MVI Architecture in Android

Overview

MVI (Model-View-Intent) is an architectural pattern for Android development that promotes a unidirectional data flow and improves the separation of concerns. The Model represents the state of the application, View displays the UI and sends user interactions as Intents, and the Intent represents user actions or events. The View observes the state changes from the Model and renders the UI accordingly. Intents are dispatched from the View to update the Model. This pattern ensures a predictable and testable codebase by enforcing immutability and isolating side effects. MVI helps in building scalable, maintainable, and robust Android applications with a clear separation of concerns.

Introduction

In the world of Android development, creating scalable and maintainable applications is a top priority. To achieve this, software engineers rely on well-designed architectural patterns that promote clean code and separation of concerns. One such pattern gaining popularity is the Model-View-Intent (MVI) architecture.

The MVI architecture follows a clear separation of concerns by dividing the application into three main

components: Model, View, and Intent.

The Model component in MVI represents the state of the application. It encapsulates the data and business logic, ensuring that it remains immutable. The Model component is responsible for managing the application state and providing it to the View for rendering.

The View component in MVI is responsible for rendering the user interface (UI) based on the current state received from the Model. It observes the state changes and updates the UI accordingly. The View is also responsible for capturing user interactions, which are then transformed into Intents and sent to the Model.

The Intent component represents the user actions or events within the application. It encapsulates the user's intention, such as button clicks, text input, or other UI interactions. Intents are dispatched from the View to the Model, triggering state updates or other operations.

What is MVI architecture?

MVI (Model-View-Intent) is an architectural pattern specifically designed for building Android applications. It follows a unidirectional data flow approach, emphasizing the separation of concerns and immutability to create scalable, maintainable, and testable code.

At its core, the MVI architecture consists of three main components: Model, View, and Intent. Each component

has distinct responsibilities and contributes to the overall structure and behavior of the application.

Model

The Model component represents the state of the application. It encapsulates the data and business logic necessary for the app's functionality. The Model is typically implemented as an immutable data structure or an immutable object that can be updated via state transformations. Immutability ensures predictability and eliminates unexpected side effects. Any changes to the Model result in a new state being created, maintaining the integrity of the previous state. The Model component is responsible for managing the application's state and providing it to the other components.

View:

The View component is responsible for rendering the user interface (UI) and displaying the application's state to the user. It observes changes in the Model's state and updates the UI accordingly. The View component should ideally be passive, without any business logic or state management. In MVI, the View is typically implemented as a passive observer that receives state updates from the Model. When the Model's state changes, the View re-renders the UI based on the new state. The View component also captures user interactions, such as button clicks or text input, and transforms them into

Intents.

Intent:

The Intent component represents the user's intention or action within the application. It encapsulates the user interactions captured by the View and serves as a signal to the Model for state updates or other operations. Intents are dispatched from the View to the Model and trigger specific actions based on the user's actions. Intents in MVI are not the same as Android's Intent class used for inter-component communication. Instead, they represent user actions or events within the application, such as requesting data, submitting a form, or navigating to a different screen. The Model reacts to these Intents, performs the necessary operations, and updates the state accordingly.

How does the MVI work?

The MVI (Model-View-Intent) architecture follows a unidirectional data flow approach, which provides a clear and predictable flow of data within an Android application. Understanding how MVI works involves exploring the interactions between its three main components: Model, View, and Intent.

- **Model:** The Model component in MVI represents the state of the application. It encapsulates the data and business logic necessary for the application's

functionality. The Model is typically implemented as an immutable data structure or an immutable object. The Model is responsible for managing the application state. It receives Intents from the View and performs operations based on those Intents. These operations may include fetching data from a remote server, processing user input, or updating the state based on specific business rules. It ensures that the state transitions are consistent and predictable.

- **View:** The View component in MVI is responsible for rendering the user interface (UI) and displaying the current state to the user. It observes changes in the Model's state and updates the UI accordingly. The View is typically implemented as a passive observer, without any business logic or state management. The View receives the current state from the Model and uses that information to render the UI. When the state changes, the View re-renders the UI to reflect the updated state. This ensures that the UI is always in sync with the application's state.

- **Intent**: The Intent component in MVI represents the user's intention or action within the application. It encapsulates the user interactions captured by the View and serves as a signal to the Model for state updates or other operations. Intents are dispatched from the View to the Model. When the Model receives an Intent, it processes the Intent and produces a new state based on the current state and

the nature of the Intent. The new state is then emitted by the Model, notifying the view that a state change has occurred. The View observes this state change and re-renders the UI to reflect the updated state.

Advantages of MVI

- **Predictability:** The unidirectional data flow in MVI ensures a predictable flow of data within the application. Changes to the state are driven by Intent, and the state is updated in a controlled manner. This predictability makes it easier to understand and reason about the application's behavior.
- **Separation of Concerns:** MVI promotes a clear separation of concerns between the Model, View, and Intent components. Each component has distinct responsibilities, making the codebase modular and maintainable. The Model handles state management, the View focuses on UI rendering, and the Intent represents user actions. This separation facilitates easier development, debugging, and testing.
- **Testability:** MVI architecture enhances testability by decoupling the UI interactions from business logic and state management. The unidirectional data flow allows for comprehensive unit tests and UI tests since the state changes can be easily predicted and verified. With immutability in the Model, testing becomes more reliable, as it prevents unexpected

modifications to the state.

- **Scalability**: MVI architecture supports scalability by breaking down the application into smaller, modular components. Each component has well-defined responsibilities and can be developed and maintained independently. Adding new features or modifying existing ones becomes more manageable as the codebase remains structured and organized.
- **Maintainability**: With a clear separation of concerns and a unidirectional data flow, MVI makes the codebase more maintainable. Developers can easily understand and modify the application's behavior since the data flow is confined to a single direction. This architectural pattern also facilitates code readability and reduces code duplication, making maintenance tasks less error-prone.
- **Reactive Programming:** MVI aligns well with reactive programming concepts and libraries such as RxJava or Kotlin Coroutines. Reactive programming helps handle asynchronous operations, manage streams of data, and react to state changes efficiently. By leveraging reactive programming, MVI architecture enables developers to build responsive and reactive applications.
- **Consistency in User Experience:** MVI focuses on managing the application's state and updating the UI accordingly. With the unidirectional data flow, the UI always reflects the current state, providing a consistent user experience. Users can interact with

the application, knowing that the UI will accurately reflect the changes in real-time.

- **Debugging and Troubleshooting**: The unidirectional data flow simplifies debugging and troubleshooting processes. Since the data flow is deterministic and predictable, developers can track and trace the flow of data, making it easier to identify the source of issues. The separation of concerns also helps isolate problems to specific components, speeding up the debugging process.

Disadvantages of MVI

While MVI (Model-View-Intent) architecture offers several advantages, it is important to consider potential disadvantages that developers may encounter when implementing this architectural pattern. Here are some of the possible drawbacks of MVI:

- **Learning Curve**: MVI introduces a different way of thinking about application architecture compared to traditional patterns like MVC or MVP. This can result in a learning curve for developers who are new to MVI or reactive programming concepts. Understanding the unidirectional data flow and the separation of concerns requires time and effort to grasp fully.
- **Increased Complexity**: MVI adds a layer of complexity to the application architecture. The unidirectional data flow, along with reactive programming concepts, may introduce additional

complexity in terms of code structure, data transformations, and event handling. This complexity can make the codebase harder to understand, especially for developers unfamiliar with reactive programming.

- **Boilerplate Code**: MVI often requires writing additional code to handle the unidirectional data flow, event handling, and state management. This can lead to increased boilerplate code, which may be seen as redundant or unnecessary. Writing and maintaining this additional code can be time-consuming and can potentially increase the risk of introducing bugs.
- **Performance Overhead**: Depending on the implementation and the size of the application, MVI can introduce performance overhead. The unidirectional data flow and the reactive nature of MVI may involve additional data transformations, event handling, and state updates, which can impact the overall performance of the application. Careful consideration should be given to optimizing performance-critical sections of the code.
- **Reactive Programming Complexity**: MVI often goes hand-in-hand with reactive programming libraries such as RxJava or Kotlin Coroutines. While these libraries offer powerful tools for handling asynchronous operations and managing data streams, they can introduce additional complexity. Developers need to understand and become proficient in reactive programming concepts, which

can be challenging for those who are new to this paradigm.

- **Codebase Size**: MVI architecture may result in a larger codebase compared to simpler architectural patterns like MVC or MVP. The additional layers, data transformations, and event-handling mechanisms can increase the overall size of the codebase. This can potentially impact build times, code readability, and maintenance efforts, especially in larger projects.
- **Initial Development Time:** Implementing MVI from scratch may require additional development time compared to using more familiar patterns. Developers need to design the unidirectional data flow, establish communication between components, and handle state updates and UI rendering. This initial development time investment should be considered, especially for smaller projects or tight deadlines.
- **Project Suitability**: While MVI can be beneficial for many Android applications, it may not be the best fit for every project. Simple applications with limited state management requirements may find the additional complexity of MVI unnecessary. It is important to evaluate the specific needs of the project and consider whether the benefits of MVI outweigh its potential drawbacks.
- **Tooling and Community Support**: MVI is a relatively newer architectural pattern compared to more established patterns like MVC or MVP. As a result, the

availability of tools, libraries, and community support specific to MVI may be more limited. This can pose challenges in finding ready-made solutions or seeking guidance when encountering issues.

Creating a Project with MVI Architecture

Creating a project with MVI (Model-View-Intent) architecture involves several steps to set up the necessary components and establish the unidirectional data flow. Here's a step-by-step guide to creating a project with MVI architecture:

Step 1: Project Setup Start by setting up a new Android project using your preferred IDE. Ensure that you have the necessary tools and dependencies installed, such as Android Studio and the Android SDK.

```
// Added Dependencies
implementation "androidx.recyclerview:recyclerview:1.1.0"
implementation 'android.arch.lifecycle:extensions:1.1.1'
implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.2.0'
implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.2.0'
implementation 'com.github.bumptech.glide:glide:4.11.0'

//retrofit
implementation 'com.squareup.retrofit2:retrofit:2.8.1'
implementation "com.squareup.retrofit2:converter-moshi:2.6.2"

//Coroutine
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.6"
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.6"
```

Step 2: Define Project Structure Define the project structure based on the separation of concerns in the MVI architecture. Typically, you'll have separate packages for the Model, View, and Intent components. Additionally, you may have packages for data sources, repositories, and other relevant modules.

Step 3: Model Component Create the Model component, which represents the state of the application. The Model can be implemented as an immutable data class or an immutable object. Define the necessary properties and methods to represent the state and

perform state transformations.

Step 4: View Component Implement the View component responsible for rendering the UI and observing the Model's state changes. The View should be passive and not contain any business logic. Define the UI elements and layouts as per your design requirements.

The View should observe the changes in the Model's state and update the UI accordingly. This can be done using reactive programming libraries such as RxJava or Kotlin Coroutines. Subscribe to state updates from the Model and update the UI based on the received state.

Step 5: Intent Component Create the Intent component, which represents the user's intentions or actions within the application. Intents capture user interactions such as button clicks, text input, or other UI events. Intents are dispatched from the View to trigger specific actions in the Model.

Step 6: Implement Data Sources and Repositories
Create data source classes or repositories to handle data fetching and manipulation. These classes abstract the data layer and provide methods to retrieve data from remote APIs, local databases, or other sources.

Step 7: Implement the Unidirectional Data Flow
Establish the unidirectional data flow by defining the interactions between the Model, View, and Intent components. The flow typically follows these steps:

- The View captures user interactions and transforms them into Intents.
- The View dispatches the Intents to the Model.
- The Model processes the Intents, performs necessary operations (e.g., data fetching or state updates), and produces a new state.
- The Model emits the new state, which is observed by the View.
- The View updates the UI based on the new state.
- Ensure that the data flow remains unidirectional, with changes flowing from the View to the Model and back to the View for UI updates. This ensures a predictable and maintainable architecture.

Step 8: Handle Asynchronous Operations `Handle asynchronous operations, such as data fetching or API calls, within the Model. Use reactive programming libraries like RxJava or Kotlin Coroutines to handle asynchronous tasks in a structured and efficient manner.

Reactive programming allows you to handle streams of data and asynchronous events, ensuring that the application remains responsive and reactive to user interactions.

Step 9: Unit Testing Write unit tests to verify the functionality of individual components. Test the Model's state transformations, View's UI updates, and Intent handling to ensure that the application behaves as expected.

Unit tests play a crucial role in maintaining the integrity of the architecture and validating the correctness of the implementation.

Step 10: Integration Testing Perform integration testing to verify the interactions between the Model, View, and Intent components. Test the unidirectional data flow, state updates, and UI rendering to ensure that the components

work together seamlessly.

Step 11: Iterative Development and Refinement
Continue the iterative development process, refining the implementation based on feedback, testing, and user requirements. MVI architecture allows for incremental changes and modular development, making it easier to adapt and refine the application over time.

Conclusion

- MVI (Model-View-Intent) architecture promotes a predictable and structured approach to Android app development.
- The Model component represents the state of the application and handles state management and transformations.
- The View component is responsible for rendering the UI and observing the changes in the Model's state.
- The Intent component captures user actions and triggers specific actions within the Model.
- Implementing the unidirectional data flow ensures that changes in the state flow from the View to the Model and back to the View for UI updates.
- Reactive programming libraries like RxJava or Kotlin Coroutines can be leveraged to handle asynchronous operations and manage data streams efficiently and it also enhances testability by decoupling UI interactions from business logic and providing a clear way to predict and verify state changes.

- MVI architecture promotes maintainability by providing a clear separation of concerns, reducing code duplication, and facilitating code readability.
- While MVI offers advantages such as predictability, separation of concerns, and scalability, it also has considerations such as a learning curve, increased complexity, and potential performance overhead.