LESSON 2

MOBILE APPLICATION DEVELOPMENT ARCHITECTURE

Before starting building an app right away, a team of experts should define customer's requirements and decompose them into small logical chunks for mobile developers to code. Once the team gets the idea of what features they need to build in the app, they prepare the ***app architecture*** – (a skeleton that binds these features together into one app). The success of any mobile app requires a careful consideration of architecture and tech stack. Paying attention to your business requirements and user stories can help plan what features to include in the app, how to include them, and how these features will be connected through layers. There are many factors in the decision which, when incorrect assumptions are at play, can result in an app that fails to meet expectations.

What is Mobile App Architecture?

Mobile app architecture refers to a set/body of rules, techniques, processes, and patterns to develop a mobile application. These rules help developers create an app that meets both the customer's requirements as well as industry standards.

Mobile App Architecture vs Mobile Tech Stack

Mobile app architecture is often used interchangeably, although incorrectly, with the mobile tech stack. The ***mobile technology stack*** is the set of technologies and technical frameworks that make up the front and back-end of a mobile or web app (the what of the app), but is less concerned with the business / customer requirements (the why of the app) or the development process (the how of creating the app).
The mobile app architecture is made up of all the parts of the app – all the questions about why, what, how – including what data is collected, how the data moves, what the app looks like, for what platform, using what tech stack.
An application stack is a suite or set of application programs that help in performing a
certain task

What Makes a Good Mobile App Architecture

There are many apps developed today without any architecture or reference to standards. A lack of architecture results in an app that is:

- Longer and more costly to develop
- Hard to maintain, particularly if staff change
- Harder to build upon or scale
- Difficult to test
- More prone to errors

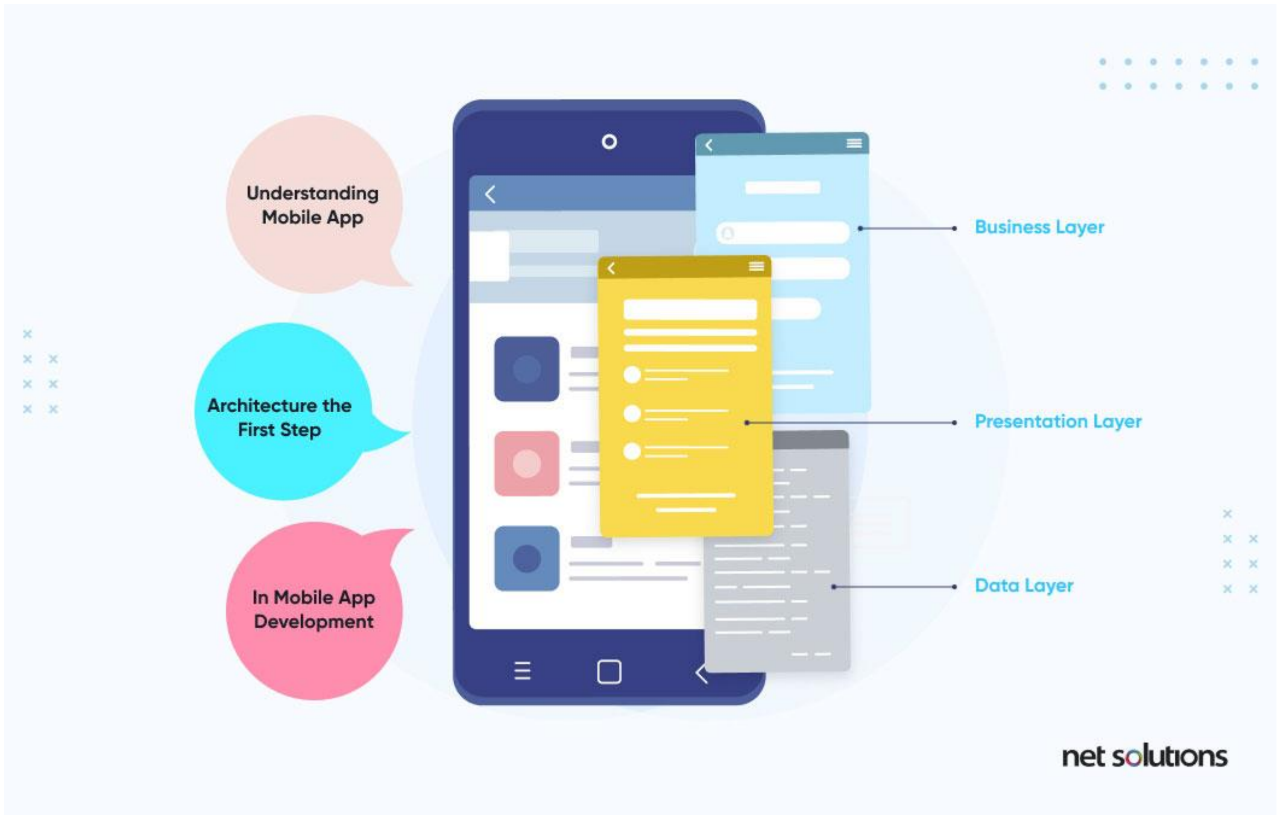
A good mobile application architecture will enforce good software development principles (KISS, DRY, SOLID) in the appropriate stages of development to help accelerate development, providing a clear path for data flow that makes work easier and also supports clarity over how to scale or expand the app in the future.

- SOLID – 5 principles of object-oriented programming for building easy to maintain and scalable apps,
- KISS – a principle of keeping the system and code simple to minimize the number of errors,
- DRY – a principle of reducing repetition in software patterns to avoid redundancy, and other principles.

A clearly defined mobile app architecture helps to support flexibility and Agile development methods, makes testing more efficient, and makes future maintenance easier and less prone to bugs. A strong mobile app architecture will save both time and money in the short and long term. A good architecture will not be platform-specific, but rather apply to native and cross-platform choices, resulting in a unified approach to development. If we think of mobile app architecture as a skeleton for how we create a mobile app, we can then define layers (the bones, if you will) for how we build out the vital components of the app.

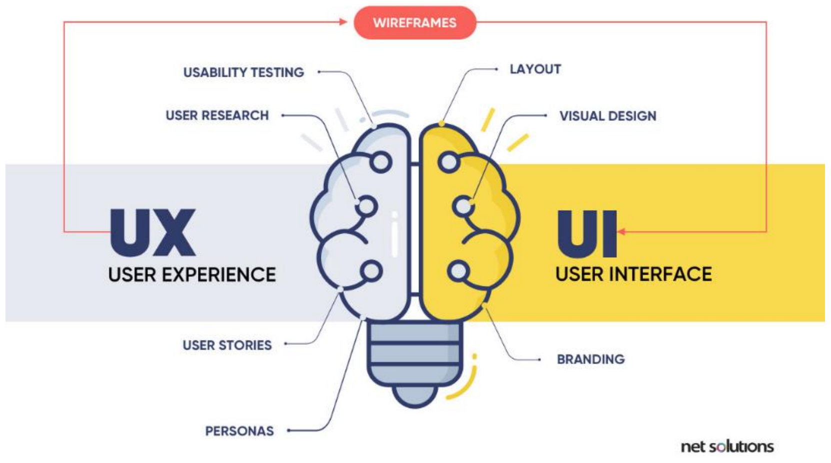
Layers in Mobile App Architecture

The most popular representation of mobile app architecture is represented by 3 layers: presentation, business logic, and data.



1. **Presentation Layer**

The presentation layer consists of all the processes and components to deliver the app to the user. When building the presentation layer, developers are concerned with what the user sees and feels when using the app. In other terms, the presentation layer is made up of the user interface (UI) and user experience (UX).



- **User Interface (UI)** is concerned with design questions such as colors, fonts, placement, and overall design.
- **User Experience (UX)** manages the way a customer interacts with the app through a detailed understanding of what a user wants and feels.

When designing the presentation layer, developers need to determine the right platform and device type so that the presentation is consistent with the standards for each.

2**. Business Layer**

The business layer is concerned with the logic and rules responsible for data exchange, operations, and workflow regulation. This layer is responsible for:

- Security
- Data caching
- Logging
- Data validation
- Exception management

The business layer can be deployed/ can exist on the backend server or on a user's device, depending on the quantity of operations your app performs and the number of resources available on a user's device..

3. **Data Layer**

The data layer is responsible for ***data safety and maintenance***. It includes all the data utilities, service agents, and data access components to support data transactions. This layer can be thought of in two parts:

- Persistence – data access with data sources via API
- Network – network communication, routing, error reporting

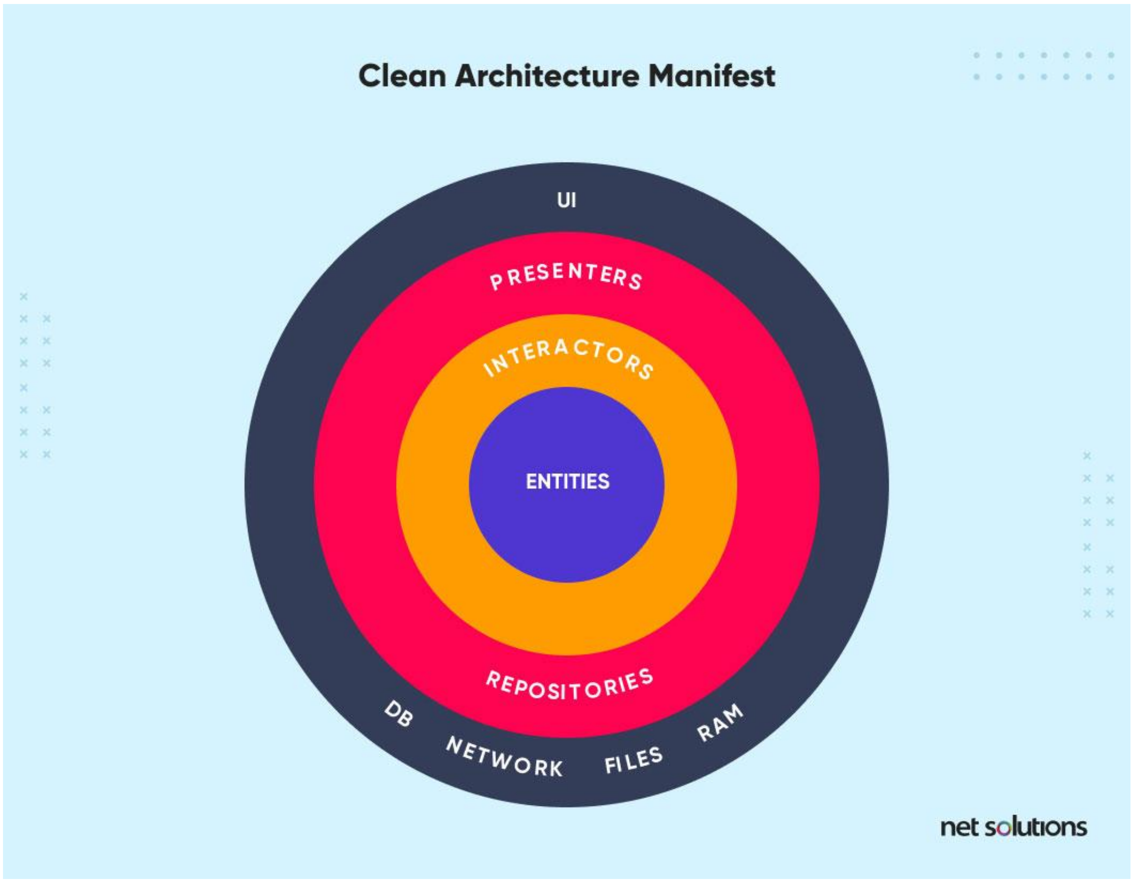
When building this layer, software developers should keep in mind that it can scale with the change of business requirements in the future. It is also important to choose the right technology for data access and validation to ensure that this layer is secure from invalid data input and functions well.

CHOOSING THE RIGHT ARCHITECTURE FOR MOBILE APP

Android Mobile Application Architecture

Apps developed specifically for Android are one kind of native app –(an app developed for a specific mobile platform). Android apps are developed to support Android languages (Kotlin and Java) for devices from a variety of manufacturers including Google, Samsung, Sony, and Nokia. There is not a single architecture recommended for Android, but the most commonly accepted Android architecture for mobile apps is ***Clean Architecture***.

In Clean, the architecture is built on the principles of layers and inversion of control. Clean focuses on the same 3-layer structure modeled above, with the business layer sometimes referred to as the domain layer. In Clean architecture, the domain / business layer must not depend on the other layers, but instead must leverage interfaces. While this can be difficult to understand, it does make it easy to add to and scale apps over time.



The Clean architecture is usually represented as a circle of four layers:

i. **Entities** – a business logic;
ii. **Use Cases/Interactors** – app logic;
iii. **Interface Adapters** – the adapters that convert data from the use cases format to the external agency format such as Database or Web; Interface adapters include Presenters Controllers, and Gateways;
iv. **Frameworks and Drivers** – is the outer layer composed of frameworks and tools, e.g. The Databases, UI, http-client, and others.

To maintain the Dependency Rule, when the business logic and the app logic don't depend on Presenters, UIs, and Databases, each layer has its ***Boundaries.*** These Boundaries establish communication between layers, providing them with two interfaces – the output port for the answer and the input port for the request.

An explicit structure and layer independency make the Clean Architecture:

- simple to test and troubleshoot;
- independent from UI;
- independent from databases, external frameworks, and libraries;
- easy to install various plug-ins.

iOS Mobile Application Architecture

Unlike Android, the Apple system offers software developers more guidance on how to develop iOS mobile architecture based on the **MVC** model (Model-View-Controller). However, iOS developers aren't limited to only one architectural pattern, this one is the most commonly used in iOS apps. Native iOS apps are developed using Objective-C and Swift languages, with Apple providing clear best practices over app architecture with the MVC model (Model-View-Controller). While other options are available for iOS, the MVC model is made up of:

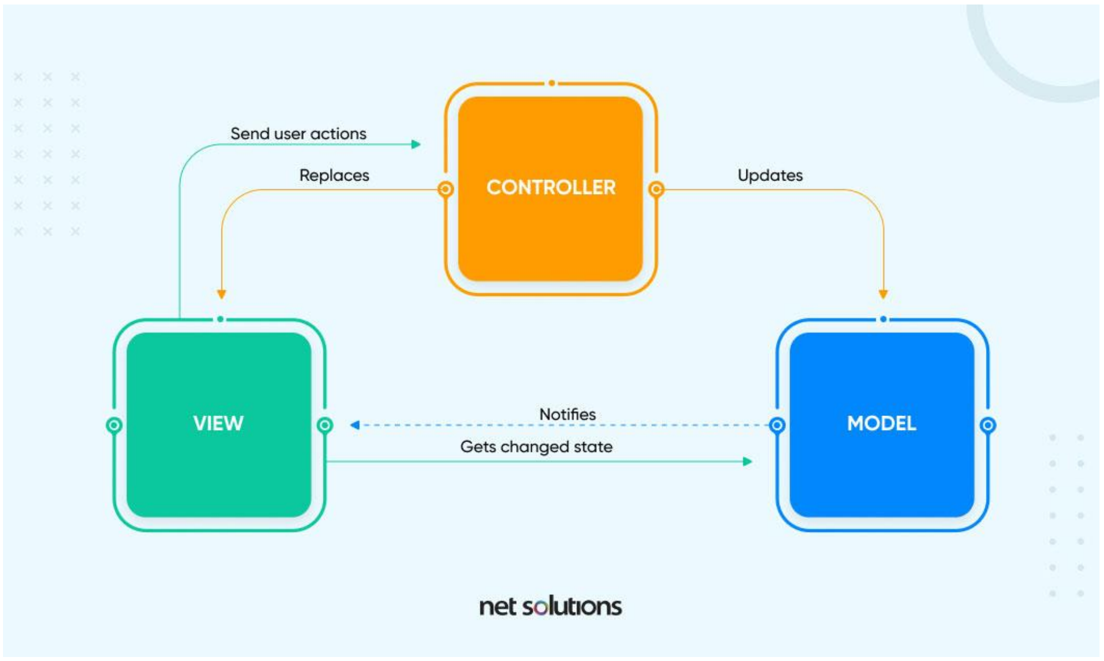
- ***Model*** – The data layer (persistence, model objects, parsers, managers, networking code).
- ***View*** – Similar to the presentation layer, this is the layer that interacts with the users and doesn't have any domain-specific logic, therefore, the classes in this layer are reusable;.

- ***Controller*** – is the layer that mediates the interaction between the view and the model

The MVC working principle is pretty simple. The user interacts with the iOS app and performs an action in the *view layer*. The view transfers the action to the controller to handle it. The controller processes the received action and makes some decisions. If necessary, it can turn to Model and implement some changes there. The model changes data values and sends them back to the controller. The controller, in its turn, sets the values to the view and the view presents the results to the user.

Using the MVC model, iOS developers:

- Significantly accelerate mobile app development process;
- Develop transparent communication between app layers;
- Get a well-structured, simple-to-maintain codebase;
- Get a simple-to-test codebase.

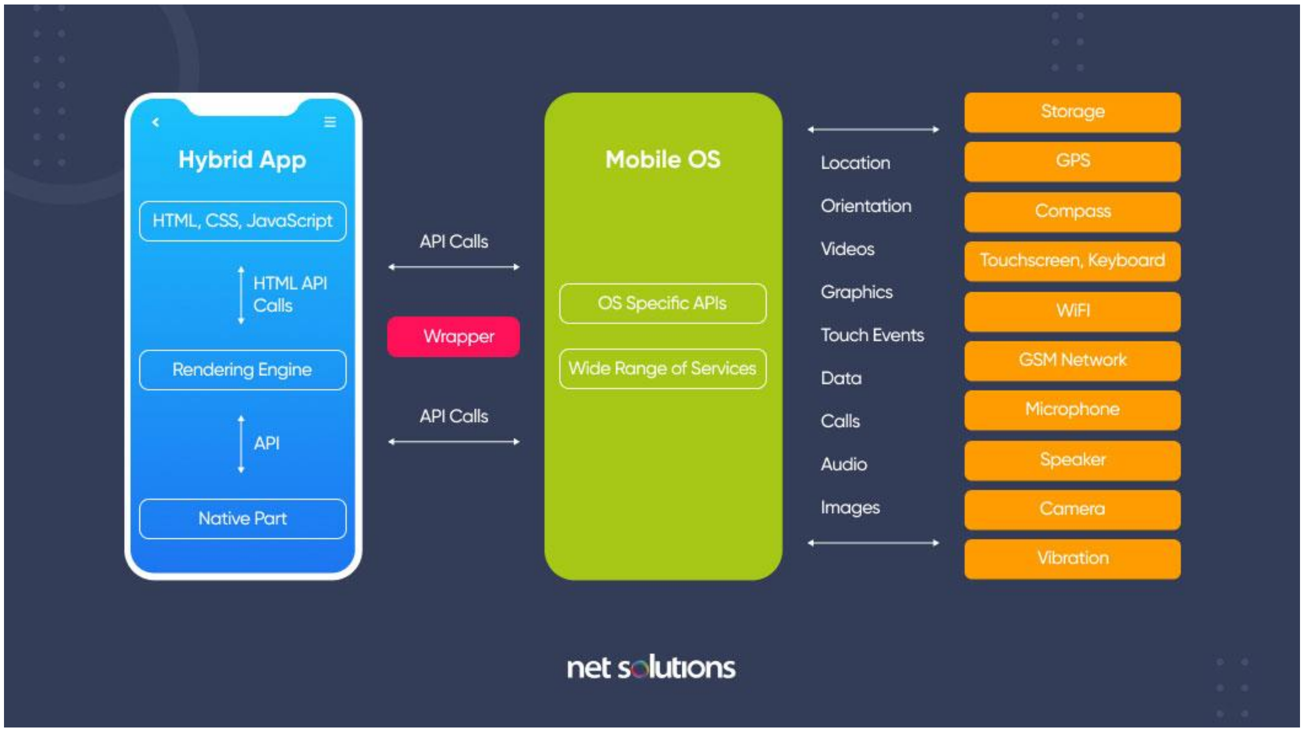


Hybrid Mobile Application Architecture

Hybrid mobile apps leverage both native and web solutions. Hybrid apps use native apps as "shells" for the back-end, but platform-neutral JavaScript, HTML and CSS for the front-end. Hybrid apps use plugins such as ***Apache Cordova*** or ***Ionic Capacitor*** to access native platform features.

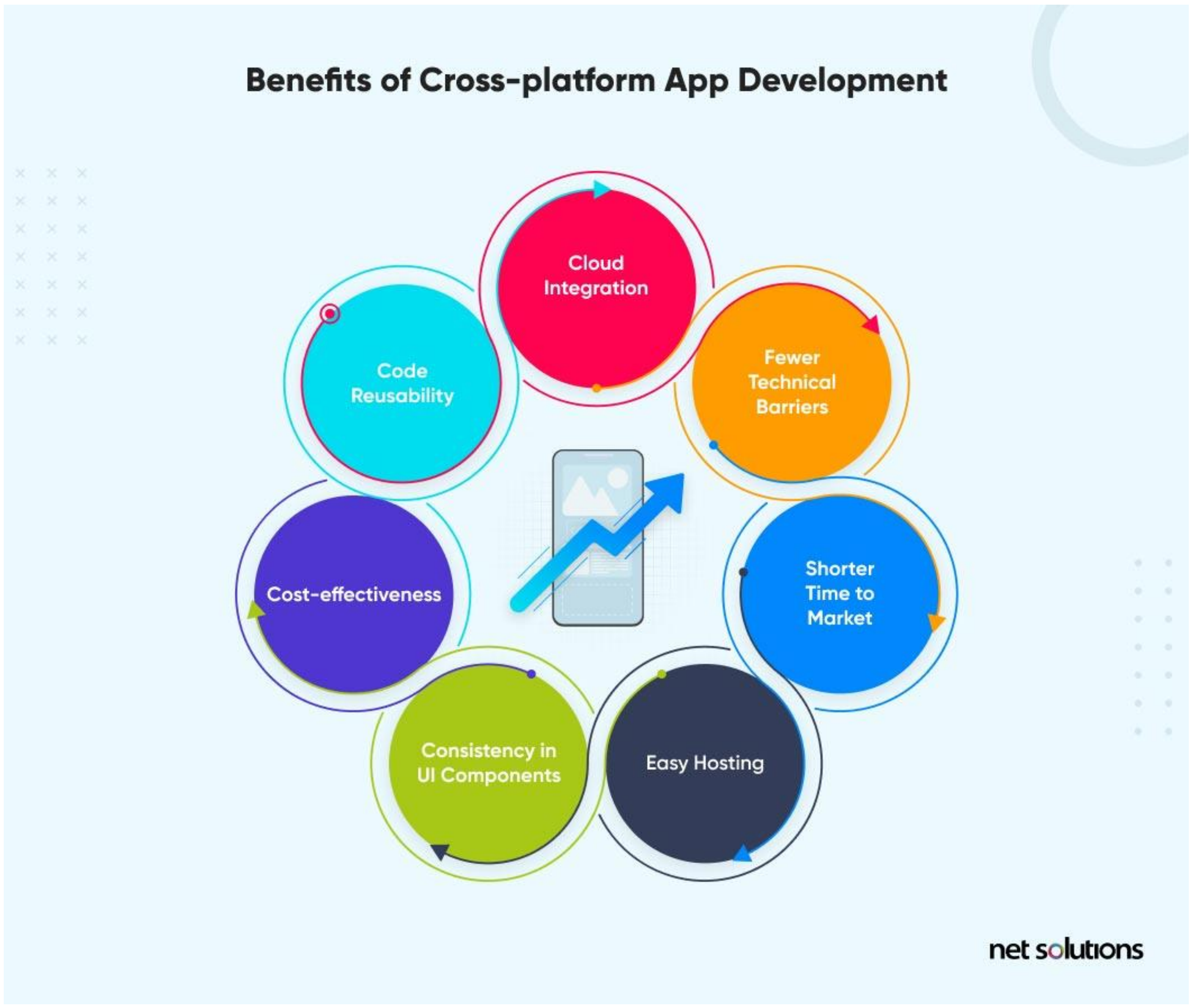
Hybrid mobile apps are among the fastest apps to create across a variety of platforms and easy to update, but are not appropriate for complex, interactive, or feature-rich applications.

The Hybrid architecture typically looks the following way. The ***presentation layer*** has a web-based view. It's built with the help of web technologies such as HTML, CSS, and JavaScript which is powered by the ***rendering engine***. Software developers use PhoneGap, Apache Cordova, or Ionic Capacitor that communicate with the native part via API calls.



Cross Platform Application Architecture

Like hybrid architecture, cross platform development leverages a common codebase with platform-specific abilities in each native shell. Cross platform apps rely on frameworks, rather than web language, including React Native Vs Xamarin. Cross-platform apps offer a User Experience that is a closer approximation to native, often making the approach more attractive.



IMPORTANT FACTORS TO CONSIDER WHEN DEVELOPING MOBILE APP ARCHITECTURE

Mr. Mutiso

1. **Device Type**

When designing a mobile app, first choose the platform (iOS, iPadOS, Android, Windows, Cross-platform), then consider the different models of smartphone that are in use – and there are many! These are important pieces of information to help determine the ideal dimensions for development. The following factors are important to consider when designing a mobile app:

- Screen size and DPI
- Screen resolution
- CPU (processor)
- RAM (memory)

The goal in this stage is trying to deliver the most consistent experience across various platforms and device sizes (tablet vs mobile, model varieties) so that every user – regardless of their device choice – has the best possible experience.

2. **Developmental Frameworks**

Developmental frameworks are a consideration in designing the mobile app architecture as well as in establishing the tech stack. Frameworks provide libraries and basic templates and components for building web apps, both for the front and the back-end.

Front-end frameworks for building mobile apps include Bootstrap, Foundation, React, Angular, Vue, and Backbone. On the ***back-end*** (server-side), developmental frameworks depend on the chosen programming language and target platforms, including Ruby on Rails, Flask, Django, Laravel, Swift, Xamarin, React Native and Flutter – among many others.

3. **Bandwidth Scenarios**

User research is important to understand more about the target user. Around the world, users experience different bandwidth limitations, with some countries on 5G and others still experiencing spotty connections. A highly interactive, graphic-heavy app will not be appropriate for apps that target rural users, for example.

4. User Interface/ User Experience Design (UI/UX Design)

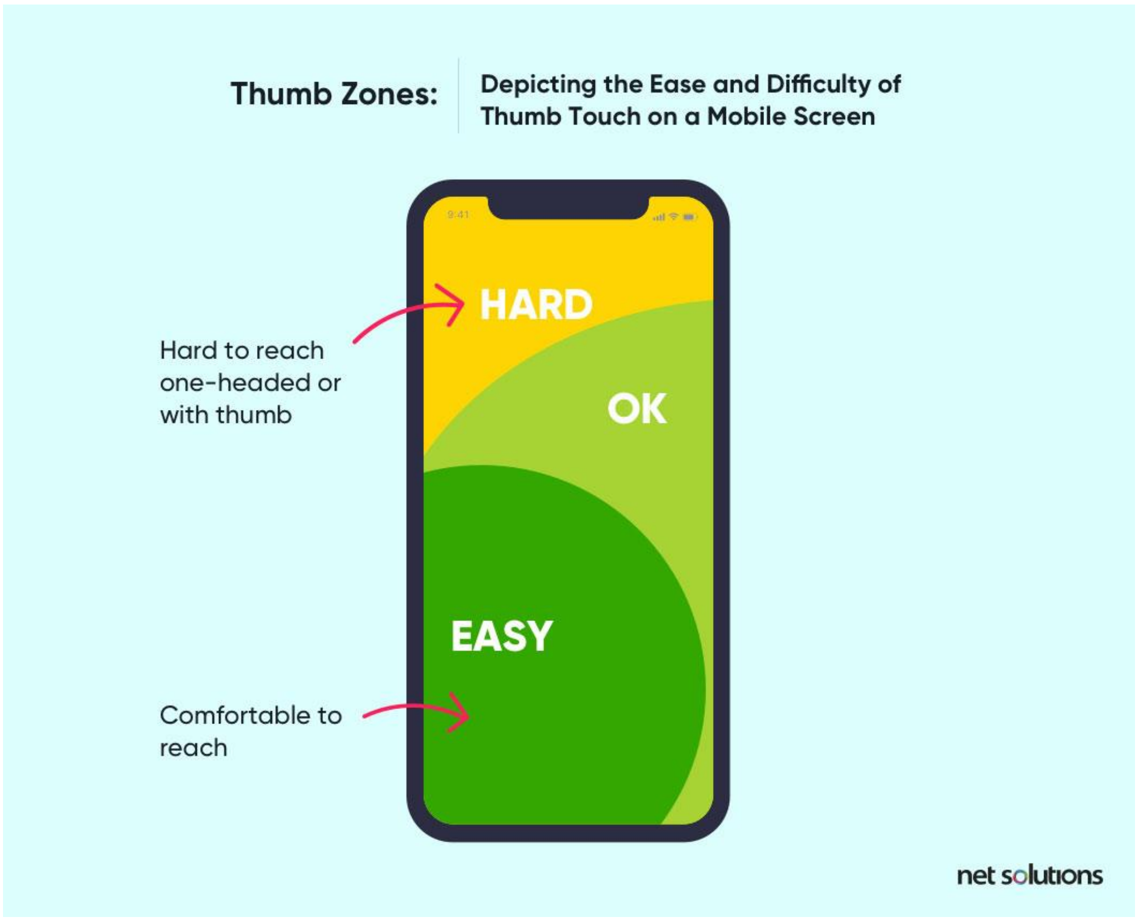
When it comes to mobile app architecture, design plays an important role in first impressions (how it looks – a strong UI) as well as keeping users around (how it works – a strong UX). A strong UX design is critical to online success, but mobile UX is complicated by changing user expectations and best practices of each operating system (OS) and device type.
The mobile app architecture must balance UI against UX in the design phase. Start by understanding the basics of mobile UX design and the latest mobile UX design trends for 2021 to ensure the app is delivering value.

5. Navigation

Navigation is the user's direct contact with the design, impacting both the front-end and the back-end. A great mobile UX design helps users easily identify how to move around the page and explore further sections. ***Familiarity*** is key to navigation. The following navigation best practices help ensure the mobile app is easy to use, reducing friction in the customer journey:

- **Hamburger menu** – the three-line menu in the navigation drawer or top bar is popular with mobile apps because of its familiarity. Enhance navigation with a hover over menu, reducing clicks.
- **Search** – A well-positioned search bar levels up usability, with standard position being the top right.
- **Bars, rails, drawers, or tabs** – There are many options to navigate around an app, including fixed bars of buttons (top and/or bottom), rails (a vertical bar), drawers (hidden navigation), and tabs (screened content with fixed titles).
- **Familiar icons** – familiar icons such as home, search, photos, folder, etc. make navigation easier.
- **Intuitive labeling** – helping distinguish information with labels that spell out the intent of a button, option, or feature.
- **Site organization** – categorization can make site navigation easier – or more complicated – depending on how the categories are named and organized.
- **Gestures** – supporting gesture-based navigation (swipes) can streamline navigation.
- **Scrolling** – how the app supports scroll vs fixed elements.

- **Thumb zone navigation** – Design with the thumb zone in mind for enhanced usability.


Thumb Zones:
Depicting the Ease and Difficulty of Thumb Touch on a Mobile Screen
9:41
HARD
OK
EASY
Hard to reach one-headed or with thumb
Comfortable to reach
net solutions

The best way to ensure a positive user experience (UX) with navigation is through research and user feedback. Wireframes are an important part of early testing of what the app looks like and how it can be navigated.

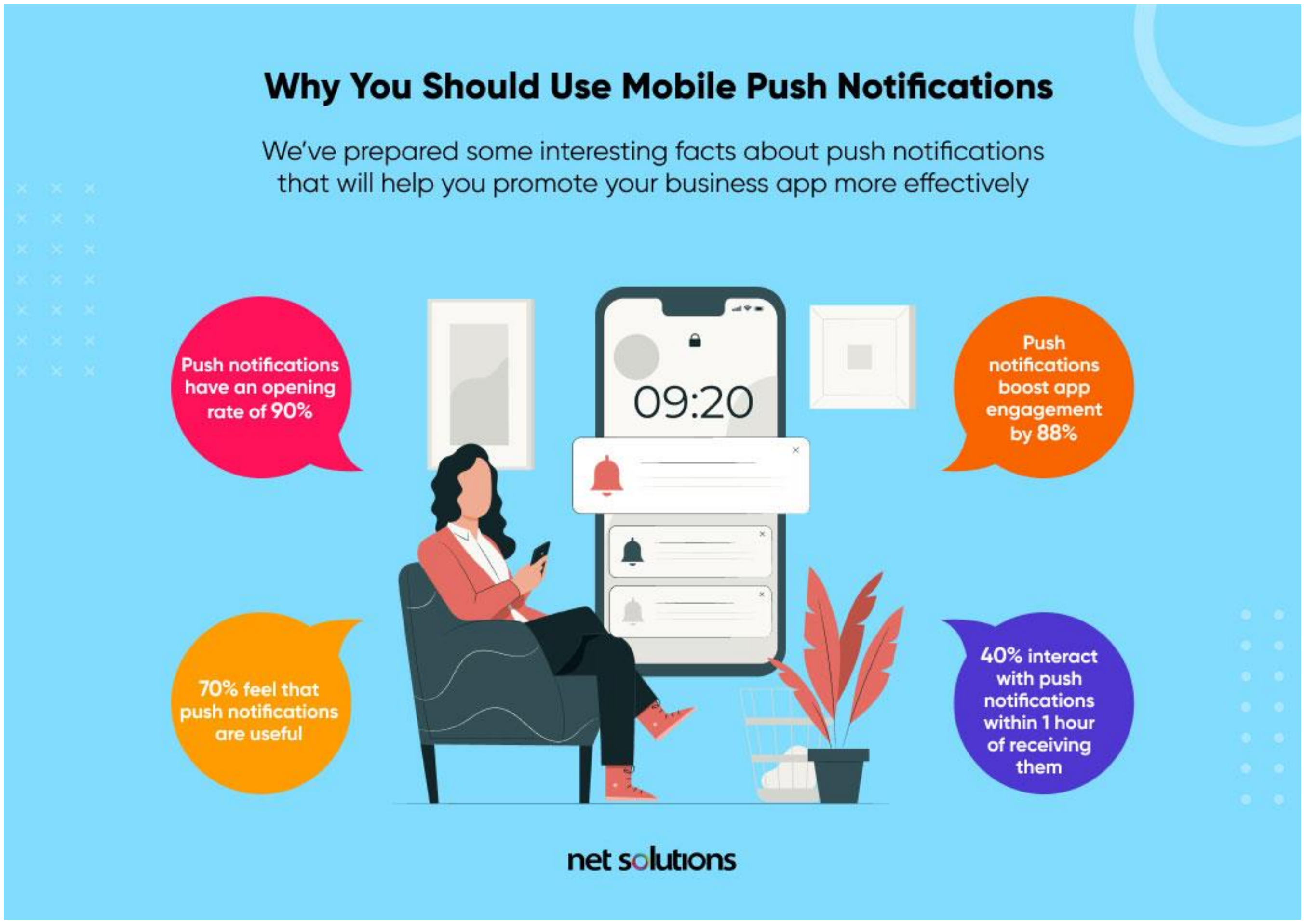
6. Push Notifications vs Real-time Updates

There is a careful balance to be had between nudging users and annoying them when it comes to notification frequency and method. Notification frequency can also have an impact on device battery life, which can in turn impact user retention.

Mobile push notifications are messages that apps send to inform users they have a new in-app message, either from the app itself (such as product updates, offers, or reminders) or from another user. Android users are automatically opted in for push notifications, but iOS users must opt-in, and typically do so at a lower rate (51% iOS vs 81% Android).

While some may consider them annoying, push notifications can be a great marketing tool. Push marketing is most effective when it is personalized or offers a sense of urgency.

Moving past scheduled push notifications, mobile apps are leveraging real-time technologies to add greater interactivity to their apps. For example, *a delivery app can use real-time updates to provide updates about when a food order has been accepted, prepared, picked-up, or tracked on a live map.* Other examples include health and fitness apps or home technologies that benefit from real-time alerts (such as security systems). Like push notifications, real-time updates have their time and place in order to avoid overwhelming users with notifications.


Why You Should Use Mobile Push Notifications
We've prepared some interesting facts about push notifications that will help you promote your business app more effectively
Push notifications have an opening rate of 90%
Push notifications boost app engagement by 88%
09:20
70% feel that push notifications are useful
40% interact with push notifications within 1 hour of receiving them
net solutions

How to Choose the Right Mobile App Architecture

Here are the things to look at when finalizing the design architecture of mobile product:

Budget

The choice of architecture depends on developer skill set, market analysis, and development approach. Unfortunately, the need for rapid, agile development often outpaces availability of IT skills for native and cross-platform developers – time and delay which can make an app obsolete by the time it hits the market.

Audience Analysis

Identify users, their personas (backgrounds, needs, and goals), competitive analysis, agile user stories, flows, mapping, wireframes, and prototyping tests.

Key Feature Requirements

Let the business requirements drive the feature lists, determining whether the app is best suited for Native, Hybrid, or Cross-Platform development.

Platform Choice

The best practices around UI and UX will vary depending on the platform.

Development Time

Some architectures require more time to develop certain elements or integrations, which should be built into the overall plan and choice.

Start Developing A Test App

Agile development focuses on iterative development and continuous feedback, focusing on customer-centric input that can only come from a test app. This very basic app contains basic functionality and navigation and provides valuable feedback as to whether the architecture is working to satisfy the target user.

IMPORTANT TERMINOLOGIES

The Linux kernel is a free and open-source, monolithic, Unix-like operating system kernel. The Linux family of operating systems is based on this kernel and deployed on both traditional computer systems.

Dalvik is a discontinued process virtual machine in Google's Android operating system that executes applications written for Android.

A Software Development Kit (SDK) is typically a set of software development tools that allows the creation of applications for a certain software package, software framework, hardware platform, computer system, video game console, operating system, or similar development platform.

Reference architecture is a document or set of documents to which a project manager or other interested party can refer for best practices.

Native script this has been ranked as the top frameworks for the development of the hybrid apps. Telerik, a Bulgaria based software company, which supports this framework and the creator is Progress. The apps using the Native Scripts utilize the same APIs similar to using Xcode or the Android Studio. You will get sufficient help in the form of tutorials from the official website.

Flutter was launched by Google and is an open-source mobile application development SDK. It is a popular cross-platform app development and is written in the Dart language.

React Native is basically an open-source framework offering ample support to the IDEs and the other mobile app development tools.

An integrated development environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development.

The graphical user interface (GUI) is a form of user interface that allows users to interact with electronic devices through graphical icons and visual indicators such as secondary notation, instead of text-based user interfaces, typed command labels or text navigation.

A **mobile emulator** is a resource for emulating or simulating a mobile device or smartphone environment. It allows developers to test URLs or other technologies on a mobile device's operating system and display interface.

AndroidManifest.xml file in android

The AndroidManifest.xml file contains information of your package, including components of the application such as activities, services, broadcast receivers, content providers etc.

It performs some other tasks also:

- It is responsible to protect the application to access any protected parts by providing the permissions.
- It also declares the android api that the application is going to use.
- It lists the instrumentation classes. The instrumentation classes provides profiling and other informations. These informations are removed just before the application is published etc.

This is the required xml file for all the android application and located inside the root directory.

A simple AndroidManifest.xml file looks like this:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.javatpoint.hello"
  android:versionCode="1"
  android:versionName="1.0" >

  <uses-sdk
    android:minSdkVersion="8"
    android:targetSdkVersion="15" />

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme" >
```

```
        <activity
            android:name=".MainActivity"
            android:label="@string/title_activity_main" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Elements of the AndroidManifest.xml file

The elements used in the above xml file are described below.

<manifest>

manifest is the root element of the AndroidManifest.xml file. It has package attribute that describes the package name of the activity class.

<application>

application is the subelement of the manifest. It includes the namespace declaration. This element contains several subelements that declares the application component such as activity etc.

The commonly used attributes are of this element are icon, label, theme etc.

android:icon represents the icon for all the android application components.

android:label works as the default label for all the application components.

android:theme represents a common theme for all the android activities.

<activity>

activity is the subelement of application and represents an activity that must be defined in the AndroidManifest.xml file. It has many attributes such as label, name, theme, launchMode etc.

android:label represents a label i.e. displayed on the screen.

android:name represents a name for the activity class. It is required attribute.

<intent-filter>

intent-filter is the sub-element of activity that describes the type of intent to which activity, service or broadcast receiver can respond to.

<action>

It adds an action for the intent-filter. The intent-filter must have at least one action element.

<category>

It adds a category name to an intent-filter.

Dalvik Virtual Machine | DVM

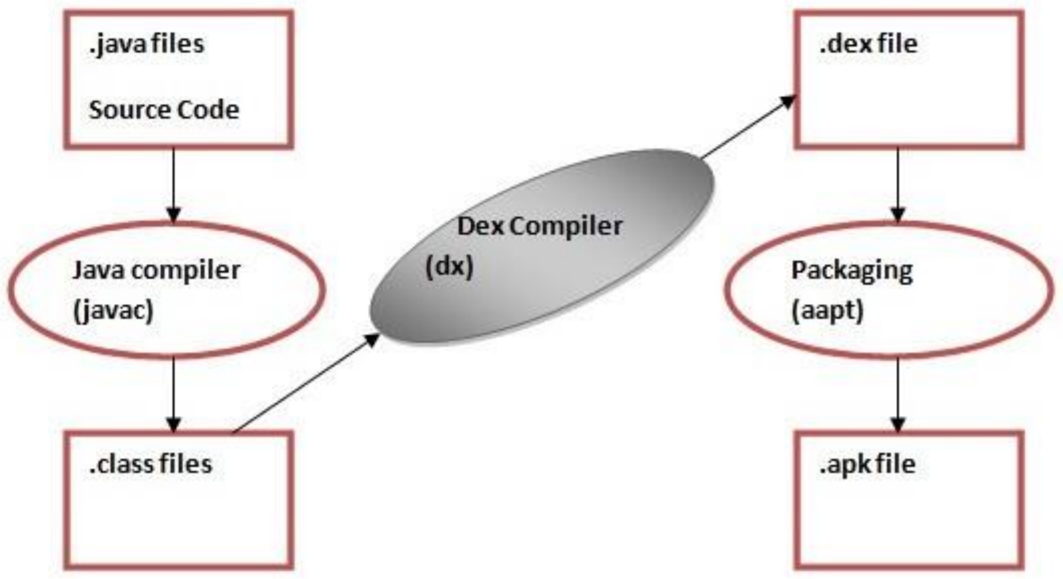
As we know the modern JVM is high performance and provides excellent memory management. But it needs to be optimized for low-powered handheld devices as well.

The **Dalvik Virtual Machine (DVM)** is an android virtual machine optimized for mobile devices. It optimizes the virtual machine for *memory*, *battery life* and *performance*.

Dalvik is a name of a town in Iceland. The Dalvik VM was written by Dan Bornstein.

The **Dex compiler** converts the class files into the .dex file that run on the Dalvik VM. Multiple class files are converted into one dex file.

Let's see the compiling and packaging process from the source file:



The **javac tool** compiles the java source file into the class file.

The **dx tool** takes all the class files of your application and generates a single .dex file. It is a platform-specific tool.

The **Android Assets Packaging Tool (aapt)** handles the packaging process.

What are Virtual Devices?

To put it in a nutshell, a virtual device is a ***software program*** mimicking a smartphone or desktop's abilities and features. This virtual device lets testers run their app and get an understanding of the functioning of an app on real devices. In case of any bugs, testers have the liberty to rectify them then and there. Modern developers leverage virtual devices for websites and app testing in 'near-to-native' environments when the product is developing.

Types of Virtual Devices

Virtual devices have been in practice since the Internet age. Since the 1990s, video game enthusiasts and geeks have found them adorable to the core, when they can emulate and simulate their favorite games and play whenever they want on their PC.

As a matter of fact, the widely accepted virtual devices are of two types:

1. Emulators
2. Simulators

Emulators: An emulator is a piece of software that imitates the behavior of (and in some cases, exact functionality of) another program by either providing an executable version of the other program's machine code or by re-implementing its functionalities.

In a technical sense, emulators are hardware-independent because they can be executed on machines that do not match their native hardware specifications.

Simulators: A simulator is a program or application that attempts to simulate, or mimic, the behavior of a real-world process or system.

Difference between Android Emulator & Simulator

What it tests?

The Android Emulator tests both the hardware and the software whereas the Android Simulator tests only the software. It means you can test the associated parts of the product as well by using an Emulator. But it is not possible with a simulator. You could simply create test cases for the source code related issues and could fix them before the making the product.

When an Emulator or a Simulator is preferable?

Whenever the Android device's external behaviors need a testing, we require a Simulator. For example, we can use an emulator to do mathematical computations, execution of step-by-step transactions, etc. They could be handled with the source code and we are not bothered about the hardware.

At the same time, when we need to test the Android device's internal behavior such as to verify the operation of a hardware part, firmware, etc, we prefer an emulator. The terms might be slightly confusing as 'external' I s being used for the simulator and the term 'internal' is being used for the emulator. When you read it the second time, you would get a clear idea.

In what Language they are written?

We are well-known that any Android device is composed of the Android Operating System (OS) and it obviously requires some source code. Well, the Emulator needs an interaction with the hardware parts of the Android device and therefore, it has to be written in the Machine Language i.e. using zeros and ones. It is often referred as the Assembly Language. But when it comes to the simulator, we are not bothered about the hardware here. So, it could be written in any of the preferable High-Level Languages.

The Debugging Process:

Imagine a situation that you are testing an Android mobile device with an emulator. Whenever you are stuck at a point or if you find an error in the device, then you need to trace back not only the source code but also the associated machine language code of the respective hardware. Only then, you could fix it and could ensure the correctness of the system. This looks easier as you are exposed to both the code right now. But in the case of a simulator, you just need to work on the high-level machine language code. It might sound easier but for your surprise it comparatively a tougher job. Do you why? Here we go! When you are stuck a point here, you could debug just the source code of the software. But when the problem is related to the hardware, it requires a further investigation and you do not have the option in a simulator.

Partial or Complete?

We can consider the Android simulator as a partial implementation of the original device as we miss the hardware testing here. But the Android emulator is a complete implementation of the original device as it includes both the hardware and the software.