REPUBLIC OF CAMEROON **REPUBLIQUE DU CAMEROUN**

PEACE-WORK-FATHERLAND **PAIX-TRAVAIL-PATRIE**

FACULTY OF ENGINEERING AND TECHNOLOGY
DEPARTMENT OF COMPUTER ENGINEERING
COURSE CODE/TITLE: CEF 440/ INTERNET PROGRAMMING AND MOBILE PROGRAMMING
COURSE INSTRUCTOR: DR. NKEMENI VALERY

Group 9

PRESENTED BY :

AMINATU MOHAMMED AWAL	FE22A147
DEFANG MARGARET AKUMAYUK	FE22A185
EKWOGE JUNIOR	FE22A200
ESIMO GODWILL EYABI	FE22A207
SUH AKUMAH TILUI-NTONG	FE22A299

Question 1:

REVIEW AND COMPARE THE MAJOR TYPES OF MOBILE APPS AND THEIR DIFFERENCES (NATIVE, PROGRESSIVE WEB APPS, HYBRID APPS)

Breaking down the major types of mobile apps (Native, Progressive Web Apps (PWAs), and Hybrid Apps), and reviewing their characteristics, comparing their key differences and some examples of each.

1. NATIVE APPS:

- **Definition**: Native apps are built specifically for a single mobile operating system (OS), such as iOS (for iPhones and iPads) or Android (for various Android devices). They are developed using the languages and tools that the OS vendor provides (e.g., Swift/Objective-C for iOS, Java/Kotlin for Android).

- **Characteristics:**

- Performance: Generally offer the best performance and responsiveness. They have direct access to the device's hardware and software capabilities.

- User Experience (UX): Provide the most seamless and integrated UX. They adhere to the OS's design guidelines, resulting in a familiar and intuitive feel.

- Feature Access: Full access to device features, including the camera, GPS, accelerometer, Bluetooth, push notifications, etc.

- Offline Capabilities: Can often function offline or with limited connectivity, depending on how they are designed.

- App Store Distribution: Distributed through the respective app stores (Apple App Store, Google Play Store).

- Security: Typically considered more secure due to the stringent app store review processes and the OS's security features.

- **Advantages:**

- Optimal Performance: Fast and responsive.

- Superior UX: Best possible user experience, blending seamlessly with the OS.

- Full Device Access: Access to all device features.

- Offline Functionality: Can be designed for offline use.

- Enhanced Security: Benefit from OS and app store security measures.

- **Disadvantages:**

- Higher Development Costs: Requires separate development teams and codebases for each platform (iOS and Android).

- Longer Development Time: Development can take longer due to the need for separate codebases.

- Maintenance Overhead: Maintaining two separate apps can be more complex and costly.
- Updates**:** Users need to download and install updates through the app store.

2. **PROGRESSIVE WEB APPS (PWAs):**

- **Definition**: PWAs are web applications that use modern web capabilities to deliver an app-like experience to users. They are built using standard web technologies like HTML, CSS, and JavaScript.

- **Characteristics:**
  - Web-Based: Deployed to a web server and accessed through a URL.
  - Responsive: Adapt to different screen sizes and devices.
  - Installable: Users can "install" PWAs on their home screen, creating an app icon.
  - Offline Capabilities: Can work offline or with unreliable network connections using service workers (JavaScript files that run in the background).
  - Push Notifications: Can send push notifications to users.
  - Discoverable: Search engines can index PWAs, making them easier to find.
  - Secure**:** Should be served over HTTPS to ensure secure data transmission.
  - App-Like Feel**:** Aim to provide a user experience that is similar to a native app.

- **Advantages:**
  - Lower Development Costs: One codebase for all platforms.
  - Faster Development Time: Easier and faster to develop compared to native apps.
  - Easier Maintenance: Only one codebase to maintain.
  - SEO Benefits**:** Search engine discoverability.
  - No App Store Required (Initially): Users can access them directly through a URL. (However, some app stores are now allowing PWAs).
  - Automatic Updates: Updates are deployed to the web server and automatically available to users.

- **Disadvantages:**
  - Limited Device Access: May have limited access to certain device features compared to native apps (this is improving, but still a limitation).
  - Performance Limitations: Performance may not be as optimal as native apps, especially for complex tasks.
  - UX Inconsistencies: User experience may not be as polishedor consistent as native apps.
  - Browser Compatibility: Features and performance may vary across different web browsers.

- Security Concerns (Potentially): While PWAs should be served over HTTPS, they may be more vulnerable to certain web-based attacks if not properly secured.

- Not fully supported by all operating systems IOS support is more limited than Android.

3. **HYBRID APPS:**

- **Definition**: Hybrid apps are essentially web applications (built with HTML, CSS, and JavaScript) wrapped in a native container. They use frameworks like Cordova, Ionic, or React Native to access device features.

- **Characteristics:**

- Cross-Platform Development: One codebase can be used for multiple platforms (iOS and Android).

- Web Technologies: Built using standard web technologies.

- Native Container: Wrapped in a native container that allows access to device features.

- Plugin Architecture: Use plugins to access device features that are not available through standard web APIs.

- App Store Distribution: Distributed through the app stores.

- Performance: Performance is generally better than PWAs but not as good as native apps.

- **Advantages:**

- Lower Development Costs: One codebase for multiple platforms.

- Faster Development Time**:** Faster to develop compared to native apps.

- Access to Device Features: Can access device features through plugins.

- Cross-Platform Compatibility**:** Run on both iOS and Android.

- **Disadvantages:**

- Performance Limitations: Performance may not be as optimal as native apps.

- UX Inconsistencies: User experience may not be as polished or consistent as native apps.

- Plugin Dependencies**:** Rely on plugins for device access, which can be unreliable or outdated.

- Debugging Challenges**:** Debugging can be more challenging due to the hybrid nature of the app.

- Security Considerations: Security depends on the underlying web application and the plugins used.

COMPARISON TABLE

FEATURE	NATIVE APPS	PROGRESSIVE WEB APPs (PWAs)	HYBRID APPs
Technology	iOS (Swift), Android(kotlin)	HTML, CSS, JavaScript, Web APIs	HTML, CSS, JavaScript, Hybrid Frameworks
Performance	Best, optimized for the platform	Moderate, depends on the browser	Moderate, can suffer in resource-heavy tasks
User Experience(UX)	Best, smooth and platform-optimized	Decent, but not as fluid as native apps	Similar to native, but limited by performance
Offline Support	Fully functional offline	Limited offline capabilities	Can work offline if implemented properly
Access to Device Features	Full access to device features	Limited Access to device features	Good access to device feature (depends on framework)
Development Time	Long, requires separate development for each platform	Short, single codebase for all platforms	Moderate, single codebase but requires framework setup
Cost	High	Low	Moderate
App Store Approval	Required (app store/Google play)	No app store approval required	Requires app store approval, similar to native apps

EXAMPLES OF APPLICATIONS

Here are some examples:

A. **NATIVE APPS:**

1. Instagram (iOS, Android): A photo and video sharing app built specifically for each platform.
2. Facebook (iOS, Android): A social media app with native versions for each platform.
3. Twitter (iOS, Android): A microblogging app with native versions for each platform.
4. Uber (iOS, Android): A ride-hailing app built natively for each platform.
5. Snapchat (iOS, Android): A multimedia messaging app built natively for each platform.

B. **PROGRESSIVE WEB APPS (PWAs):**

1. Twitter Lite (Web): A lightweight version of Twitter that provides a native app-like experience.
2. Google Maps (Web): A web-based mapping service that provides a native app-like experience.
3. Facebook Lite (Web): A lightweight version of Facebook that provides a native app-like experience.
4. Pinterest (Web): A web-based discovery and planning website that provides a native app-like experience.
5. Alibaba (Web): A Chinese e-commerce website that provides a native app-like experience.

C. **HYBRID APPS**

1. Ionic Framework (iOS, Android): An open-source framework for building hybrid apps using web technologies.
2. React Native (iOS, Android): A framework for building hybrid apps using React and JavaScript.
3. PhoneGap (iOS, Android): A framework for building hybrid apps using web technologies.
4. Tinder (iOS, Android): A dating app built using a hybrid approach.
5. Walmart (iOS, Android): A retail app built using a hybrid approach.

IN SUMMARY,

- Choose **Native Apps** if: Performance, UX, and access to device features are paramount, and budget is less of a constraint. Critical apps that require the best possible experience.
- Choose **PWAs** if: Speed of development, cost-effectiveness, and SEO are important, and you don't need full access to all device features. Content-heavy apps, e-commerce sites, or marketing campaigns.
- Choose **Hybrid Apps** if: You need a cross-platform solution with access to device features and a balance between cost, development time, and performance. Apps that need to access some device features, but don't require extreme performance.

Question 2:

REVIEW AND COMPARE MOBILE APP PROGRAMMING LANGUAGES

Mobile app programming languages are used to develop applications for mobile platforms like **Android** and **iOS**. Choosing the right language depends on **performance needs**, **platform compatibility**, **development time** and **community support**. Popular Mobile App Programming Languages and their characteristics include:

A. NATIVE DEVELOPMENT LANGUAGES:

These are programming languages that are specifically designed to build applications for a particular platform, operating system or hardware architecture. Examples of native development languages include:

i. SWIFT:

- **Description**: A modern, powerful and intuitive programming language developed by Apple. It is the primary language for building iOS applications. It is designed for safety, performance and modern programming paradigms.

- **Strengths**:
  - Performance: Optimized for Apple's platforms, resulting in fast and responsive apps.
  - Safety: Strong type system and error handling mechanisms reduce runtime crashes.
  - Modern Syntax: Clean, readable and expressive syntax, making development more efficient.
  - Ecosystem Integration: Seamless access to all iOS-specific features, APIs and frameworks.
  - Growing Community: A large and active community with extensive documentation and libraries.

- **Weaknesses:**
  - Platform Specific: Code written in Swift cannot be directly used for Android development.
  - Learning Curve: While designed to be easier than its predecessor Objective-C, it still requires dedicated learning.

- **Use Cases:**

  Primarily used for high-performance, feature-rich iOS applications where optimal user experience and access to platform-specific functionalities are crucial.

ii. **KOTLIN:**

- **Description**: A modern, statically-typed programming language developed by JetBrains. It is the officially preferred language for Android development for Google. Kotlin is designed to be concise, safe and interoperable with Java.

- **Strengths:**
  - Conciseness: Reduces boilerplate code compared to Java, leading to faster development and more readable code.
  - Interoperability with Java: Can seamlessly work with existing Java code and libraries, making migration easier.
  - Strong Community Support: Backed by Google and JetBrains, with a growing and supportive community.
  - Feature-Rich: Supports modern programming paradigms like functional programming and extension functions.

- **Weaknesses:**
  - Platform-Specfic: Code written in Kotlin cannot be directly used for iOS development.
  - Slightly Larger Binary Size: Kotlin apps might have a slightly larger size compared to purely Java-based apps (though often negligible).
  - Learning Curve: While designed to be easier than Java for many, developers familiar with other paradigms might need some adjustment.

- **Use Cases:**

  Primarily for building native Android applications, from simple utilities to complex, high-performance apps. It's also increasingly used for server-side development and Android multiplatform projects.

iii. **OBJECTIVE-C:**

- **Description**: This is the original primary programming language used for developing iOS and macOS applications before Swift. While still supported, it's largely been superseded by Swift for new projects.

- **Strengths:**
  - Mature Ecosystem: A vast number of existing libraries and frameworks are written in Objective-C.
  - Runtime Flexibility: Dynamic runtime allows for more flexibility in certain scenarios.

- **Weaknesses:**

- Verbose Syntax: Can be more complex and less readable compared to Swift.
- Lack of Modern Features: Doesn't have the modern safety features and syntax of Swift.
- Memory Management: Requires manual memory management which can be prone to errors.
- Declining Popularity: Less common for new development.

❖ **Use Cases:**

Primarily for maintaining and updating older iOS and macOS applications.

iv. **JAVA:**

❖ **Description:** The original primary programming language for Android development. While still supported, Kotlin is now the preferred language.

❖ **Strengths**:

- Large Existing Codebase: A massive amount of existing Android code and libraries are written in Java.
- Vast Community and Resources: A very large and established community with extensive documentation.
- Platform Independence: While Android-specific, Java itself is designed to be platform-independent (write once, run anywhere).

❖ **Weaknesses**:

- Verbose Syntax: Can be more verbose compared to Kotlin, leading to more boilerplate code.
- Null Safety Issues: Prone to NullPointerExceptions.
- Less Modern Features: Lacks some of the modern language found in Kotlin.

❖ **Use Cases:**

Primarily for maintaining and updating older Android applications. New Android projects are strongly recommended to use Kotlin.

B. **CROSS-PLATFORM DEVELOPMENT LANGUAGES:**

These languages allow developers to write code and deploy it on multiple platforms (Android & iOS). Examples include:

i. **DART:**

❖ **Description:** Dart is a modern, object-oriented, class-based programming language developed by Google. It is the primary language used to build applications with the Flutter framework.

❖ **Strengths:**

- Performance: Dart's ahead-of-tome (AOT) compilation to native code results in fast and performant applications, often comparable to native apps. It's just-in-time (JIT) compilation during development eanbles rapid iteration with hot-reloading.
- Developer Productivity: The language is designed to be easy to learn and use, with features like expressive syntax, strong typing and excellent tooling.
- Rich Standard Library: Dart comes with a comprehensive set of built-in libraries that cover many common programming tasks.
- Growing Community and Ecosystem: While newer than some other langauges, Dart has a rapidly growing and active community driven by the popularity of Flutter.
- Cross-Platform Focus: Dart is specifically designed to work seamlessly with the Flutter framework to build applications for multiple platforms from a single codebase.

❖ **Weaknesses:**

- Relatively Newer Ecosystem: While growing rapidly, the ecosytem of third-party libraries and tools might not be as extensive as for older, more establisehd languages like Java or JavaScript.
- Primary Association with Flutter: Dart's widespread adoption is largely tied to the Flutter framework. While it can be used for other purposes (like server-side development), it doesn't have the same broad independent usage as some other languages.
- Learning Curve: While designed to be easy, developers coming from purely dynamically-typed languages might have a slight learning curve with Dart's type system.

❖ **Use Cases:**

Used in Mobile App Development, Web Development, Desktop Application Development, Embedded Systems and Server-Side Development.

ii. **React Native:**

❖ **Description:** Based on React, was developed by Facebook and allows for cross-platform mobile development using JavaScript.

❖ **Strengths:**

- Code reusability between web and mobile apps.
- Large developer community and extensive libraries.

❖ **Weaknesses:**

- Performance slightly lower than native apps.

- Requires bridging for some native functionalities.

❖ **Use Case**:
Ideal for startups and companies wanting a single codebase for iOS and Android.

iii. **Xamarin:**

❖ **Description**: Developed by Microsoft and uses C# and .NET to build cross-platform apps.

❖ **Strengths:**
- Deep integration with the .NET ecosystem.
- Access to native APIs.
- Strong performance compared to other cross-platform solutions.

❖ **Weaknesses**:
- Slower updates and smaller community compared to Flutter and React Native.
- UI development can be complex.

❖ **Use Case**: Best for enterprises using Microsoft technologies.

C. **HYBRID DEVELOPMENT LANGUAGES:**

Hybrids apps combine both mobile and web development. They are built using web technologies wrapped in a native shell. Examples include:

i. **Ionic (HTML, CSS, JavaScript):**

❖ **Description:** Ionic is a framework that allows developers to build mobile apps using web technologies.Uses web technologies with frameworks like Angular or React.

❖ **Strengths:**
- Easy to learn for web developers.
- Large library of pre-built UI components.
- Supports multiple platforms.

❖ **Weaknesses:**
- Lower performance compared to native and cross-platform solutions
- Relies heavily on web technologies

❖ **Use Case**: Suitable for apps that prioritize development speed over performance.

ii. **Apache Cordova(HTML, CSS, JavaScript):**

- **Description:** Cordova enables the use of web technologies to create mobile apps with access to native features. It allows web apps to run as mobile apps.

- **Strengths:**
  - Simple for web developers to transition into mobile app development.
  - Large number of plugins available.

- **Weaknesses:**
  - Poor performance for complex applications.
  - UI and UX limitations

- **Use Case**: Best for simple apps with minimal native functionality.

RECOMMENDATIONS:

- If you want the best performance, choose **native development (Swift** for **iOS, Kotlin** for **Android)**.
- If you want a balance of performance and development speed, **Flutter** or **React Native** are great choices.
- If you are a web developer looking to build mobile apps quickly, **Ionic** or **Cordova** may work for you.

Question 3:

REVIEW AND COMPARE MOBILE APP DEVELOPMENT FRAMEWORKS BY COMPARING THEIR KEY FEATURES

Mobile app development frameworks provide a structured environment with pre-built components, tools, and libraries that simplify and accelerate the app development process. They often aim to address the challenges of building for multiple platforms or provide specific advantages in terms of language, performance, cost & time to market, UX & UI, complexity, community support etc.

A. **NATIVE DEVELOPMENT FRAMEWORKS:**

These are software tools and libraries designed for building mobile applications specifically for a single platform (Android or iOS). These frameworks provide direct access to the operating systems's APIs and features, ensuring high performance and a seamless user experience. **Swift** and **Kotlin** are examples of Native Development Frameworks. Below is a detailed comparison of these frameworks:

FEATURE	SWIFT	KOTLIN
Language	Swift	Kotlin
Performance	Optimized for iOS, delivering the best performance on Apple devices.	Optimized for Android, delivering the best performance on Android devices.
Cost & Time to Market	Medium cost, iOS focused	Medium cost, developers take more time to market.
UX & UI	Delivers seamless UI experiences for iOS.	Best for Android devices.
Complexity	Easy for iOS developers.	Complex for beginners.
Community Support	Growing support but limited to iOS developers.	Large Android community
Where it is Used	iOS only apps	Android only apps.

B. **CROSS-PLATFORM FRAMEWORKS:**

Cross-platform frameworks allow developers to write a single codebase that runs on multiple operating systems (Android & iOS). These frameworks save time and cost by eliminating the need to build separate apps for each platform. Examples of Cross-Platform Frameworks and their characterstics include:

FEATURE	FLUTTER	REACT NATIVE	XAMARIN
Programming Language	DART	JavaScript/TypeScript	C#
Performance	High performance	Higher performance but relies on a JavaScript bridge which may cause delays	High performance but with some overhead.
Cost & Time to Market	Faster development with hot reload, reducing time to market.	Quick development due to JavaScript's popularity and large community.	Medium cost but development speed depens on familiarity with C# and .NET
UX & UI	Uses custom UI widgets, providing a consistent look across platforms, expressive UI.	Uses native components, leading to a more natural UX.	Offers native UI but requires platform-specific customization.
Complexity	Medium complexity due to Dart's learning curve	Low complexity if familiar with JavaScript.	Medium complexity, requiring knowledge of .NET and C#.
Community Support	Strong support from Google and a large developer community	One of the most popular frameworks, backed by Facebook.	Supported by Microsoft but has a smaller community.
Where it can be used	Cross-platform mobile apps with high performance and custom UI	Apps requiring native UI and fast development uisng JavaScript.	Used in Enterprise applications with integration into the .NET ecosystem.

C. **HYBRID FRAMEWORKS:**

These frameworks build apps using web technologies (HTML, CSS and JavaScript) wrapped in a native container. These frameworks prioritize ease of web development transition and faster development for simpler apps but may have performance limitations and a less native look and feel. Examples of Hybrid frameworks and their characteristics include:

FEATURE	**IONIC**	**APACHE CORDOVA**
Language	HTML, CSS, JavaScript	HTML, CSS, JavaScript
Performance	Moderate	Moderate
UI & UX	Pre-built UI components	Basic UI customization
Complexity	Very easy to learn and implement.	Very easy learn to implement
Community Support	Large	Medium
Where it can be used	Apps with heavy web UI	Simple hybrid apps

Question 4:

MOBILE APPLICATION ARCHITECTURE AND DESIGN PATTERNS.

What is mobile application architecture?

Mobile app architecture is the blueprint for building applications. The rules, processes, and structure that determines how components like the front-end UI, backend database, APIs, etc, interact to process inputs and give outputs. Think of architecture as the internal skeleton that supports the application's outward function and form.

Importance of a good mobile app architecture.

A well-designed mobile app architecture is crucial to create robust, scalable, and user-friendly applications. It offers several benefits that contribute to app's success like:

- ✓ Firstly, **it enhances modularity**, allowing different app components to be developed and maintained independently, leading to easier updates and modifications.
    - For instance, in an e-commerce app, a modular architecture enables seamless integration of new payment gateways without affecting other functionalities.
- ✓ Secondly, **robust security measures** in the architecture ensure data protection and prevent unauthorized access.
    - A banking app, for instance, can be fortified with encryption protocols and secure authentication methods, instilling trust among users and safeguarding their sensitive information
- ✓ Moreover, **reliability is heightened** as a well-structured architecture minimizes bugs and errors, providing users a seamless experience.
    - Think about a messaging app that rarely crashes or experiences glitches due to its meticulously designed architecture.
- ✓ Furthermore, **performance and scalability are optimized**, allowing the app to handle increasing user loads and adapt to growing demands over time.
    - An example is a social media platform capable of accommodating millions of users concurrently without compromising speed or functionality.
- ✓ Lastly, an excellent mobile app architecture has different layers that structure the application into logical components with distinct roles for **robust and scalable mobile apps**.

Layers of mobile application architecture.

Most mobile app architectures have three layers: presentation, business, and data.

1. **Presentation layer:**

The presentation layer, or front end, is the user interface (UI) you see when opening an app. It includes the screens, navigation, controls, and visual elements.

For example, in a messaging app like WhatsApp, the presentation layer comprises chat screens, contact lists, settings menus, etc. Its primary function is to enable user interactions by taking input from users and displaying output from the lower layers.

2. **Business layer:**

The business layer contains the core application logic that handles tasks like computations, validations, analytics, notifications, background jobs, etc.
In a messaging app, this layer handles functions like sending and receiving messages, encrypting data, detecting spam, managing notifications, etc. It takes input from the presentation and data layers, processes it, and prepares the responses displayed in the UI.

3. **Data layer:**

The data layer handles connections to databases and storage systems, allowing the app to save and retrieve data. For example, in WhatsApp, message data is stored in various databases.

The data layer abstracts the physical storage, so other layers don't need to worry about the specifics of databases. It handles queries, connections, caching, concurrency, and other data access mechanics. The business layer interacts with the data layer by calling methods it exposes.

Types of mobile application architecture.

There are three major types of mobile application architecture including: layered, monolithic, and microservice.

1. **Layered architecture:**

Layered architecture organizes the application into layers, each responsible for a specific aspect of the application's functionality. Typically, these layers include presentation, business logic, and data access layers.

This architecture promotes modularity and separation of concerns, which makes it easier to maintain and scale the application. However, if not implemented carefully, it may lead to tight coupling between layers.

Layered architecture works well for large, complex applications that require frequent updates. By isolating frontend, business, and data layers, you

can focus on specific components and accelerate development and testing cycles for iterative delivery.

2. **Monolithic architecture:**

Monolithic architecture structures the entire application as a single, tightly integrated unit. All application components, including the user interface, business logic, and data access, are packaged together and deployed as a single entity.

While this architecture can simplify development and deployment, it can also lead to scalability and maintainability issues as the application grows in size and complexity.

Monolithic architecture gives you a lightweight, all-in-one bundle for simple apps with well-defined, stable requirements. Since everything is tightly coupled, development and deployment can be fast, especially for apps with limited scope and low chances of change.

3. **Microservice architecture:**

The microservice architecture breaks the application into smaller, independent services, each responsible for specific functionalities. These services communicate through APIs, offering flexibility and scalability.

You can develop, deploy, and scale microservices independently, making updating and maintaining the application easier. However, managing many services can introduce complexity, and additional overhead may be associated with coordinating communication between services.

Microservices architecture excels when you need to update complex apps and scale them across multiple teams frequently. By decomposing into discrete services, you can independently develop, deploy, and scale components to accelerate iteration for large, evolving applications.

Key principles of mobile application architecture

Like pillars supporting a building, principles in mobile architecture provide foundational guidance so you can construct an app that is stable, scalable, and prepared for future growth. Let's explore vital architectural principles to be considered, when designing mobile apps:

Principles	Summary
Flexibility	The architecture can adapt to changing requirements and new technologies
Maintainability	The app is easy to maintain via modularity, loose coupling, and encapsulation
Reusability	Components and modules can be reused across applications

Security	Data and identity are protected through access controls and encryption
Performance	The app delivers speed, reliability, and resource efficiency
Sustainability	The architecture supports continuity over changes and future growth
Extensibility	New capabilities can be added via plugins, extensions, and integrations
Testability	Components can be easily tested in isolation
Intuitiveness	The architecture follows established and familiar patterns
Portability	The app can be deployed on different mobile platforms

Factors to consider while designing a mobile app architecture

Device type, user interface, push notifications, navigation method, etc., are a few of the several factors that help build a robust mobile app architecture.
Let's discuss them briefly:

A. **Device type:**

The device type determines the hardware capabilities and screen sizes your utility needs to support. For example, designing an app for smartphones and tablets requires accommodating different screen resolutions and aspect ratios.

Imagine you're developing a fitness-tracking application. In that case, you should ensure that these small smartphone screens are optimized for large tablet displays for an immersive user experience regardless of the device used.

When evaluating device types for your mobile app architecture, you must consider features like screen size, resolution, memory, processors, battery life, sensors, camera capabilities, and OS versions.

B. **Development framework:**

Selecting the right development framework is vital for efficient application development and maintenance. For example, if you aim to build cross-platform applications, frameworks like React Native or Flutter allow you to write code once and deploy it across multiple platforms.

On the other hand, if you prioritize performance and native experience, platform-specific frameworks like Swift for iOS or Kotlin for Android might be preferable.

C. **Bandwidth:**

The available bandwidth directly impacts how your app communicates with servers and fetches data. Designing your app to minimize data usage becomes paramount in regions with limited internet connectivity. For instance, if creating a video streaming app, you might implement adaptive bit streaming to adjust video quality based on the user's network speed, ensuring smooth playback even in low bandwidth conditions.

D. **Network fluctuations:**

Mobile networks are prone to varying signal strength and stability fluctuations, which can affect app performance. Designing your app to handle network fluctuations gracefully is essential for providing a seamless user experience. Implementing caching mechanisms to store frequently accessed data locally can reduce reliance on real-time network requests and mitigate the impact of temporary network outages.

E. **User interface:**

The user interface (UI) plays a crucial role in shaping the overall user experience of your app. It should be intuitive, visually appealing, and consistent across different devices and screen sizes. For instance, if you are designing a shopping app, you should prioritize easy navigation, intuitive search functionality, and visually appealing product displays to enhance the shopping experience for users.

F. Navigation method:

The navigation method defines how users navigate your app and access its features and content. Whether you opt for tab-based navigation, drawer menus, or bottom navigation bars depends on the complexity of your app and user preferences. For example, a news app might use tab-based navigation to allow users to quickly switch between different sections like top stories, sports, and entertainment.

g. **Real-time updates vs. push notifications:**

Deciding between real-time updates and push notifications depends on the nature of your app and the importance of timely information delivery. For example, a messaging app requires real-time updates to ensure instant user communication. In contrast, a news app might use push notifications to notify users of breaking news stories or personalized updates based on their interests.

Mobile app architecture patterns

Mobile app architecture patterns serve as blueprints that define the structure, behavior, and interaction of various components within an app.

Here are the four major patterns:

1. **Model-View-Controller (MVC):**

MVC separates an app into three interconnected components: Model, View, and Controller. The Model manages your data and app logic; the View displays the UI, and the Controller handles user input and interactions.
This separation of concerns allows for better code organization, reusability, and testability. However, it can lead to massive View Controllers and tight coupling between components, which can make maintenance challenging as the app grows.

Using MVC is better when you need to manage complex user interfaces and want to organize your codebase in a structured and maintainable manner.

2. **Model-View-Presenter (MVP):**

MVP addresses some of the limitations of MVC, particularly concerning testability and separation of concerns. In MVP, the View only shows data and captures user input. The Presenter acts as an intermediary between the View and the Model, handling all the logic related to user interactions and data manipulation. This separation makes it easier to test the Presenter without involving the Android framework or UI components.

MVP promotes cleaner code and improved maintainability by reducing the dependencies between different app layers.

MVP architecture is better for faster development and easier maintenance because it promotes a clear separation of concerns and facilitates iterative development cycles.

3. **Model-View-ViewModel (MVVM):**

MVVM, originating from Microsoft and widely used in Android app development, introduces architectural components like LiveData and ViewModel. In MVVM, the ViewModel acts as a link between the View and the underlying data sources.

Unlike MVP, MVVM enables data binding, where changes in the ViewModel are automatically reflected in the View without manual updates. This enhances the separation of concerns, improves code readability, and supports reactive UIs.

MVVM is a good choice for apps that require a structured and modular design, especially those with complex UIs and interactions.

4. **View-Interactor-Presenter-Entity-Router (VIPER):**

VIPER stresses modularity, scalability, and testability by dividing an app into layers: View, Interactor, Presenter, Entity, and Router. Each layer has a specific responsibility, such as handling user interactions, business logic, data manipulation, and navigation.

VIPER enforces firm boundaries between components, reducing coupling and making it easier to replace or modify individual modules without

affecting the rest of the app. While VIPER offers excellent separation of concerns and scalability, it introduces more complexity upfront, requiring careful planning and implementation.

You should prefer VIPER architecture when you want a clear separation of concerns in your iOS apps, especially in complex projects where scalability and maintainability are crucial.

Example of modern mobile application architectures

A few examples of modern mobile app architecture include Android, iOS, hybrid, and enterprise.

1. ANDROID MOBILE APP ARCHITECTURE:

Android mobile app architecture typically follows the MVVM pattern, leveraging frameworks like Jetpack for robust mobile app development. Components like LiveData and ViewModel help manage lifecycle awareness and data persistence, ensuring a responsive and scalable app.

2. iOS MOBILE APP ARCHITECTURE:

iOS mobile app architecture often revolves around the MVC or MVVM design patterns.

Apple's SwiftUI framework has gained popularity for its declarative syntax and real-time previews for rapid development. With SwiftUI, you can create responsive and intuitive user interfaces while adhering to modern design principles. Additionally, the Combine framework facilitates reactive programming, enabling seamless data flow and event handling in iOS apps.

3. HYBRID MOBILE APP ARCHITECTURE:

Hybrid mobile app architecture combines elements of web and native app development, allowing you to build cross-platform apps using web technologies like HTML, CSS, and JavaScript. While hybrid apps offer broad platform compatibility and faster time-to-market than web apps, they may face performance limitations and restricted access to native APIs.

4. ENTERPRISE MOBILE APP ARCHITECTURE:

Enterprise mobile app architecture addresses the unique requirements of large-scale business applications, emphasizing security, scalability, and integration with existing systems.

Enterprise mobility platforms like SAP Mobile Platform, IBM MobileFirst, and Oracle Mobile Cloud provide tools and services for building, deploying, and managing enterprise-grade mobile apps with security features such as data encryption, authentication, and role-based access control at the forefront.

Choosing the right mobile application architecture

Selecting the right mobile architecture is crucial because it impacts user experience, performance, scalability, security, and the overall success of the

app. To level up your app to enterprise-grade, you need to opt for the right architecture, which comes with challenges such as balancing client-server communication and offline support, determining the optimal granularity for different components, and managing the complexity of transitioning to microservices

Question 5:

HOW TO COLLECT AND ANALYZE USER REQUIREMENTS FOR A MOBILE APPLICATION

A. **COLLECTING REQUIREMENTS**

1. DEFINE OBJECTIVES:

Identify the goals of the app and the problem it aims to solve and also define the target audience for more tailored insights

2. CONDUCT STAKEHOLDER INTERVIEWS:

Identify stakeholders like users, clients or team members and ask them open ended questions to gather insights

3. SURVEYS AND QUESTIONAIRES:

Design a survey covering key aspects of the app like features and usability and distribute it online using tools like Google forms and monkey survey to reach a broader audience

4. FOCUS GROUPS:

Organize sessions with small groups of target users and facilitate discussions to encourage open sharing of thoughts

5. USER PERSONA:

Create fictional characters representing different types of users focusing on demographics, goals, behaviors and pain points

6. USER JOURNEY MAPPING:

map out the steps the users take when interacting with the app and identify critical touch points and analyze pain points

7. COMPETITIVE ANALYSIS:

Research similar apps to identify successful features and feedback and benchmark against competitors to find gaps and opportunities

8. PROTOTYPING:

Develop low fidelity wire frames to visualize the layout and the flow and share prototypes with users for initial reactions

ANALYZING REQUIREMENTS

1.REQUIREMENT DOCUMENTATION

Compile findings into a comprehensive document and include both functional and non-functional requirements

2. PRIORITIZE REQUIREMENTS

Use prioritization techniques like the MOSCOW METHOD to distinguish between important features from less important and useless ones. It includes the must have features, should have features, could have features and the would have features. Categorizing the requirements under these would help clarify what is needed and what is not.

3. REVIEW AND VALIDATE:

Present the requirements documents to stakeholders for feedback and revise the requirements based on the stake holder input. Ensure that it meets the user's needs and expectations.

4. COUNTINUOUS FEEDBACK

Establish feedback loops for ongoing user input throughout development and be flexible to adapt to requirements based on user testing and feed back

TOOLS

1. JIRA: A tool for issue tracking project management and requirement management
2. Trello : A visual tool for organizing and prioritizing requirements
3. Survey monkey and Google Forms: Tools for creating and sharing surveys
4. UserTesting: A tool used to collect feedback from users in the form of video recordings and written comments
5. MOSCOW METHOD: Technique for prioritizing requirements
6. Use case diagrams: Tool used to show what each participant can do in the app etc.

Question 6:

STUDY HOW TO ESTIMATE MOBILE APP DEVELOPMENT COST.

INTRODUCTION

Estimating mobile app development cost is crucial to ensure projects are completed within budget and on time. The following steps below can be use to obtain this estimated cost.

1. **Define Project Requirements**

A. **Identify the app's purpose, features, and functionality**.

- Complexity of features: More complex features, such as payment gateways or social media integration, increase development time and cost.
- Number of features: More features require more development time, increasing costs. E.g Facebook and Go student.
- Determine the platforms (iOS, Android, or both).

B. **Consider the technology stack, architecture, and infrastructure**.

- Choice of programming languages, frameworks, and libraries: Some technologies, such as React Native, may be more cost-effective than others, like native iOS and Android development.
- Architecture and infrastructure: Complex architectures or infrastructure requirements, such as backend servers or databases, increase development time and cost.

2. **Choose a Development Methodology**

A. **Waterfall**: Linear approach with fixed requirements.

- Higher upfront costs due to detailed planning and documentation.
- Less flexibility, leading to potential rework and cost overruns.

B. **Agile**: Iterative approach with flexible requirements.

- Lower upfront costs due to iterative development and flexible requirements.
- More flexibility, reducing rework and cost overruns.

C. **Hybrid**: Combination of Waterfall and Agile.

- Estimated cost: Moderate (between Waterfall and Agile).

3. **Estimate Development Time**

A. **Break down the project into smaller tasks.**

- Designing the user interface.
- Developing the backend API.
- Implementing payment gateway integration.

B. **Estimate the time required for each task considering:**

- Complexity of the task: The more complex the task is, the more time is allocated to it.
- Development team's experience and efficiency: Critical tasks are allocated to the development team with more experience and efficiency, in order to have the best result within a significant time.

4. **Calculate Development Costs**

A. **Determine the hourly rate of the development team.**

- **Definition**: The hourly rate of a development team refers to the amount of money charged by the team for one hour of their work.
- **Factors affecting the hourly rate:**
  - Experience and Expertise: More experienced and skilled developers typically charge higher hourly rates.
  - Technology Stack: Developers with expertise in specific technologies, such as Native iOS/ Android may charge higher than React Native or Flutter, due to their high demand.

B. **Multiply the estimated development time by the hourly rate.**

C. **Add additional costs (Design and prototyping, testing, Deployment and maintenance**, **Marketing and promotion).**

5. **Consider Additional Costs**

A. **Design and prototyping.**

- **UX/UI Design:** Estimate the hourly rate for user experience (UX) and user interface (UI) design and multiply by the estimated time for Design
  - Hourly rate: 500 FCFA – 1,500 FCFA per hour
  - Estimated time: 40-100 hours
  - Total cost: 20,000 FCFA - 150,000 FCFA

- **Prototyping**: Estimate the number of hours required for creating a functional prototype and multiply by the hourly rate of the expertise.
  - Hourly rate: 500 FCFA - 1500 FCFA per hour
  - Estimated time: 20-50 hours
  - Total cost: 10,000 FCFA -75,000 FCFA

B. **Testing and quality assurance.**

- **Manual Testing**: Estimate the number of hours required for manual testing, and multiply by the hourly rate of the expertise.
  - Hourly rate: 200 FCFA – 500 FCFA per hour
  - Estimated time: 40-100 hours
  - Total cost: 8,000 FCFA – 50,000 FCFA
- **Automated Testing**: Estimate the number of hours required for automated testing, and multiply by the hourly rate of the expertise.

C. **Deployment and maintenance**.

- **Deployment**: Estimate the cost of deploying the app to app stores or play store.
- **Ongoing Maintenance and Updates**: Estimate the cost of ongoing maintenance and updates.
  - Cost: 1,000 FCFA - 5,000 FCFA per month

D. **Marketing and promotion.**

- **App Store Optimization (ASO):** Estimate the cost of ASO services.
  - Cost: 100,000 FCFA -200,000 FCFA
- **Social Media Marketing**: Estimate the cost of social media marketing services.
  - Cost: 1,000 FCFA -5,000 FCFA per month
- **Influencer Marketing**: Estimate the cost of influencer marketing services.
  - Cost: 1,000 FCFA-10,000 FCFA per month

TOTAL ADDITIONAL COST

The estimated costs for design and prototyping, testing and quality assurance, deployment and maintenance, and marketing and promotion are all added to get the **total additional cost.**

CONCLUSION:

By following these steps, you can create a comprehensive and accurate estimate for mobile app development costs.