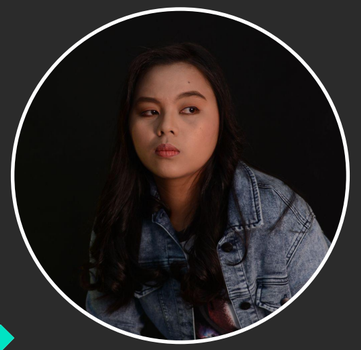
droidcon academy

Guide to Managing States in Android Jetpack Compose



Loreleen Mae “Yoye” Sablot

@yoyesablot

in/loreleensablot

Course Prerequisites

- Knowledgeable in Kotlin Syntax
- Basic knowledge of Compose

droidcon academy

Course Overview

- Introduce State and MutableState in Jetpack Compose
- Apply State Hoisting to Create Reusable Components
- Integrate LiveData and ViewModel for robust state handling
- Handle asynchronous operations within state management
- Apply techniques to preserve app state across restarts and configurations changes
- Explore navigation scenarios and their implications for state management
- Implement UI design patterns to efficiently represent state
- Diagnose and debug common state management issues
- Optimize state management for performance and consistency

droidcon academy

This course is a four-video series

1. Getting Started

2. Mastering the Essentials: State Management in Jetpack Compose

3. Expanding State Management Techniques in Jetpack Compose

4. State Management Troubleshooting in Jetpack Compose

droidcon academy

Getting Started

droidcon academy

Import the starter project

droidcon academy

Dependencies Overview

droidcon academy

Summary

- Demonstration of the final project
- Importing the Starter Project
- Dependencies Overview

droidcon academy

Mastering the Essentials: State Management in Jetpack Compose

droidcon academy

Overview

- Role of state in UI development
- Explore State and MutableState.
- Introduction and benefits of state hoisting.
- Understand Unidirectional Data Flow (UDF).
- Integration of LiveData and ViewModel in Compose.

The concept of State in UI Development

What is State?

State is the current condition or context of an app at any given moment, controlled by the data it has and how users interact with it, which in turn decides what you see on the screen.

droidcon academy

Role of State

- User Interface (UI) State Management
- Data Persistence
- Lifecycle Awareness
- Reactivity and LiveData
- Handling Configuration Changes
- Memory Management
- Jetpack Components for State Management

droidcon academy

Contrast with Traditional UI Development

Traditional Android Views (Imperative Approach)	Jetpack Compose (Declarative Approach)
Manual UI Updates	Automatic UI Updates
View Hierarchy Management	Simplified Hierarchy
Lifecycle Dependency	Lifecycle-Aware
State Management	State Hoisting
Boilerplate Code	Reduced Boilerplate
Imperative Programming	Declarative Programming
	Recomposition
	Integrated State Management Tools

Embracing Modern UI with Jetpack Compose

droidcon academy

Embracing Modern UI with Jetpack Compose

What is Jetpack Compose?

Jetpack Compose is a modern, Kotlin-based toolkit for building native Android user interfaces. It simplifies and streamlines UI development by using a declarative approach, allowing developers to describe what their UI should look like, rather than focusing on the process of building it. This results in less boilerplate code and a more intuitive development experience.

Declarative UIs: How Jetpack Compose simplifies UI development by allowing developers to describe UIs in terms of state

Jetpack Compose simplifies UI development through its declarative approach, where developers describe UIs in terms of state.

- State-Driven Design
- Declarative Syntax
- Automatic UI Refresh
- Reduced Boilerplate
- Composable Functions
- Improved Maintainability
- Intuitive State Management

droidcon academy

How Compose automatically updates the UI when the underlying state changes

Reactivity in Jetpack Compose refers to its ability to automatically update the UI in response to changes in the underlying state. This feature is central to the declarative nature of Compose and greatly simplifies the development of dynamic user interfaces.

- State Observability
- Declarative UI Definitions
- Automatic Recompositions
- Efficient Updates
- Unidirectional Data Flow
- Declarative Nature

droidcon academy

Implementing Basic State Management

droidcon academy

State<T>

- State<T> is an interface that holds a value of type T and allows composable functions to observe this value. When the value changes, it triggers a recomposition of the composables that use it.
- It is immutable.
- It's typically used when you want to display a value in your UI and have the UI update automatically when the value changes.

MutableState<T>

- MutableState<T> is a subclass of State<T> that allows for mutability. It provides a way to both read and write the value.
- It has a value property for getting and setting the current state. Setting this value will automatically notify the Compose runtime to recompose any affected composables.
- It's used when you want to manage a piece of state that can change over time, especially in response to user interactions or other events.

How They Work in Compose

- Recomposition
- Creating State
- Lifecyle Awareness

droidcon academy

The lifecycle of state objects in Compose

droidcon academy

How state drives recomposition in UI elements

State Observability in Compose
- State objects are observable.
- Changes in state notify the Compose runtime.

Reactive UI Updates
- Composables automatically listen for state changes.
- State changes trigger recomposition of related UI elements.

Selective Recomposition Process
- Only parts of the UI affected by the state change are recomposed.
- Efficient rendering: minimizes unnecessary UI updates.

Reinvoking Composables
- Composable functions are reinvoked with new state values.
- Results in an updated UI reflecting current state.

Codelab #1: State and Mutable State

droidcon academy

Stateful and Stateless

droidcon academy

Stateful Composables

Stateful composables are those that manage their own state internally. They create and maintain state within themselves.

Characteristics:

- **Own State Management**
- **Self-contained**
- **Less Reusable**

Usage: Ideal for components with self-contained behavior and no need to share their state with other parts of the app.

droidcon academy

Stateless Composables

Stateless composables do not hold any state within themselves. Instead, they rely on external sources for their state.

Characteristics:

- **State Hoisting**
- **Reusability**
- **Simpler Testing**

Usage: Suitable for creating generic, reusable UI components that can adapt to different states.

Comparison and Best Practices

Separation of Concerns:
Stateless composables promote separation of concerns by separating UI rendering from state logic.

Flexibility in Architecture:
Using stateless composables allows for more flexible and maintainable UI architecture.

Combination Use: Often, stateful and stateless composables are used together. Stateful composables can manage state and pass it down to stateless composables for rendering.

Codelab #2: Stateful and Stateless Composable in WeatherApp

droidcon academy

State Hoisting Concepts

droidcon academy

State Hoisting

What is State Hoisting?

- Moving state up to a higher level in the composable hierarchy.
- State is passed down to composables as parameters.

droidcon academy

Advantages of State Hoisting

Enhanced Reusability
- Makes composables more adaptable to different contexts.
- Encourages creating flexible UI components.

Improved Testability
- Easier to test stateless composables.
- Simplifies unit testing by isolating state from UI logic.

Better State Management
- Encourages clear separation of concerns.
- Facilitates centralized state management.

Increased Control Over State
- Provides more control over how and when state changes.
- Helps in managing side-effects and dependencies.

Promotes Unidirectional Data Flow
- Aligns with modern UI architecture patterns.
- Ensures a more predictable and consistent data flow.

Codelab #3: State Hoisting

Unidirectional Data Flow in Compose

Unidirectional Data Flow

UDF, or Unidirectional Data Flow, is a design pattern commonly used in modern application development, including in frameworks like Jetpack Compose for Android. It refers to a pattern where data follows a single direction through the system, making the data flow predictable and manageable.

Data Flow Sequence:

1. **Action** A user interaction or event triggers an action (like a button click).
2. **State Update:** This action updates the application's state (often managed in a ViewModel or similar state holder).
3. **UI Update:** The updated state then flows down to the UI components, which reactively update to reflect the new state.

Importance of Unidirectional Data Flow

Predictability: Simplifies tracking data changes and understanding state effects on the UI.

Maintainability: Facilitates debugging and testing by clarifying state change processes.

Consistency: Ensures all UI parts are updated uniformly, avoiding outdated displays.

Simplified State Management: Structures complex state changes, aiding in clearer state handling.

Enhanced Scalability: Suitable for growing applications, supports modular development.

Alignment with Modern Architecture: Complements patterns like MVVM, promoting clean and efficient code.

droidcon academy

UDF's Contribution to Predictable State Management

Single Source of Truth: Centralizes state management, reducing conflicts and discrepancies.

Clear Data Path: Establishes a transparent and logical flow of data, easing traceability.

Reduced Side Effects: Minimizes unexpected behaviors by isolating state changes.

Streamlined Debugging: Simplifies identification and resolution of state-related issues.

Consistent State Updates: Ensures uniform application of state changes across the UI.

Benefits of UDF in Complex UI Scenarios

Simplified Complexity: Manages intricate UI interactions in a more structured manner.

Enhanced Modularity: Facilitates breaking down the UI into manageable, independent components.

Improved Scalability: Adapts well to expanding features and growing application size.

Easier Maintenance: Streamlines updates and maintenance of complex UIs.

Robust Testing: Supports more effective testing strategies for intricate UI elements and interactions.

Code Lab #4: Implementing Unidirectional Data Flow

droidcon academy

LiveData and ViewModel Integration

droidcon academy

LiveData in State Management

LiveData is an observable data holder class from the Android Architecture Components. It's lifecycle-aware, meaning it respects the lifecycle of other app components, such as activities, fragments, or services.

Usage in State Management:

- **Reactive UI Updates:** LiveData can be observed for changes, allowing the UI to react and update automatically when the underlying data changes.

- **Lifecycle Awareness:** Automatically manages UI updates based on the component lifecycle, preventing memory leaks.

ViewModel in State Management

ViewModel is a class that's designed to store and manage UI-related data in a lifecycle-conscious way. It allows data to survive configuration changes such as screen rotations.

Role in Maintaining UI State:

- **Data Persistence:** Maintains data during configuration changes, preventing data loss.
- **Separation of Concerns:** Separates UI data from the Activity or Fragment, reducing the bloat in these components and making them more manageable.
- **Centralized Data Management:** Acts as a communication center between the repository (data layer) and the UI.

Benefits of Integrating LiveData with Jetpack Compose

- Seamless UI Updates
- Reduced Boilerplate
- Improved Lifecycle Management
- Easy Data Sharing
- Enhanced Robustness

droidcon academy

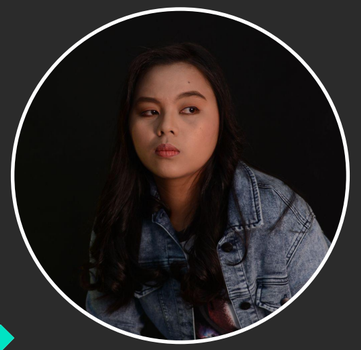
Summary

- Embarked on Understanding State in Jetpack Compose
- Learned Basic State Management by using of State and MutableState
- Explored State Hoisting Concepts
- Applied Unidirectional Flow in our project
- Integrated LiveData and ViewModel

droidcon academy

droidcon academy

Guide to Managing States in Android Jetpack Compose



Loreleen Mae “Yoye” Sablot

@yoyesablot

in/loreleensablot

Expanding State Management Techniques in Jetpack Compose

droidcon academy

Overview

- Explore techniques for ensuring **state restoration** and **persistence** in Jetpack Compose apps. Learn about **rememberSavable** and other tools for preserving state across app restarts and configuration changes.
- Dive into managing state during navigation between composables/screens.
- Various UI state design patterns (Loading, Content, Error) and their significance. Learn best practices for implementing these patterns in a user interface.

Mastering State Restoration and Persistence

Ensuring state restoration and persistence in Jetpack Compose apps is crucial for a seamless user experience, especially across configuration changes like screen rotations and process death.

droidcon academy

rememberSaveable

Functionality: **rememberSaveable** is used to automatically save and restore the state during configuration changes or process death. It works similarly to remember, but with persistence capabilities.

Usage: Ideal for simple data types like String, Int, Boolean, and custom types that are Parcelable or Serializable.

Limitations: It's not suitable for large or complex data structures due to memory and performance constraints.

ViewModel

State Preservation: ViewModels are designed to hold and manage UI-related data in a lifecycle-conscious way, surviving configuration changes.

Usage with LiveData or StateFlow: Combine ViewModel with LiveData or StateFlow for a reactive UI that responds to data changes.

Process Death Handling: For handling process death, additional persistence mechanisms (like a database or SharedPreferences) might be needed.

DataStore and SharedPreferences

Persistent Storage: Jetpack DataStore or SharedPreferences can be used for lightweight persistence needs, like user preferences.

Advantages: They provide a simple key-value storage solution and are easy to implement.

DataStore Over SharedPreferences: DataStore is recommended over SharedPreferences for better performance and more robust features.

Room Database

Local Database: For more complex data, Room provides a robust solution to persist data in a local SQLite database.

Integration with Compose: Use Room with LiveData or Flow to automatically update the UI based on database changes.

Additional Techniques

SavedInstanceState: In traditional Android development, SavedInstanceState is used for saving state. In Compose, you might still use it in combination with composables for complex scenarios.

Handling Asynchronous Operations: Manage loading and error states effectively when dealing with asynchronous operations like network requests or database operations.

UI State Separation: Keep UI state separate from business logic for cleaner code and easier state management.

Testing: Regularly test state preservation across configuration changes and app restarts to ensure reliability.

Codelab #1: Updating WeatherScreen, WeatherViewModel, WeatherCard and WeatherApp


LET'S GET STARTED!

Implementing State in Navigation with Jetpack Compose

Navigation in Jetpack Compose revolves around the use of NavHost and NavController, which are part of the Android Jetpack Navigation component. This system provides a way to navigate between different composables in an application.

NavHost

- **Functionality**: NavHost is a composable function that displays different composables based on the current navigation state. It acts as a container for different navigation destinations in your app.

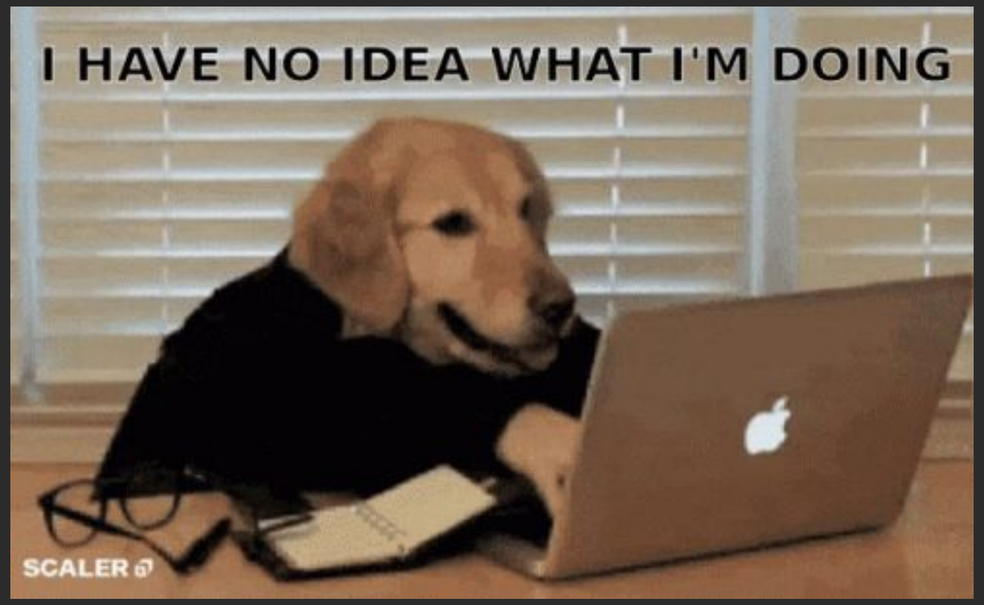
- **Usage**: You define a NavHost in your main composable, and it hosts the different screens or destinations of your app.

Implementing State in Navigation with Jetpack Compose

NavController

- **Functionality**: NavController manages the app navigation within a NavHost. It's responsible for swapping content in the NavHost as users navigate through the app.

- **Usage**: Use NavController to navigate between composables, passing data between them, and handling deep links.



Codelab #2
Implementing State in Navigation

UI State Design Pattern

UI state design patterns are essential for handling different states of an application in a user-friendly way. The most common patterns are Loading, Content, and Error states.

1. **Loading State**

    - **Purpose**: Indicates data is being fetched or processed.
    - **UI Elements:** Progress indicators like spinners, progress bars.
    - **User Experience:** Keeps users informed that an operation is in progress, preventing confusion.

UI State Design Pattern

2. Content State

- **Purpose**: Displays the main content of the application.
- **UI Elements:** Text, images, lists, and other primary content.
- **User Experience:** Core of the app's functionality, where users interact with the app's main features.

UI State Design Pattern

3. Error State

- **Purpose**: Shows when an error occurs (e.g., network issues, data fetch failure).
- **UI Elements:** Error messages, retry buttons.
- **User Experience:** Informs users of issues, often with an option to retry or receive further instructions.

UI State Design Pattern

Best Practices for State Management

- **Smooth Transitions:** Ensure seamless transitions between states for a better user experience.
- **Clear Indications:** Make sure each state is clearly indicated, so users are never left guessing.
- **Accessibility:** Consider accessibility in each state (e.g., readable error messages, visible loading indicators).
- **Consistency:** Maintain consistent design across different states for user familiarity.

Coding Challenge

Instructions for Coding Challenge:

1. Loading State:

- When the app is launched or a new city is selected, display a loading state in the `WeatherScreen` while fetching weather data.
- Use a loading indicator (e.g., CircularProgressIndicator) to indicate that data is being fetched.
- Implement the loading state in the `WeatherScreen`.

Instructions for Coding Challenge:

2. Success State:

- After fetching weather data successfully, display the weather information in the `WeatherScreen`.
- Show the current temperature, description, and a list of short-term weather forecasts.
- Use the `WeatherCard` composable to display each forecast item.
- Implement the success state in the `WeatherScreen`.

droidcon academy

Instructions for Coding Challenge:

3. Error State:

- Handle the case where there is an error while fetching weather data.
- Display an error message indicating that something went wrong along with a retry button.
- When the retry button is clicked, attempt to fetch weather data again.
- Implement the error state in the `WeatherScreen`.


6

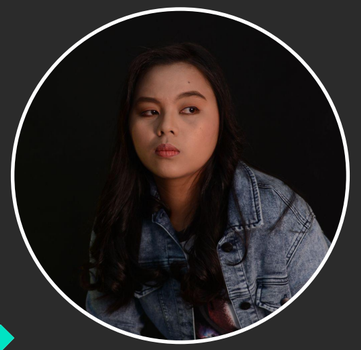
Summary

- Know how to do state restoration and persistence techniques in Jetpack Compose.
- Explored how to manage state while navigating between different screens in apps.
- Improved user interfaces by implementing responsive UI state design patterns.

droidcon academy

droidcon academy

Guide to Managing States in Android Jetpack Compose



Loreleen Mae “Yoye” Sablot

@yoyesablot

in/loreleensablot

Troubleshooting State Management in Jetpack Compose

droidcon academy

Overview

- Identify Common State Management Pitfalls
- Debugging Techniques for State Changes
- State Optimization for Enhanced Performance
- Ensuring State Synchronization and Consistency

droidcon academy

Diagnosing State Management Issues

In Jetpack Compose, managing state effectively is key to building responsive and robust applications. However, several common issues can arise with state management. Here are some of it:

```
@Composable
fun Counter() {
    // Issue: Using mutable state excessively for simple UI updates
    var count by remember { mutableStateOf(0) }

    Button(onClick = { count++ }) {
        Text("Count is $count")
    }
}
```

Overuse of Mutable State

- **Issue**: Excessive use of MutableState<T> can lead to complex and hard-to-maintain code.
- **Impact**: Can cause unnecessary recompositions and performance issues.

```
@Composable
fun UserProfile(userId: String) {
    // Issue: State should be hoisted but is managed within the composable
    var userProfile by remember { mutableStateOf<UserProfile?>(null) }

    LaunchedEffect(userId) {
        userProfile = fetchUserProfile(userId)
    }

    // ... UI logic
}
```

Incorrect State Hoisting

- **Issue:** Improper hoisting of state (not lifting it enough or too much).
- **Impact:** Leads to tightly coupled components, affecting reusability and testability.

```
@Composable
fun UserData(userId: String) {
 // Issue: Not handling loading and error states effectively
    val userData by produceState<Resource<UserData>>(initialValue = Resource.Loading()) {
        value = try {
            Resource.Success(fetchUserData(userId))
        } catch (e: Exception) {
            Resource.Error(e)
        }
    }

    // ... UI logic
}
```

Managing Asynchronous Data

- **Issue:** Challenges in handling loading, success, and error states effectively.
- **Impact:** Can result in poor user experience and unhandled errors.

droidcon academy

```
@Composable
fun MessageList(userId: String) {
    // Issue: Potential inconsistency with UI updates on state changes
    for (message in messages) {
        Text(message)
    }
}
```

State and UI Consistency

- **Issue:** Ensuring UI consistency with state, especially across configuration changes.
- **Impact:** Inconsistencies can confuse users and lead to incorrect app behavior.

droidcon academy

```
@Composable
fun Dashboard(userProfile: UserProfile, settings: Settings) {
    // Issue: Observing more state than necessary
     val theme by settings.theme.observeAsState()

     // ... UI logic that might not need to observe 'theme'
}
```

Unnecessary State Observations

- **Issue:** Observing more state than necessary in composables.
- **Impact:** Triggers unwanted recompositions, impacting performance.

droidcon academy

```
@Composable
fun UserInput() {
    // Issue: Should use rememberSaveable for input state preservation
    var input by remember { mutableStateOf("") }

    TextField(value = input, onValueChange = { input = it })
}
```

Improper Use of remember and rememberSaveable

- **Issue:** Confusion over when to use remember vs. rememberSaveable.
- **Impact:** Incorrect usage can lead to loss of state on configuration changes.

droidcon academy

Diagnosing State Management Issues

Diagnosing state-related problems in modern application development, especially in frameworks like Jetpack Compose, requires a combination of techniques and tools. Understanding and resolving these issues is crucial for maintaining a robust and efficient application. Here's an overview of various techniques and tools you can use:

1. Logging and Debugging

- **Logging**: Use logging extensively to trace state changes. Libraries like Timber or Android's built-in logging can be helpful.
- **Debugging**: Utilize the debugging tools in your IDE (like Android Studio) to step through code and inspect state at runtime.

Diagnosing State Management Issues

2. Jetpack Compose Specific Tools

- **Compose Preview**: Leverage Compose's Preview feature to visualize composables in different states without needing to run the entire app.
- **Layout Inspector**: Use the Layout Inspector in Android Studio to examine the UI and its associated properties, which can help identify issues with state-driven UI changes.

3. Unit and Integration Testing

- **JUnit Tests:** Write unit tests to ensure that your state management logic behaves as expected.
- **Espresso or Compose Test:** For integration testing, use Espresso (for traditional Android UI) or the Compose Testing library to simulate user interactions and verify UI states.

Diagnosing State Management Issues

4. State Flow Analysis

- **Flow Visualization Tools:** For apps using Kotlin Flow or LiveData, tools that visualize the flow of data can help in diagnosing issues. Android Studio Profiler can be used for LiveData.
- **Code Review:** Regularly reviewing code that involves state changes can prevent and identify potential issues early.

5. Performance Profiling

- **Android Profiler:** Use Android Studio's Profiler to monitor the app's performance, which can indicate issues like excessive recompositions in Compose or memory leaks.
- **Memory Profiler:** Specifically look for memory leaks which might be a result of improper state management.

Diagnosing State Management Issues

6. Threading and Coroutine Analysis

- **Coroutine Debugger:** In apps using Kotlin Coroutines, use the Coroutine Debugger in Android Studio to trace and understand asynchronous state changes.
- **Thread Analysis:** Ensure that state changes are happening on the correct threads to avoid concurrency issues.

7. Community and Resources

- **Online Forums:** Platforms like Stack Overflow, Reddit, and Kotlin Slack can be valuable resources for seeking advice on specific state-related problems.
- **Documentation and Guides:** Refer to official documentation and community-written guides or blog posts for best practices and common pitfalls.

Diagnosing State Management Issues

8. Lint Tools and Static Analysis

- **Kotlin Lint Tools:** Use linting tools to analyze your code for common mistakes and anti-patterns in state management.
- **Static Analysis Tools:** Tools like SonarQube or Android Lint can help identify potential issues in your codebase.

Strategies for Tracking and Debugging State Changes

1. **Logging:**
   Implement detailed logging at state change points.
   Utilize frameworks like Timber for enhanced logging capabilities.

2. **State Change Listeners:**
   Set up listeners or observers on state variables to log or breakpoint on changes.

3. **Breakpoints:**
   Use breakpoints in IDE to halt execution at critical state change points.

4. **Unit Tests:**
   Write tests to validate state transitions and logic.

Tools and Methods for Effective State Debugging in Compose

1. **Compose Preview:**
   Utilize to visualize different states without full app execution.

2. **Layout Inspector:**
   Examine the UI and properties in different states.

3. **Compose Testing Library:**
   Write tests for composables to check state-driven UI changes.

4. **Android Studio Profiler:**
   Monitor app performance for clues about state management issues.

Codelab #1
Implementing Debugging Tools

droidcon academy

1. Logging Implementation

droidcon academy

2. Compose Preview Usage

droidcon academy

3. Breakpoints in ViewModel

droidcon academy

Techniques to Optimize State Management for Improved App Performance

Minimize State Usage:
Reduce unnecessary mutable states to avoid excessive recompositions.

Efficient State Partitioning:
Break complex state into smaller, focused states for targeted UI updates.

Lazy State Initialization:
Initialize state lazily to reduce initial load time.

Avoid Deep Observables:
Prevent deeply nested observables; they can lead to complex renders.

State Hoisting and Sharing:
Hoist state to the right level and share across composables to prevent duplication.

Identifying and Addressing Performance Bottlenecks Related to State

Profiler Use:
Utilize Android Profiler to identify performance issues related to state changes.

Recomposition Analysis:
Use tools like Layout Inspector to analyze unnecessary recompositions.

Memory Leak Detection:
Detect and fix memory leaks that may be caused by improper state handling.

Benchmarking State Changes:
Benchmark critical paths in your app where state changes occur.

Strategies to Maintain State Consistency

Single Source of Truth:
Centralize state management to maintain a consistent state across the app.

Immutable State Objects:
Use immutable objects for state to prevent unintended modifications.

State Hoisting:
Hoist state to higher-level composables to facilitate sharing and consistency.

Observer Pattern:
Implement observer pattern to notify components of state changes.

Synchronization Mechanisms:
Use tools like LiveData or StateFlow for automatic UI updates on state changes.

Handling Challenges in State Synchronization

Concurrent State Modifications:
Address potential race conditions in state updates.

Complex State Structures:
Simplify complex state structures to ease synchronization.

Lifecycle Management:
Ensure state consistency across different lifecycle stages of components.

Asynchronous Operations:
Handle timing issues in asynchronous data updates.

Summary

- **Identify Pitfalls:** Recognized common state management mistakes and pitfalls.
- **Debug State Changes:** Learned strategies for tracking and debugging state changes.
- **Handle Complexity:** Managed complex and nested states in Compose applications.
- **Optimized for Performance:** Implemented state optimization for enhanced app performance.
- **Ensure Consistency:** Ensured state synchronization and consistency across app components.

droidcon academy

Course Summary

- Explored State Management Fundamentals: Covered the basics of state management in Jetpack Compose, including State and MutableState.
- Applied State Hoisting Techniques: Learners applied state hoisting to create reusable components
- Integrated LiveData and ViewModel: Integrated LiveData and ViewModel for enhanced state handling
- Managed State During Navigation: Handled state effectively during screen transitions in Compose apps

droidcon academy

Course Summary

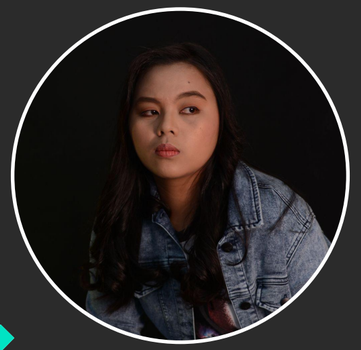
- Restore and Persisted State: Preserved app state across restarts and configuration changes
- Implemented UI State Design Patterns: Employed various UI design patterns for state representation
- Troubleshoot Common State Issues: Diagnosed and debugged prevalent state management problems
- Optimized State for Performance: Enhanced state management for improved performance and consistency

droidcon academy

droidcon academy

droidcon academy

Guide to Managing States in Android Jetpack Compose



Loreleen Mae “Yoye” Sablot

@yoyesablot

in/loreleensablot