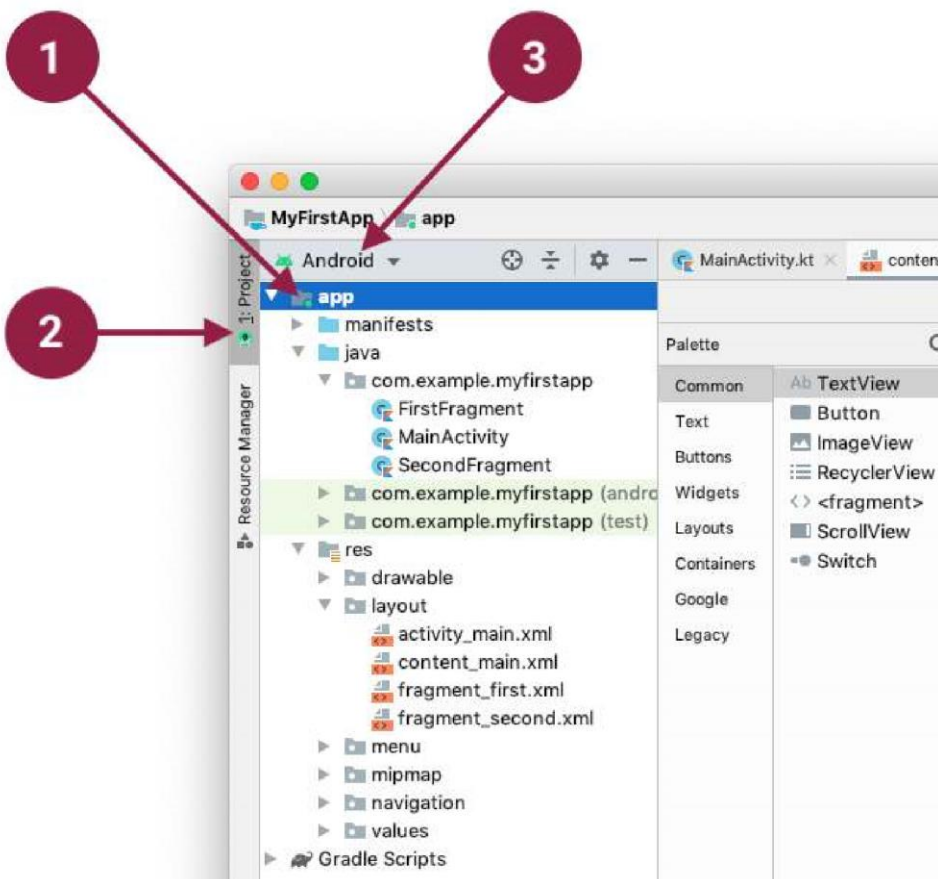
1. Double-click the **app** (1) folder to expand the hierarchy of app files. (See (1) in the screenshot.)
2. If We click **Project** (2), We can hide or show the **Project** view. We might need to select **View > Tool Windows** to see this option.
3. The current **Project** view selection (3) is **Project > Android**.

In the **Project > Android** view we see three or four top-level folders below our **app** folder: **manifests**, **java**, **java (generated)** and **res**. We may not see **java (generated)** right away.

1. Expand the **manifests** folder.

This folder contains **AndroidManifest.xml**. This file describes all the components of our Android app and is read by the Android runtime system when our app is executed. 2. Expand the **java** folder. All our Java language files are organized here. The **java** folder contains three subfolders: **com.example.myfirstapp**: This folder contains the Java source code files for our app. **com.example.myfirstapp (androidTest)**: This folder is where we would put our instrumented tests, which are tests that run on an Android device. It starts out with a skeleton test file.

com.example.myfirstapp (test): This folder is where we would put our unit tests. Unit tests don't need an Android device to run. It starts out with a skeleton unit test file. 3. Expand the **res** folder. This folder contains all the resources for our app, including images, layout files, strings, icons, and styling. It includes these subfolders: **drawable**: All our app's images will be stored in this folder.



layout: This folder contains the UI layout files for our activities. Currently, our app has one activity that has a layout file called **activity_main.xml**. It also contains **content_main.xml**, **fragment_first.xml**, and **fragment_second.xml**.

menu: This folder contains XML files describing any menus

in our app. **mipmap**: This folder contains the launcher icons for our app.


navigation: This folder contains the navigation graph, which tells Android Studio how to navigate between different parts of our application.

values: This folder contains resources, such as strings and colors, used in our app.

Step 4: Create a virtual device (emulator)

In this task, we will use the [Android Virtual Device \(AVD\) manager](#) to create a virtual device (or emulator) that simulates the configuration for a particular type of Android device.

The first step is to create a configuration that describes the virtual device.

1. In Android Studio, select **Tools > AVD Manager**, or click the AVD Manager icon in the toolbar. 
2. Click **+Create Virtual Device**. (If you have created a virtual device before, the window shows all of our existing devices and the **+Create Virtual Device** button is at the bottom.) The **Select Hardware** window shows a list of pre-configured hardware device definitions.
3. Choose a device definition, such as **Pixel 2**, and click **Next**. (For this codelab, it really doesn't matter which device definition We pick).
4. In the **System Image** dialog, from the **Recommended** tab, choose the latest release. (This does matter.)
5. If a **Download** link is visible next to a latest release, it is not installed yet, and We need to download it first. If necessary, click the link to start the download, and click **Next** when it's done. This may take a while depending on our connection speed.



Note: System images can take up a large amount of disk space, so just download what We need.

6. In the next dialog box, accept the defaults, and click **Finish**.

The AVD Manager now shows the virtual device we added.

7. If the **Our Virtual Devices** AVD Manager window is still open, go ahead and close it. **Step 5:**

Run our app on our new emulator

1. In Android Studio, select **Run > Run 'app'** or click the **Run** icon in the toolbar.  The icon will change when our app is already running. 

If we get a dialog box stating "Instant Run requires that the platform corresponding to our target device (Android N...) is installed" go ahead and click **Install and continue**.

2. In **Run > Select Device**, under **Available devices**, select the virtual device that We just configured.

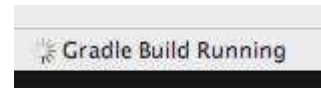
This menu also appears in the toolbar.



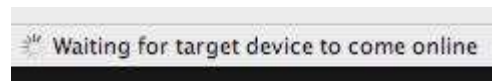
The emulator starts and boots just like a physical device. Depending on the speed of our computer, this may take a while. We can look in the small horizontal status bar at the very bottom of Android Studio for messages to see the progress.

Messages that might appear briefly in the status bar

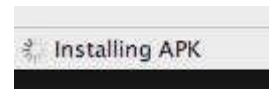
Gradle build running



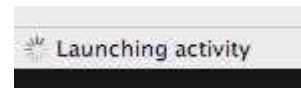
Waiting for target device to come on line



Installing APK



Launching activity



Once our app builds and the emulator is ready, Android Studio uploads the app to the emulator and runs it. We should see our app as shown in the following screenshot.



Note: It is a good practice to start the emulator at the beginning of our session. Don't close the emulator until we are done testing our app, so that we don't have to wait for the emulator to boot again. Also, don't have more than one emulator running at once, to reduce memory usage.

Step 6: Run our app on a device (if we have one)

What we need:

- An Android device such as a phone or tablet.
- A data cable to connect our Android device to our computer via the USB port.
- If we are using a Linux or Windows OS, We may need to perform additional steps to run our app on a hardware device.

Run our app on a device


To let Android Studio communicate with our device, we must turn on USB Debugging on our Android device.

On Android 4.2 and higher, the Developer options screen is hidden by default. To show Developer options and enable USB Debugging:



1. On our device, open **Settings > About phone** and tap **Build number** seven times.
2. Return to the previous screen (**Settings**). **Developer options** appears at the bottom of the list.
Tap **Developer options**.
3. Enable **USB Debugging**.

Now we can connect our device and run the app from Android Studio.

1. Connect our device to our development machine with a USB cable. On the device, we might need to agree to allow USB debugging from our development device.
2. In Android Studio, click **Run**  in the toolbar at the top of the window. (You might need to select **View > Toolbar** to see this option.) The **Select Deployment Target** dialog opens with the list of available emulators and connected devices.
3. Select our device, and click **OK**. Android Studio installs the app on our device and runs it.

Note: If our device is running an Android platform that isn't installed in Android Studio, We might see a message asking if we want to install the needed platform. Click **Install and Continue**, then click **Finish** when the process is complete.

Troubleshooting

If we're stuck, quit Android Studio and restart it.

If Android Studio does not recognize our device, try the following:

1. Disconnect our device from our development machine and reconnect it.
2. Restart Android Studio.

If our computer still does not find the device or declares it "unauthorized":

1. Disconnect the device.
2. On the device, open **Settings->Developer Options**.
3. Tap **Revoke USB Debugging authorizations**.
4. Reconnect the device to our computer.
5. When prompted, grant authorizations.

If you are still having trouble, check that we installed the appropriate USB driver for our device.

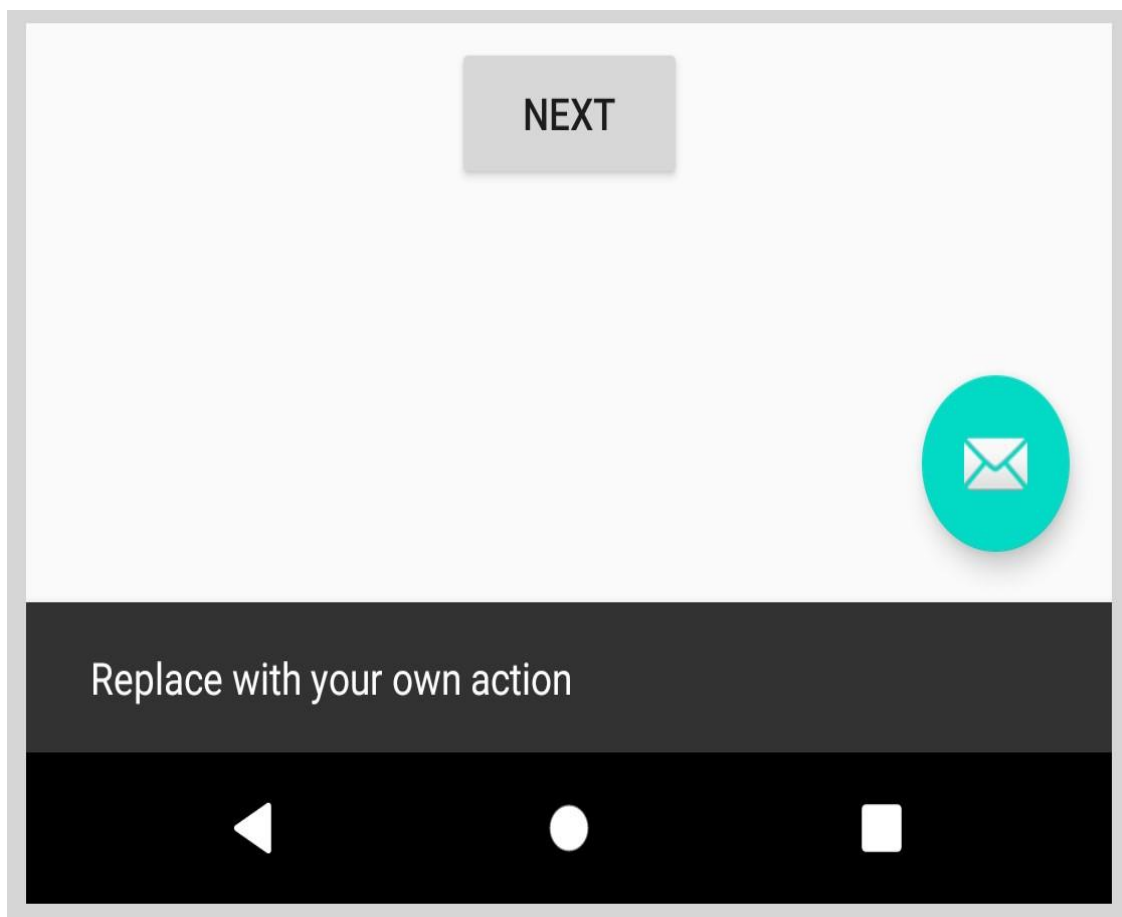
Step 7: Explore the app template

When we created the project and selected **Basic Activity**, Android Studio set up a number of files, folders, and also user interface elements for us, so we can start out with a working app and major components in place. This makes it easier to build our application.

Looking at our app on the emulator or our device, in addition to the **Next** button, notice the floating action button with an email icon.



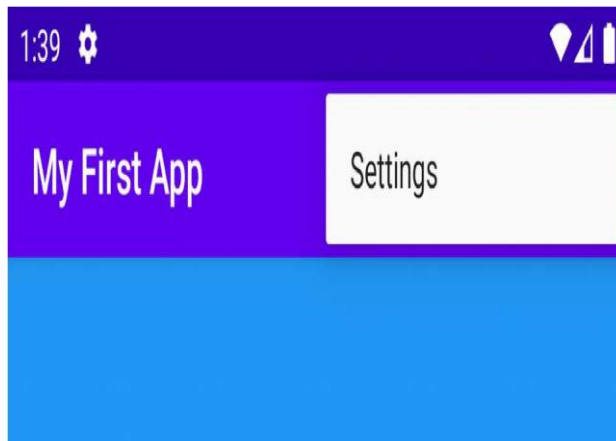
If you tap that button, you'll see it has been set up to briefly show a message at the bottom of the screen. This message space is called a snackbar, and it's one of several ways to notify users of our app with brief information.



At the top right of the screen, there's a menu with 3 vertical dots.

If we tap on that, we'll see that Android Studio has also created an options menu with a **Settings** item.

Choosing **Settings** doesn't do anything yet, but having it set up for us makes it easier to add user-configurable Settings to our app.



Later in this codelab, we'll look at the **Next** button and modify the way it looks and what it does.

References

<https://www.mobiloud.com/blog/native-web-or-hybrid-apps>