JetWeatherApplication:

Parts and procedure to make it through:

Prompt: @project "so tell me the procedure that i am trying to follow to build this app, i mean like what would be the chronology of the app, how does the app is structured, like what component is made first and then followed in sequence. What approach is im trying to take, to follow to implement, also tell me does my approach is good to do like as compared to future aspects and modern techniques, if not what should I change, or alter in my approach, and which approach is recommended and must be adhered to do so."

OR

Here's a refined, comprehensive prompt that combines all the previous details:

"I'm seeking a detailed explanation of the development process for an application from the ground up. Could you provide a comprehensive breakdown that covers:

1. **Initial Development Steps:**
   * What are the very first components or features a developer should build to create a basic, working version of the app?
   * How should these initial components be structured or organized to serve as a solid foundation?
2. **Sequential Component Development:**
   * What should be the next steps after establishing the basic app framework?
   * In what order should components be developed and integrated (for example, starting with core functionalities, then backend services, followed by user interface enhancements, etc.)?
   * How do these components build on one another throughout the development process?
3. **Development Chronology and Structure:**
   * Outline the ideal sequence of building an application—from initial planning and architectural design to component development, integration, and testing.
   * What is the recommended chronological order for these phases, and how does each phase contribute to the overall structure and functionality of the app?
4. **Methodologies and Best Practices:**
   * Describe the overall approach and methodology for the codebase, including any recommended design patterns, frameworks, or architectures to ensure scalability and maintainability.
   * How does this sequential approach compare with modern and emerging development techniques?
   * What are the pros and cons, and are there any adjustments or alternative strategies you would recommend to improve the process and future-proof the codebase?

This detailed prompt is intended to provide insights into the exact order and method a developer might follow when building an application, ensuring that every stage—from initial component creation to final integration—is clearly defined and aligned with best practices for any future project."

1. Initially we usually do this:

1. Configuring App Permissions

* Why:  
  Before the app can access data from the Internet (such as retrieving API data), it needs explicit permission. Forgetting this can lead to runtime errors when trying to perform network operations.
* What is done:  
  The instructor shows how to add the Internet permission to the AndroidManifest.xml file by inserting the following line:

xml

CopyEdit

<uses-permission android:name="android.permission.INTERNET"/>

This permission enables the app to perform network operations.

2. Organizing the Project Structure

* Purpose:  
  Following clean architecture principles makes the codebase easier to maintain, scale, and manage as the project grows.
* Steps taken:
  + Creating Packages:  
    The instructor creates multiple packages within the main directory to keep the code organized:
    - di: For dependency injection-related classes.
    - data: To hold data-related classes such as data sources.
    - model: For the data models that represent the app's entities.
    - navigation: To manage navigation within the app.
    - network: For handling API calls and network-related operations.
    - repository: Where repository classes reside, acting as the mediator between the data sources and the rest of the app.
    - utils: For utility functions or helper classes.
    - components: For reusable UI components or other shared code.
    - widgets: For small, custom UI widgets.
    - screens: To organize the different screen implementations or UI layouts.
  + Benefit:  
    Structuring your project in this way enforces separation of concerns and makes it much easier to locate and manage specific parts of the code as the project expands.

3. Setting Up Dependency Injection (DI) with Hilt

* Why DI?  
  Dependency Injection simplifies how dependencies are provided to different parts of the application. This results in code that is more testable and modular.
* Steps taken:
  + Creating the Application Class:
    - A new Kotlin class named WeatherApplication is created.
    - This class extends AndroidApplication, serving as the top-level entry point for the app.
    - Knows Everything about all the assets in the project
  + Integrating Hilt:
    - The class is annotated with @HiltAndroidApp.
      * Purpose: This tells Hilt that this is the main application class where dependency injection should start.
    - Registration:
      * The WeatherApplication class is then registered in the AndroidManifest.xml by specifying its name. This step ensures that Android knows to use this custom application class.
  + Setting Up DI Modules:
    - A new module (or DI class) called AppModule is created. Although it's only briefly introduced here, modules in Hilt are used to define how and where dependencies should be provided.
  + Main Activity Setup:
    - In MainActivity, the annotation @AndroidEntryPoint is added.
      * Purpose: This marks the activity as a candidate for dependency injection. Hilt will automatically generate and supply the necessary dependencies for this activity.

Key Takeaways

* Permissions:  
  Always ensure necessary permissions (like Internet access) are declared early in development.
* Structured Project Organization:  
  Building a well-organized folder structure from the start sets a strong foundation for clean architecture, which eases future development and maintenance.
* Dependency Injection with Hilt:  
  Setting up Hilt by annotating the application class and activities simplifies dependency management, making the app more modular and easier to test.

By following these steps, the lecture demonstrates how to create a robust foundation for an Android project, making future enhancements and maintenance more manageable while adhering to best practices.

1. Now Navigations:

**1. Overview of the Navigation Concept**

* **Multiple Screens:**  
  The weather app includes several screens (splash, main, search, favorites, about, settings, etc.). Instead of managing them all at once, the instructor uses a “divide and conquer” approach.
* **Starting Small:**  
  The initial focus is on setting up the navigation system with a splash screen as the entry point. This basic structure will later be expanded to include other screens.

**2. Creating an Enum for Screens**

* **Purpose:**  
  An enum (enumeration) is used to list all possible screens/routes in one centralized location. This way, if new screens need to be added or routes modified, you only update the enum.
* **Example Steps:**
  + **Define an Enum:**  
    Create a new Kotlin enum class (e.g., WeatherScreens) to store identifiers for each screen:

kotlin

CopyEdit

enum class WeatherScreens {

SplashScreen,

MainScreen,

AboutScreen,

FavouriteScreen,

SettingsScreen

}

* + **Benefits:**  
    This practice separates the concern of naming and managing routes, making your navigation system easier to maintain.

**3. Setting Up the Navigation Controller**

* **NavController Role:**  
  The NavController (provided by Jetpack Compose) manages app navigation by keeping track of the back stack and handling navigation actions between composables.
* **Implementation:**
  + **Creating the NavController:**  
    Inside a composable function (WeatherNavigation), the controller is created using rememberNavController().
  + **Example:**

kotlin

CopyEdit

@Composable

fun WeatherNavigation() {

val navController = rememberNavController()

// ...

}

**4. Building the Navigation Graph with NavHost**

* **NavHost Concept:**  
  A NavHost acts as a container for your composable screens. It uses the NavController to determine which composable should be displayed.
* **Key Parameters:**
  + **navController:**  
    Pass the created NavController to the NavHost.
  + **startDestination:**  
    Define the initial screen (in this case, the splash screen) using the enum value:

kotlin

CopyEdit

startDestination = WeatherScreens.SplashScreen.name

* **Setting Up the NavHost:**

kotlin

CopyEdit

NavHost(navController = navController, startDestination = WeatherScreens.SplashScreen.name) {

// Define composable routes here

}

**5. Defining Individual Screens as Composables**

* **Mapping Routes to Screens:**  
  Inside the NavHost lambda, each screen is registered as a composable function. The route is specified by using the enum name.
* **Splash Screen Registration:**
  + **Code Explanation:**  
    When the route equals WeatherScreens.SplashScreen.name, the corresponding composable WeatherSplashScreen is displayed.

kotlin

CopyEdit

composable(WeatherScreens.SplashScreen.name) {

WeatherSplashScreen(navController = navController)

}

* + **Parameters:**  
    The WeatherSplashScreen receives the NavController to enable navigation (for example, moving to the main screen after the animation).
* **Main Screen Registration:**
  + **DI Integration:**  
    For the main screen, the instructor uses Hilt to inject the MainViewModel by calling hiltViewModel<MainViewModel>().

kotlin

CopyEdit

composable(WeatherScreens.MainScreen.name) {

val mainViewModel = hiltViewModel<MainViewModel>()

MainScreen(navController = navController, mainViewModel)

}

* + **Purpose:**  
    This setup not only registers the main screen but also ensures that the screen is provided with its required dependencies.

**7. Key Takeaways**

* **Centralized Screen Management:**  
  Using an enum for screen routes centralizes navigation management, making the code cleaner and easier to update.
* **Seamless Navigation Flow:**  
  The combination of NavController, NavHost, and composable functions provides a smooth navigation experience within the app.
* **Dependency Injection Integration:**  
  Incorporating Hilt (via hiltViewModel()) within your composable functions ensures that dependencies are managed efficiently, promoting a modular and testable architecture.
* **Scalability:**  
  The structured approach allows for effortless addition of new screens and routes as the application grows.



1. Main Activity as the Entry Point

Purpose of MainActivity:  
MainActivity serves as the starting point of the app. It’s where the entire Compose UI is set up and where navigation is initiated.

Hilt Integration:  
The class is annotated with @AndroidEntryPoint, which integrates Dagger Hilt into the activity so that dependencies can be injected throughout the composables (such as the MainViewModel used later).

Edge-to-Edge Display:  
The method enableEdgeToEdge() is called in onCreate, which configures the activity to display content edge-to-edge, making full use of the device screen.

Setting the Content:  
The setContent block replaces the traditional XML layout with a Compose-based UI. Here, the function WeatherApp() is invoked to build the app’s UI.

2. The WeatherApp Composable Function

Theme Application:  
Inside WeatherApp(), the custom JetWeatherForecastTheme is applied to ensure that all components follow a consistent design and styling.

Scaffold Usage:  
A Scaffold is used as the basic layout structure. It provides default slots for things like app bars, floating action buttons, and a content area. In this case, it fills the entire screen (Modifier.fillMaxSize()).

Handling Inner Padding:  
The Scaffold passes an innerPadding value which is applied to the main content. This ensures that the content doesn’t overlap with any system UI elements (like status bars or navigation bars).

Column Layout:  
A Column composable is placed within the scaffold’s content area.

Vertical Arrangement:  
Set to Arrangement.Center so that its children are centered vertically.

Horizontal Alignment:  
Set to Alignment.CenterHorizontally to center items horizontally.

Padding and Fill:  
The column uses both the provided innerPadding and fillMaxSize() to occupy the full available space.

Embedding Navigation:  
Within the column, the WeatherNavigation() composable is called. This function (described in previous lectures) sets up the navigation controller, nav host, and the initial destination (SplashScreen).

3. How Navigation is Triggered

Calling WeatherNavigation:  
By placing the WeatherNavigation() composable inside the column, you effectively embed the navigation graph into the UI.

NavHost Initialization:  
When WeatherNavigation() is executed, it creates a NavController using rememberNavController() and initializes a NavHost with the splash screen as its starting destination.

SplashScreen Display:  
This setup ensures that, on launch, the splash screen is the first composable shown to the user.

4. Practical Flow

Activity Launch:  
When the app starts, MainActivity is created, and onCreate() is executed.

UI Set-Up:  
Inside setContent, the WeatherApp() composable builds the UI structure:

It applies the theme.

It sets up a scaffold that fills the screen.

It creates a centered column that holds the navigation composable.

Navigation Activation:  
The WeatherNavigation() composable initializes the navigation controller and host, defining the start destination as the splash screen.

Splash Screen Display:  
The splash screen (as part of the navigation graph) is rendered immediately, and the navigation system is now in place for later transitions (such as moving to the main screen after an animation).

1. First we will make model classes/objects of the JSON data from the link that the API provides:

@project Clouds Coord Main Sys WeatherItem WeatherObject Wind ,what are these files that are here and what does these signifies as i made these model class by coping the link of the api for weather app.

🡪 These files represent the data model classes for your weather app. They're designed to map the JSON response from the OpenWeatherMap API to Kotlin objects. Here's what each class signifies:

WeatherItem: The main container class that holds all weather data for a location. This is the root response object from the API.

WeatherObject: Represents specific weather condition details like description, icon, ID, and main weather type.

Clouds: Contains cloud coverage information (percentage).

Coord: Contains geographical coordinates (latitude and longitude).

Main: Contains core weather data like temperature, pressure, humidity.

Sys: Contains system metadata like country code, sunrise/sunset times.

Wind: Contains wind information like speed and direction.

These classes were created to match the JSON structure of the OpenWeatherMap API response. When your app makes a request through Retrofit (as defined in your WeatherApi interface), the JSON response gets automatically converted into these Kotlin data objects.

1. Setting up the Weather API Interface for Retrofit (HTTP library)

and the steps for using OpenWeather API using Retrofit:

Retrofit helps to make the HTTP calls to the API to retrieve information from the source/Server-Side. Retrofit abstracts(removes) the hustle to pass and call the HTTP JSON data.

1. First define Constants like:

->API key

->Base URL

1. Then we will make the WeatherApi network interface for retrofit i.e. to interact with the API server side to retrieve the information we needed:

This code defines the WeatherApi interface which is a crucial component of the weather forecast application. Here's what it does and why it's important:

API Service Definition: It creates a Retrofit interface that defines how the app communicates with the OpenWeatherMap API.

Annotated Interface:

@Singleton ensures only one instance of this interface exists in the dependency injection system.

@GET specifies the endpoint for retrieving weather data.

Network Request Method: The getWeather suspend function handles asynchronous API calls using Kotlin coroutines, with parameters:

City name (query)

Units of measurement (defaulting to imperial)

API key (pulled from Constants)

Return Type Issue: The code attempts to return a WeatherObject, but there appears to be a mismatch - the defined WeatherObject model only contains basic weather condition information (description, icon, id, main), while the OpenWeatherMap API returns a much more comprehensive response with temperature, location, and other data.

This interface is significant because it:

Provides the network layer for the application

Abstracts the API communication details

Works with Dagger Hilt for dependency injection

Uses Kotlin coroutines for efficient asynchronous operations

1. Instantiation of retrofit in appModule:

Dependency Injection (DI) helps manage class dependencies efficiently. In your app's module (e.g., AppModule), you create *providers* — these are responsible for initializing required classes or dependencies like databases, repositories. This ensures your app's setup happens smoothly in the background.

//# AppModule.kt in a Jetpack Weather Forecast App  
//  
//## What is it?  
//`AppModule.kt` is a Dagger Hilt dependency injection module that centralizes the creation and configuration of application-wide dependencies.  
//  
//## Implementation  
//The module:  
//- Uses `@Module` and `@InstallIn(SingletonComponent::class)` to define an application-level dependency module  
//- Provides a singleton instance of `WeatherApi` through the `provideOpenWeatherApi()` method  
//- Configures Retrofit with:  
// - Base URL from Constants  
// - Gson converter for JSON parsing  
// - Creates an implementation of the WeatherApi interface  
//  
//## Significance  
//  
//### Dependency Injection  
//It implements the dependency injection pattern using Dagger Hilt, allowing components throughout the app to receive dependencies without knowing how to create them.  
//  
//### Network Layer Configuration  
//The module centralizes Retrofit configuration for fetching weather data, which is critical for this weather forecast application.  
//  
//### Single Responsibility  
//By extracting network setup into this module:  
//- The code is more maintainable  
//- Network configuration is done in one place  
//- Other components can focus on their specific responsibilities  
//  
//### Testability  
//Dependency injection makes the codebase more testable by allowing dependencies to be easily mocked during testing.  
//  
//### Singleton Management  
//The `@Singleton` annotation ensures only one instance of `WeatherApi` exists throughout the app lifecycle, optimizing resource usage.  
//  
//This module acts as the foundation for the app's network operations, enabling components to request weather data through a properly configured API client.

1. Making Repository : When trying to get data, It tries to get data from directly the network Weather Api, Accessing getweather directly
2. We use data injection here: we will get weatherObject from it i.e the data directly and thus we need more metadata attached to it.
3. Thus we make a data layer and wrap that weatherObject data into it: DataOrException (Using generics), we use data coming from the weatherObject, Exception and loading.
4. Making ViewModel to call the information and show it to the user by using WeatherRepository through the mainscreen

That was a thorough debugging session! The main takeaways:

1. Instantiating ViewModel
   * Used hiltViewModel() to create an instance of MainViewModel in the Composable.
   * Passed it down to MainScreen.
2. Fixing API Response Handling
   * The JSON model had a mismatch (pop expected an Int but got a Double).
   * Corrected it in the data class.
3. Refactoring ViewModel Data Handling
   * Originally stored API response in a MutableState, leading to null issues.
   * Replaced it with a suspend function returning DataOrException<Weather, Boolean, Exception>.
   * Called getWeatherData(city) in the Composable using produceState().
4. Final Success
   * API returned the correct payload.
   * UI displayed fetched data without persistent loading issues.
5. Making Appbar:

✅ Steps Taken:

1. Rename & Structure Setup:
   * Renamed ShowData to MainScaffold.
   * Used Scaffold composable to structure the screen (with topBar).
2. Data Flow Restructure:
   * Moved produceState logic to the main screen.
   * Passed the actual Weather data object (not ViewModel) to MainScaffold.
3. Scaffold Usage:
   * Used Scaffold(topBar = { ... }) to add top bar composable.
   * Created MainContent(weather) composable for the main body content.
4. Verified Functionality:
   * Displayed city name using Text(data.city.name) to confirm setup works.
5. AppBar Component Creation:
   * Created a new composable WeatherAppBar inside a new Kotlin class.
   * Made the AppBar customizable with:
     + title: String
     + icon: ImageVector?
     + isMainScreen: Boolean
     + elevation: Dp
     + navController: NavController
     + onAddAction and onButtonClicked callbacks
6. Implemented Basic AppBar UI:
   * Used TopAppBar() with title Text(...).
   * Applied styling: bold font, 15sp, color from MaterialTheme.colors.onSecondary.
7. Tested Preview & MainScreen Integration:
   * Added WeatherAppBar(title = "Helena MT") to verify it shows up.

**✅ Refactored the TopAppBar Structure**

Instead of nesting everything inside a Row, we now:

* Use the built-in TopAppBar()'s parameters:
  + title = { ... }
  + actions = { ... }
  + navigationIcon = { ... }
  + backgroundColor, contentColor, elevation

**✅ Implemented Conditional Actions**

* If isMain == true, two action icons are shown:
  + **Search**
  + **More (3 dots)**
* Else, shows an empty Box() (i.e., no actions)

**✅ Dynamic Title**

* Title now uses data from the weather object:

kotlin

CopyEdit

"${weather.city.name}, ${weather.city.country}"

**✅ Navigation Icon Support**

* Passed an optional icon: ImageVector?
* If icon != null, show back arrow and handle click via:

kotlin

CopyEdit

onButtonClicked?.invoke()

**✅ Style Adjustments**

* Title styling:
  + Color: MaterialTheme.colors.onSecondary
  + FontWeight: Bold
  + FontSize: 15.sp

**✅ Logging Back Arrow Clicks**

* Demonstrated logging with:

kotlin

CopyEdit

Log.d("TAG", "Button Clicked")

**🔁 Reusable AppBar**

This app bar is now **modular** and can be reused across screens by:

* Passing isMain = true/false
* Supplying optional icon
* Handling click logic via onButtonClicked

**✅ Objective**

Build the **top section of the main screen** UI, starting with:

* Current date display
* Weather icon
* Temperature
* Weather description

**🔨 Steps & Implementation Details**

**1. Cleaning Up**

* Removed temporary/test UI components like the placeholder WeatherAppBar and hardcoded Text.
* Prepares to replace them with real, data-driven UI.

**2. Main Column Layout**

* Created a Column to stack items vertically (top to bottom layout).

kotlin

CopyEdit

Column(

modifier = Modifier

.padding(4.dp)

.fillMaxWidth(),

verticalArrangement = Arrangement.Center,

horizontalAlignment = Alignment.CenterHorizontally

)

**3. Date Text**

Displays the current date (e.g., **"November 29"**) with styling:

kotlin

CopyEdit

Text(

text = "November 29",

style = MaterialTheme.typography.caption,

color = MaterialTheme.colors.onSecondary,

fontWeight = FontWeight.Bold,

modifier = Modifier.padding(6.dp)

)

**4. Weather Summary "Circle" UI**

A Surface is used to draw a circular card-like widget with dynamic weather data inside.

kotlin

CopyEdit

Surface(

modifier = Modifier

.padding(4.dp)

.size(200.dp),

shape = CircleShape,

color = Color(0xFFFFFC400) // Yellow background

) {

Column(

verticalArrangement = Arrangement.Center,

horizontalAlignment = Alignment.CenterHorizontally

) {

// Image, Temperature, Description go here

}

}

**5. Temperature & Weather Description**

* **Temperature** displayed using large text with H4 style.
* **Weather description** shown in italic text style.

kotlin

CopyEdit

Text(

text = "54",

style = MaterialTheme.typography.h4,

fontWeight = FontWeight.ExtraBold

)

Text(

text = "Snow",

fontStyle = FontStyle.Italic

)

**6. Displaying the Weather Icon (Dynamically)**

**✅ Getting Image Icon from OpenWeatherMap API**

* Weather icons are **not direct image URLs**.
* They come as **codes (e.g., "10d")** inside the JSON at weather[0].icon.

**Icon Image URL Format:**

css

CopyEdit

https://openweathermap.org/img/wn/{icon}.png

**✅ Construct the Dynamic URL:**

kotlin

CopyEdit

val imageUrl = "https://openweathermap.org/img/wn/" + data.list[0].weather[0].icon + ".png"

**7. Reusable Image Composable (WeatherStateImage)**

Created a reusable composable to load images from URLs using **Coil**, a modern image loading library for Compose.

kotlin

CopyEdit

@Composable

fun WeatherStateImage(imageUrl: String) {

Image(

painter = rememberImagePainter(imageUrl),

contentDescription = null,

modifier = Modifier.size(80.dp)

)

}

This composable is then invoked inside the circle card:

kotlin

CopyEdit

WeatherStateImage(imageUrl = imageUrl)

**8. Testing with Different Cities**

* Tested the dynamic UI by changing the city (e.g., **Moscow** vs **Portland**).
* Verified that the temperature, description, and icon update based on API data.

**📌 Key Concepts Demonstrated**

| **Concept** | **Usage** |
| --- | --- |
| Scaffold | For screen layout (used earlier in setup) |
| Surface + CircleShape | To create a circular container |
| Coil (rememberImagePainter) | For loading images from the web |
| Column + alignment | Vertical arrangement of weather info |
| Dynamic data mapping | Weather icon, temperature, and description |
| Modularization | WeatherStateImage() is reusable |

**🧠 Pro Tips From the Lecture**

* Use Surface for highly customizable UI containers.
* Use 0xFFFFFFFF hex format for defining custom colors.
* Test UI with multiple cities to confirm data updates correctly.
* Reuse composables like WeatherStateImage() to keep code clean and flexible.

1. **Weather Details row:**

**🛠️ Key Steps:**

1. **Intro to the Structure**:
   * Start with a Text ("This Week") styled with subtitle1 and FontWeight.Bold.
2. **Scrollable List (LazyColumn)**:
   * Enclosed inside a Surface with:
     + fillMaxWidth, fillMaxHeight
     + Background hex color
     + Rounded corners (RoundedCornerShape(14.dp))
   * LazyColumn used to show multiple rows efficiently
   * items(weatherList) to iterate over a list of weather forecasts
   * Each item initially displayed with a simple Text(weather.maxTemp.toString()) for testing
3. **Weather Row Composable (Reusability)**:
   * A new composable WeatherDetailRow(weather: Weather) is created
   * This is where actual row content is built
4. **Surface for Each Row**:
   * With modifier.padding(3.dp).fillMaxWidth()
   * Custom shape using CircleShape.copy(topEnd = CornerSize(6.dp))
   * Background Color.White
5. **Row Layout**:
   * Row with:
     + fillMaxWidth()
     + verticalAlignment = Alignment.CenterVertically
     + horizontalArrangement = Arrangement.SpaceBetween
6. **Text for Day of Week**:
   * formattedDate.split(",")[0] to get only day name (e.g., "Thursday")
   * Modifier.padding(start = 5.dp)
7. **Weather Icon**:
   * WeatherStateImage(imageUrl = weather.iconUrl) is added after the day text
   * Nicely aligned due to the Row’s spacing rules

**💡 Takeaways:**

* This structure allows **aesthetic control** with Surface and Shape.copy()
* LazyColumn + custom Row makes the weather detail screen dynamic and **scalable**
* Breaking layout into **composables** (WeatherDetailRow) makes code clean and reusable
* Extracting day name from date using split() is a clever formatting hack

**🧠 What You Should Be Comfortable With:**

* Using LazyColumn and items()
* Creating reusable composables
* Using Surface for customizable card-like containers
* Aligning children within Row
* Parsing and formatting strings

**🌥️ Goal:**

**Add a pill-shaped background behind the weather description** and show the **high/low temperatures** with **custom colors** using **annotated strings**.

**✅ 1. Weather Description Pill (Pill-Shaped Background + Text)**

**➤ How it’s implemented:**

* Use a Surface to make the pill-shaped background.
* Set:
  + modifier with optional padding
  + shape = CircleShape
  + color = Color(hex)

**➤ Inside the Surface:**

kotlin

CopyEdit

Text(

text = weather.weather[0].description,

modifier = Modifier.padding(4.dp),

style = MaterialTheme.typography.caption

)

**➤ Result:**

A nice **rounded colored background** (pill shape) behind the **description text** like scattered clouds.

**✅ 2. High/Low Temperature (Using Annotated String)**

**➤ Why:**

To show **Max Temp in one color** (e.g., Blue) and **Min Temp in another** (e.g., Light Gray), all in the same line.

**➤ How:**

* Use Text(buildAnnotatedString { ... })
* Within that, use withStyle(style = SpanStyle(...)) for both high and low parts.
* append() to add the values.

**➤ Code Example:**

kotlin

CopyEdit

Text(

buildAnnotatedString {

withStyle(style = SpanStyle(color = Color.Blue.copy(alpha = 0.7f), fontWeight = FontWeight.SemiBold)) {

append("${formatDecimals(weather.temp.max)}° ")

}

withStyle(style = SpanStyle(color = Color.LightGray)) {

append("${formatDecimals(weather.temp.min)}°")

}

}

)

**🔧 Bonus:**

* formatDecimals() is used to clean up the temperature values (e.g., convert 28.9764 to 29).
* Unicode for the degree symbol: \u00B0 or use °.

**🧠 Summary:**

| **Feature** | **Implementation** |
| --- | --- |
| Weather description pill | Surface + CircleShape + Text |
| High/Low temp display | Text(buildAnnotatedString { ... }) |
| Styling inside string | withStyle { append(...) } |

1. **Search Screen:**

**🔄 Goal of This Part:**

When the **search icon** is clicked, the app should **navigate to a new "Search Screen"** where the user can enter a city name and search for weather data.

**📍 Where This Logic Starts:**

* Back in the **main screen** of the weather app.
* A lambda onAddActionClicked is used to define what happens when the search (add) icon is clicked.
* Inside this lambda:

kotlin

CopyEdit

navController.navigate(WeatherScreens.SearchScreen.name)

This navigates to the "SearchScreen".

**📦 Creating the Search Screen:**

1. **New package**: search
2. **New Kotlin file**: SearchScreen.kt
3. Inside it, a @Composable function:

kotlin

CopyEdit

@Composable

fun SearchScreen(navController: NavController)

* + Contains a Scaffold
  + Top bar is initially blank (but will be updated)

**🧭 Update Navigation Graph:**

* Inside WeatherNavigation.kt, add the new screen:

kotlin

CopyEdit

composable(WeatherScreens.SearchScreen.name) {

SearchScreen(navController)

}

**✔ Verifying Navigation:**

* Click the search icon ➜ Navigation to the search screen works ✅

**🎨 Custom Top Bar for Search Screen:**

* Replaces the default top bar with a **custom WeatherAppBar**:

kotlin

CopyEdit

WeatherAppBar(

title = "Search",

navController = navController,

isMainScreen = false,

icon = Icons.Default.ArrowBack

) {

navController.popBackStack()

}

* + Now, user can press the back icon to return.

**🔠 Adding a Search Field (TextField/Form):**

1. Replace the placeholder Text("Search Screen") with a vertical-aligned Column.
2. Inside it, call a composable:

kotlin

CopyEdit

SearchBar(onSearch = { cityName -> /\* handle it \*/ })

**🔧 Create SearchBar Composable:**

* Takes onSearch: (String) -> Unit as a lambda parameter.
* Internally uses a Column to organize content.

**🔨 Create CommonTextField Composable:**

Used inside SearchBar. This is a custom **OutlinedTextField** with parameters:

**Parameters:**

* valueState: MutableState<String>
* placeholder: String
* keyboardType: default to KeyboardType.Text
* imeAction: default to ImeAction.Next
* onAction: KeyboardActions

**Features:**

* Handles user input & binds it to valueState
* Single line input
* Styled with:
  + Blue focused border
  + Black cursor
  + Rounded corners (15.dp)
  + Padding (start & end = 10.dp)

**🧠 State Management Inside SearchBar:**

* searchQueryState:

kotlin

CopyEdit

val searchQueryState = rememberSaveable { mutableStateOf("") }

Keeps user input intact even on device rotation.

* keyboardController:

kotlin

CopyEdit

val keyboardController = LocalSoftwareKeyboardController.current

Allows manual control of keyboard (e.g., hiding it after input).

* valid:

kotlin

CopyEdit

val valid = remember(searchQueryState.value) {

searchQueryState.value.trim().isNotEmpty()

}

Tracks if input is non-empty (basic validation).

**CONTINUED**

**1. Capturing Keyboard Actions in the Common Text Field**

**What’s New:**

* **Keyboard Action Handling:**  
  The common text field (used in the search bar) now captures the keyboard’s “Next” or “Enter” action. This happens through the KeyboardActions parameter on the text field.

**Implementation Details:**

* **Validation Step:**  
  When the user presses the keyboard action button, the lambda checks if the entered text (the city name) is valid.
  + It uses a state variable (remembered via rememberSaveable) to hold the search query.
  + A simple validation is applied: if the trimmed query is empty, the action is aborted (returning early).
* **Data Propagation:**  
  If the input is valid, the lambda invokes the provided onSearch function:
  + It passes the trimmed search query.
  + Then, it clears the query state (setting it to an empty string) so that the text field resets.
* **Controlling the Keyboard:**  
  The keyboard controller (using LocalSoftwareKeyboardController.current) is used to hide the keyboard after the action—improving user experience.

**2. Integrating the Search Bar into the Search Screen**

**What’s New:**

* **Reusable Search Bar:**  
  The search bar is composed using a dedicated composable function that internally calls the common text field.

**Implementation Details:**

* **Layout Setup:**  
  The search screen uses a Scaffold and a centered Column layout.
  + The search bar composable is inserted in the column, with modifiers for padding and alignment (e.g., fillMaxWidth() and centered horizontally).
* **Trailing Lambda for onSearch:**  
  Since onSearch is the last parameter of the search bar composable, a trailing lambda style is used. This allows the implicit variable (it), which represents the search query, to be handled directly in the lambda.
* **Logging and Testing:**  
  For demonstration purposes, the implementation logs the search query when the keyboard’s “next/enter” action is pressed.

**3. Updating Navigation Based on User Input**

**What’s New:**

* **Dynamic Route Construction:**  
  The user-entered city name is appended as an argument to the navigation route when returning from the search screen to the main screen.

**Implementation Details:**

* **Modifying the Navigation Route:**  
  In the navigation graph, the main screen route (for example, WeatherScreens.MainScreen.name) is updated to accept a city name parameter. This is done by:
  + Converting the fixed route to a string that includes a forward slash (/) followed by a variable (e.g., "/$city").
  + A default city is provided (e.g., "San Diego") if no value is passed.
* **Passing Arguments via Nav Arguments:**  
  Navigation arguments are set up using:
  + navArgument constructs for specifying that the passed data is of type string.
  + Consistency in naming is enforced (e.g., the key is "city" both when constructing the route and when retrieving the value in the main screen).
* **Retrieving the City Value on the Main Screen:**  
  In the main screen composable, the argument is extracted from the back stack entry’s bundle (e.g., using navBackStackEntry.arguments?.getString("city")).
  + A let block is then used to run the subsequent code that repopulates the main screen with the weather data for the newly searched city.
  + The extracted city value is passed to the subsequent weather data fetching and UI update logic.

**4. Final Testing and Flow**

* **User Interaction Flow:**
  + User navigates to the search screen by clicking on the action icon in the main screen.
  + The search screen displays the search bar where the user can input a city name.
  + When the user presses the enter/next key:
    - The text field’s keyboard action lambda validates the input.
    - If valid, it calls the onSearch function with the entered city name.
    - The search screen then uses the navigation controller to navigate back to the main screen, appending the city name in the route.
* **Repopulation of Main Screen:**
  + The main screen receives the city name through its navigation arguments.
  + The weather data fetching mechanism is triggered using this new city value, updating the UI dynamically.

**5. Summary of Key Benefits and New Features:**

* **Reusable Components:**  
  A common text field is created and reused within a search bar for modularity.
* **Enhanced User Experience:**
  + Capturing keyboard actions to trigger search operations.
  + Clearing the input field and hiding the keyboard after submission.
* **Dynamic Navigation:**
  + Passing data (city name) via dynamic routes and navigation arguments.
  + Using consistent naming and type validation to ensure smooth data exchange between composables.
* **Extensibility:**  
  The structure allows future extensions, such as adding additional search parameters or UI refinements, without major changes.

**CONTINUED**

**. SearchScreen Composable**

**Code Overview**

kotlin

CopyEdit

@Composable

fun SearchScreen(navController: NavController){

Scaffold(

topBar = {

WeatherAppBar(

navController = navController,

title = "Search",

elevation = 0.dp,

isMainScreen = false,

icon = Icons.AutoMirrored.Filled.ArrowBack

) {

navController.popBackStack()

}

}

) {

Surface(modifier = Modifier.padding(paddingValues = it)) {

Column(

modifier = Modifier

.padding(4.dp)

.fillMaxWidth(),

verticalArrangement = Arrangement.Center,

horizontalAlignment = Alignment.CenterHorizontally,

) {

SearchBar(

modifier = Modifier

.padding(16.dp)

.align(Alignment.CenterHorizontally)

) { city ->

navController.navigate(WeatherScreens.MainScreen.name + "/$city")

}

}

}

}

}

**Explanation**

* **Purpose:**  
  The SearchScreen composable represents the screen where the user can input the city name.
* **Scaffold and App Bar:**
  + A Scaffold is used to structure the screen.
  + The top bar is rendered with WeatherAppBar, which is configured in non-main mode (isMainScreen = false), uses an arrow-back icon, and its click action calls navController.popBackStack() to go back.
* **Surface & Column:**
  + Inside the Surface, a Column is used to center the search bar.
* **SearchBar Integration:**
  + SearchBar is inserted as a child of the column.
  + It takes a lambda parameter onSearch: (String) -> Unit.
    - **When the user triggers the keyboard action (presses "Next/Enter"),** that lambda is invoked with the search query.
  + In this lambda, navController.navigate(WeatherScreens.MainScreen.name + "/$city") is called.
    - This constructs a dynamic route by appending the entered city name to the main screen route.
    - For example, if the city entered is "Seattle", the route becomes something like "MainScreen/Seattle".

**2. SearchBar Composable**

**Code Overview**

kotlin

CopyEdit

@Composable

fun SearchBar(

modifier: Modifier = Modifier,

onSearch: (String) -> Unit = {},

) {

val searchQueryState = rememberSaveable { mutableStateOf("") }

val keyboardController = LocalSoftwareKeyboardController.current

val valid = remember(searchQueryState.value) {

searchQueryState.value.trim().isNotEmpty()

}

Column() {

CommonTextField(

valueState = searchQueryState,

placeholder = "Search",

onAction = KeyboardActions {

if (!valid) return@KeyboardActions

onSearch(searchQueryState.value.trim())

searchQueryState.value = ""

keyboardController?.hide()

}

)

}

}

**Explanation**

* **Purpose:**  
  SearchBar provides a UI component that lets users enter a city name and handles submission.
* **State Management:**
  + searchQueryState: Holds the current text input for the search query. It uses rememberSaveable to survive configuration changes.
  + valid: A derived state that checks if the current query is non-empty (after trimming).
* **Keyboard Handling:**
  + LocalSoftwareKeyboardController is used to hide the keyboard once the search action is complete.
* **Using CommonTextField:**
  + SearchBar wraps the CommonTextField composable.
  + It passes along:
    - valueState (the query state),
    - a placeholder text,
    - and onAction as KeyboardActions.
* **Keyboard Action Lambda:**
  + Inside the lambda, it first validates the input.
  + If valid, it calls onSearch passing the trimmed query, resets the field (searchQueryState.value = ""), and hides the keyboard.
  + This onSearch lambda in SearchBar is defined in SearchScreen to perform navigation.

**3. CommonTextField Composable**

**Code Overview**

kotlin

CopyEdit

@Composable

fun CommonTextField(

valueState: MutableState<String>,

placeholder: String,

keyboradType: KeyboardType = KeyboardType.Text,

imeAction: ImeAction = ImeAction.Next,

onAction: KeyboardActions = KeyboardActions.Default

) {

OutlinedTextField(

value = valueState.value,

onValueChange = { valueState.value = it },

label = { Text(text = placeholder) },

maxLines = 1,

singleLine = true,

textStyle = MaterialTheme.typography.bodyLarge,

keyboardOptions = KeyboardOptions(

imeAction = imeAction,

keyboardType = keyboradType

),

keyboardActions = onAction,

colors = TextFieldDefaults.colors(

focusedTextColor = Color.Blue

),

trailingIcon = {

Icon(imageVector = Icons.Filled.Search, contentDescription = "Search Icon")

},

shape = RoundedCornerShape(15.dp),

modifier = Modifier

.padding(10.dp)

.fillMaxWidth()

)

}

**Explanation**

* **Purpose:**  
  CommonTextField is a reusable text input component that standardizes the look and feel of text fields across the app.
* **Parameters:**
  + valueState: Holds the text input value.
  + placeholder: Placeholder text for the field.
  + keyboradType and imeAction: Define the behavior and appearance of the soft keyboard (default to text input and “Next” action).
  + onAction: Callback for keyboard actions (e.g., when the user presses Enter).
* **Appearance:**  
  The OutlinedTextField is customized with:
  + A label using the provided placeholder.
  + A trailing search icon.
  + Rounded corners (15.dp) and padding.
  + Custom colors (e.g., blue when focused).
* **Interconnection:**  
  This composable is used inside SearchBar to capture and manage the user's input for the search query.

**4. WeatherNavigation and Dynamic Route Handling**

**Code Overview for Navigation**

kotlin

CopyEdit

@Composable

fun WeatherNavigation() {

val navController = rememberNavController()

NavHost(navController = navController,

startDestination = WeatherScreens.SplashScreen.name) {

// Splash screen route

composable(WeatherScreens.SplashScreen.name) {

WeatherSplashScreen(navController = navController)

}

// Main screen route expecting a "city" parameter

val route = WeatherScreens.MainScreen.name

composable("$route/{city}",

arguments = listOf(

navArgument(name = "city") {

type = NavType.StringType

}

)

) { navBack ->

navBack.arguments?.getString("city").let { city ->

val mainViewModel = hiltViewModel<MainViewModel>()

MainScreen(navController = navController, mainViewModel, city = city)

}

}

// Search screen route

composable(WeatherScreens.SearchScreen.name) {

SearchScreen(navController = navController)

}

}

}

**Explanation**

* **NavHost Setup:**  
  The WeatherNavigation composable defines the navigation graph with three routes:
  1. **Splash Screen:** The starting destination.
  2. **Main Screen:** Expects a dynamic parameter (city).
  3. **Search Screen:** Displays the search bar and related UI.
* **Dynamic Route for MainScreen:**
  1. The route string is defined as "$route/{city}".
     + This means that when navigating to the main screen, a city name is appended to the route.
  2. **Arguments:**
     + A list of navArgument is provided to inform the navigation system that the route accepts an argument named "city" of type String.
* **Retrieving Arguments:**
  1. When navigating to the main screen, the lambda parameter navBack contains the arguments.
  2. The code retrieves the city string using:

kotlin

CopyEdit

navBack.arguments?.getString("city").let { city -> ... }

* 1. The retrieved city (which can be null if not provided) is then passed to MainScreen.
* **Flow:**
  1. **From the SearchScreen:**  
     The search bar’s onSearch lambda calls:

kotlin

CopyEdit

navController.navigate(WeatherScreens.MainScreen.name + "/$city")

This constructs a URL-like route that carries the search query.

* 1. **At the MainScreen:**  
     The main screen’s route is defined to accept that city parameter, and the weather data is then fetched accordingly.

**5. MainScreen, MainScaffold, and MainContent**

**MainScreen Overview**

kotlin

CopyEdit

@Composable

fun MainScreen(

navController: NavController,

mainViewModel: MainViewModel = hiltViewModel(),

city: String?

) {

val weatherData = produceState<DataOrException<Weather, Boolean, Exception>>(

initialValue = DataOrException(loading = true)

) {

value = mainViewModel.getWeather(city = city.toString())

}.value

if (weatherData.loading == true) {

CircularProgressIndicator()

} else if (weatherData.data != null) {

MainScaffold(weather = weatherData.data!!, navController)

}

}

**Explanation**

* **Purpose:**  
  MainScreen fetches the weather data for the given city (passed via navigation) and, once loaded, displays the main UI using a scaffold.
* **Using produceState:**  
  This is used to invoke the API call via the mainViewModel and to store the resulting weather data as state.
* **Conditional UI:**
  + If the data is still loading, show a CircularProgressIndicator.
  + Once data is available, call MainScaffold to display the main content.

**MainScaffold & MainContent**

* **MainScaffold:**  
  Wraps the main screen UI in a Scaffold that includes the app bar (using WeatherAppBar) with dynamic titles (showing the city and country) and an action icon that navigates back to the search screen.
* **MainContent:**  
  Displays detailed weather information such as temperature, weather icon, and other metrics using a combination of layout elements (e.g., Column, Surface, Canvas for gradients) and data formatting functions.

**6. WeatherAppBar**

**Code Overview**

kotlin

CopyEdit

@OptIn(ExperimentalMaterial3Api::class)

@Composable

fun WeatherAppBar(

title: String = "Title",

icon: ImageVector? = null,

isMainScreen: Boolean = true,

elevation: Dp,

navController: NavController,

onAddActionClicked: () -> Unit = {},

onButtonClicked: () -> Unit = {},

) {

Surface(

modifier = Modifier

.padding(horizontal = 2.dp, vertical = 2.dp)

.shadow(elevation = elevation),

shape = RoundedCornerShape(14.dp),

color = Color(0xFF6FA6D0)

) {

CenterAlignedTopAppBar(

title = {

Text(

text = title,

style = TextStyle(fontWeight = FontWeight.Bold, fontSize = 18.sp),

color = Color.Black

)

},

actions = {

if (isMainScreen) {

IconButton(onClick = {

onAddActionClicked.invoke()

}) {

Icon(imageVector = Icons.Default.Search, contentDescription = "search icon")

}

IconButton(onClick = {}) {

Icon(imageVector = Icons.Rounded.MoreVert, contentDescription = "menu icon")

}

} else {

Box() {}

}

},

navigationIcon = {

if (icon != null) {

Icon(

imageVector = icon,

contentDescription = "Null",

tint = Color.Black,

modifier = Modifier

.padding(8.dp)

.clickable { onButtonClicked.invoke() }

)

}

},

colors = TopAppBarDefaults.topAppBarColors(

containerColor = Color.Transparent

),

modifier = Modifier.fillMaxWidth()

)

}

}

**Explanation**

* **Purpose:**  
  This composable renders the top app bar for the screens. It is used both in the main and search screens.
* **Parameters:**
  + title: The text to display in the center of the bar.
  + icon: An optional navigation icon (e.g., back arrow) used when not on the main screen.
  + isMainScreen: A flag that toggles whether to show the extra action buttons.
  + elevation: The elevation (shadow depth) of the app bar.
  + navController: Passed to allow on-click navigation actions.
  + onAddActionClicked: Callback for when the action icon (search) is clicked.
  + onButtonClicked: Callback for when the navigation icon is clicked.
* **Structure & Styling:**
  + Uses a Surface for background styling and shadow.
  + Internally, CenterAlignedTopAppBar is used from Material 3.
  + **Title Slot:** Displays the title text with bold style.
  + **Actions Slot:**
    - If on the main screen, it displays both a search icon (which triggers onAddActionClicked) and a menu icon.
    - If not on the main screen, nothing (or an empty Box) is shown.
  + **Navigation Icon Slot:**
    - When a non-null icon is passed (typically in the search screen), it shows this icon which, when clicked, invokes onButtonClicked (commonly to pop the back stack).

**Overall Navigation Flow Recap**

1. **SearchScreen:**
   * User enters a city name into the search bar.
   * When submitting (via keyboard action), the onSearch lambda gets executed.
   * This lambda navigates to the main screen route with the city appended.
2. **Dynamic Routing:**
   * In WeatherNavigation, the main screen route is defined as "$route/{city}".
   * A navigation argument for "city" (of type String) is declared.
   * On navigation, the city value is retrieved and passed to the MainScreen composable.
3. **MainScreen:**
   * The passed city is used to fetch new weather data and update the UI.
   * The top app bar in MainScaffold displays the city and provides an action icon to go back to the search screen.
4. **About Screen**

**✅ Goal:**

When the user taps the **“more” icon**, a dropdown menu should appear with **"About", "Favorites", and "Settings"** options.

**🧠 Implementation Breakdown:**

**1. State Handling**

Inside your WeatherAppBar, you added:

kotlin

CopyEdit

val showDialog = remember { mutableStateOf(false) }

This tracks whether the dialog is shown or not.

**2. Toggle on Icon Click**

You modified the More (⋮) icon’s onClick:

kotlin

CopyEdit

showDialog.value = true

**3. Show Dialog Conditionally**

Right below the app bar's content:

kotlin

CopyEdit

if (showDialog.value) {

ShowSettingsDropDownMenu(showDialog = showDialog, navController = navController)

}

**🛠️ ShowSettingsDropDownMenu() Details**

**a. Dropdown Dialog Styling:**

Wrapped in a Column with Modifier.align(TopEnd) and padding to position it under the icon.

**b. DropdownMenu Usage:**

kotlin

CopyEdit

val expanded by remember { mutableStateOf(true) }

DropdownMenu(

expanded = expanded,

onDismissRequest = { expanded = false; showDialog.value = false },

modifier = Modifier

.width(140.dp)

.background(Color.White)

)

**c. Items Definition:**

kotlin

CopyEdit

val items = listOf("About", "Favorites", "Settings")

**d. Item Rendering Using forEachIndexed**

kotlin

CopyEdit

items.forEachIndexed { index, text ->

DropdownMenuItem(

onClick = {

expanded = false

showDialog.value = false

}

) {

Icon(...) // Uses a `when(text)` to show different icons

Spacer(modifier = Modifier.width(8.dp))

Text(text, fontWeight = FontWeight.W300)

}

}

* Icon changes based on the item (info, favorite\_border, settings).
* Clicking an item or outside closes the dialog.

**🎉 Final Output:**

* Click the more icon → Dialog slides down with 3 options.
* Tapping any option or outside → Dialog disappears.

**✅ Objective Recap:**

You already had a dropdown menu in your app and wanted to navigate to **About**, **Favorites**, and **Settings** screens when the respective item is clicked.

**🧩 Step-by-Step Explanation:**

**1. Handling Clicks on Dropdown Menu Items**

You detected which item (About/Favorites/Settings) is clicked using a clickable on the text, and then navigated accordingly using NavController.

**Example:**

kotlin

CopyEdit

clickable {

when (text) {

"About" -> navController.navigate(WeatherScreens.AboutScreen.name)

"Favorites" -> navController.navigate(WeatherScreens.FavoritesScreen.name)

else -> navController.navigate(WeatherScreens.SettingsScreen.name)

}

}

**2. Creating Missing Screens**

You noticed that the three screens (AboutScreen, FavoritesScreen, SettingsScreen) didn’t exist yet, so you created them as separate Composables in separate packages.

**Example:**

kotlin

CopyEdit

@Composable

fun AboutScreen(navController: NavController) {

Text(text = "About") // Placeholder

}

You did the same for FavoritesScreen and SettingsScreen.

**3. Registering Screens in the Navigation Graph**

You added the new screens to your NavGraph file so that Compose can navigate to them.

**Example:**

kotlin

CopyEdit

composable(WeatherScreens.AboutScreen.name) {

AboutScreen(navController = navController)

}

Repeat for Favorites and Settings screens.

**4. Designing the About Screen (UI)**

You enhanced the AboutScreen with a **Scaffold**, **TopBar**, and **Surface**.

**🛠️ Scaffold Structure:**

kotlin

CopyEdit

Scaffold(

topBar = {

WeatherAppBar(

title = "About",

navController = navController,

icon = Icons.Default.ArrowBack,

isMainScreen = false

) {

navController.popBackStack()

}

}

) {

Surface(modifier = Modifier.fillMaxSize()) {

Column(

horizontalAlignment = Alignment.CenterHorizontally,

verticalArrangement = Arrangement.Center

) {

Text(

text = stringResource(R.string.about\_app),

style = MaterialTheme.typography.subtitle1,

fontWeight = FontWeight.Light

)

}

}

}

**5. Adding String Resources**

Instead of hardcoding text, you used stringResource and added entries in res/values/strings.xml.

**Example:**

xml

CopyEdit

<string name="about\_app">Weather App v1.0</string>

You also added others like API details.

**6. Solving Warnings/Errors**

You resolved a warning about **experimental APIs** by adding the necessary annotations (likely @OptIn(...)) in the right places, including your MainActivity.

**✅ Final Test Run:**

You tested the dropdown clicks, and all navigations (to About, Settings, and Favorites) worked fine. The About screen showed properly styled text.

**🧠 Key Learnings Recap:**

| **Feature** | **What You Did** |
| --- | --- |
| Navigation | Used NavController.navigate(...) |
| Composables | Created clean, reusable screen components |
| Scaffold Layout | Used Scaffold, TopBar, Surface, and Column |
| Resources | Used stringResource and externalized strings |
| Styling | Used typography styles and font weights |
| Navigation Graph | Updated it with new screens |
| Back Stack | Used navController.popBackStack() to handle back navigation |
| Structure | Clean folder/package separation for each screen |