ANDROID DEVELOPMENT

Class 5: Navigation & Architecture

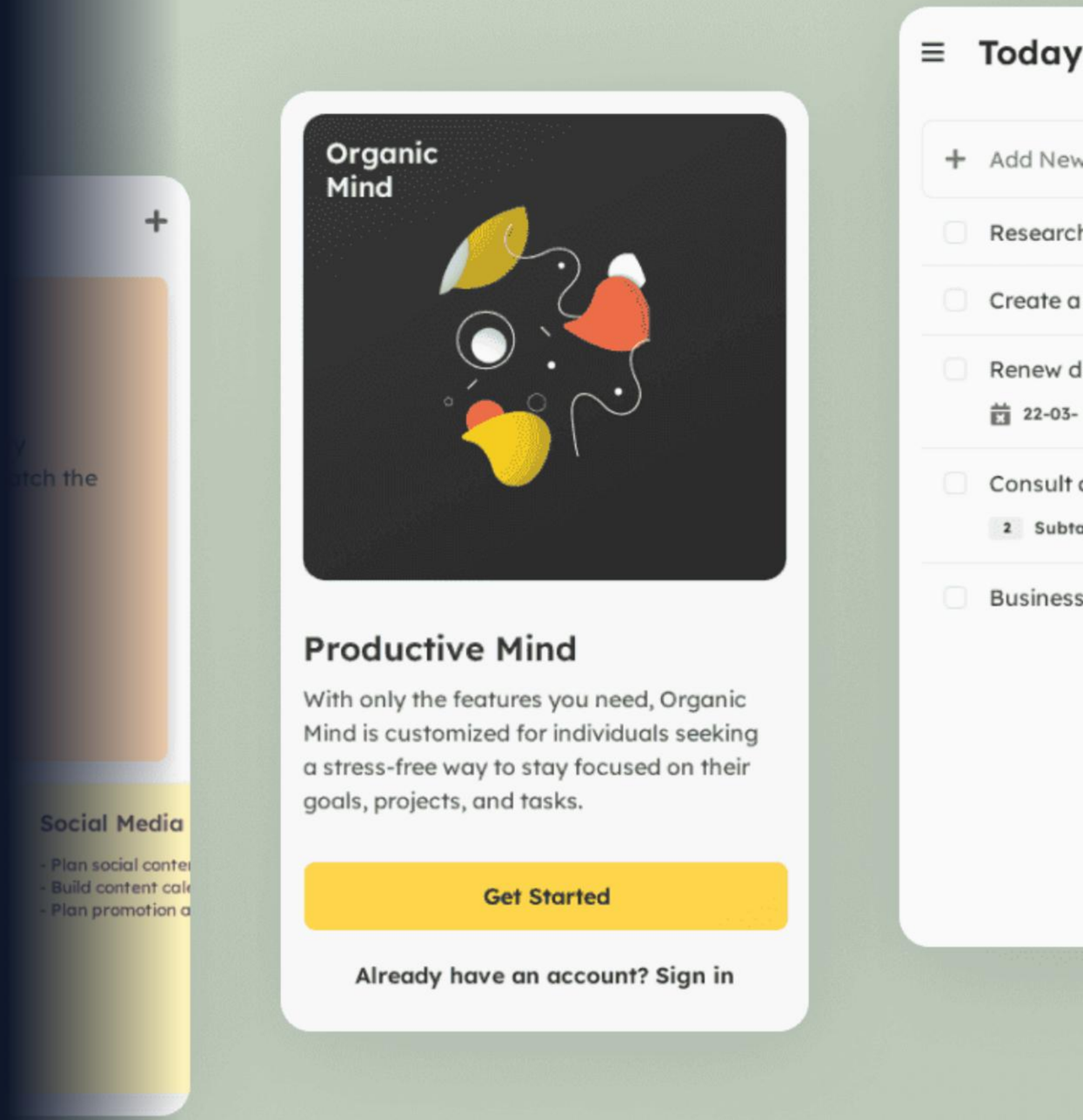
Moving between screens & separating logic from UI

Instructor: Mark Joseli

Today's Goal

Creating multi-screen apps with clean architecture.

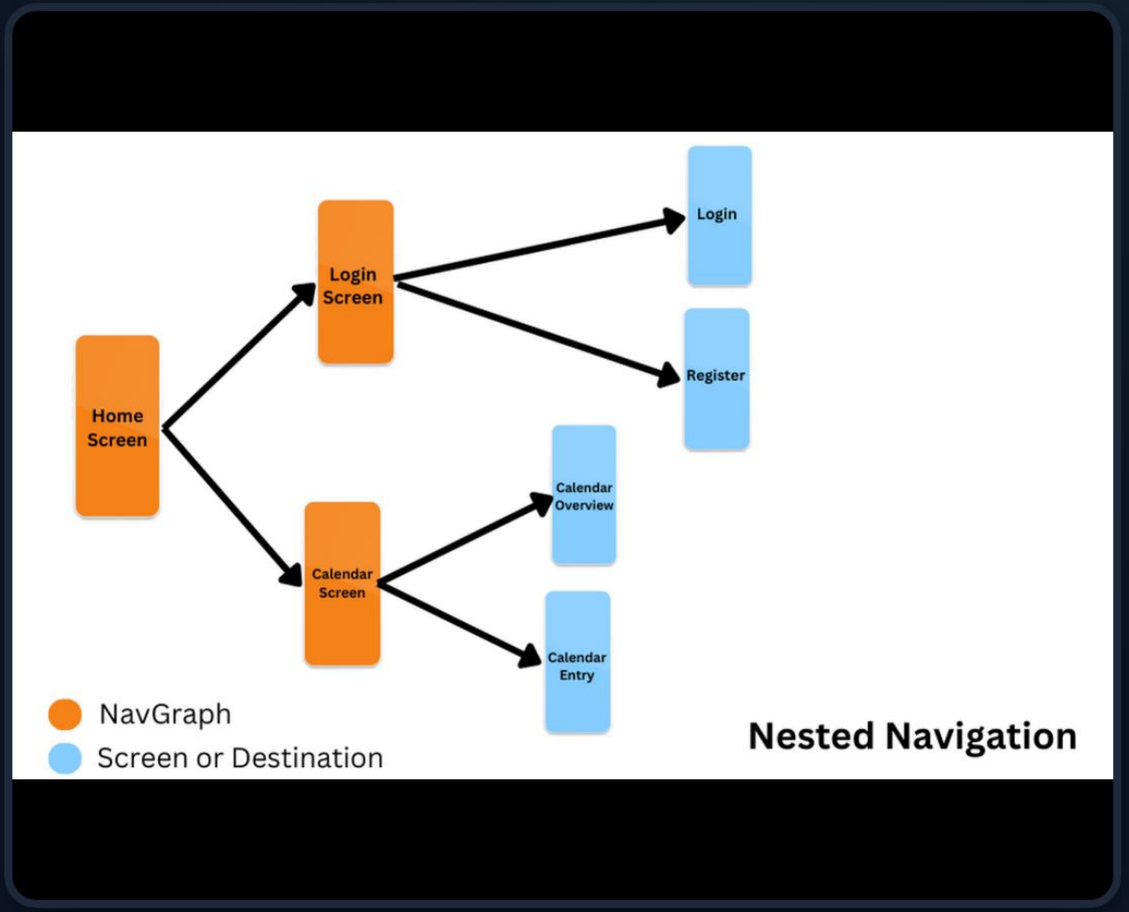
- Navigate between screens
- Separate Logic (MVVM)
- Pass data between views



Navigation Theory

The **Navigation Component** manages app navigation within a single Activity.

- **NavController:** The "Driver". It holds the back stack and knows how to swap screens.
- **NavHost:** The "Container". An empty box where the current screen is displayed.



Setting up `NavHost`

Define your map of screens (routes).

```
val navController = rememberNavController()
NavHost( navController = navController,
startDestination = "home" ) { composable("home") {
HomeScreen(...) } composable("detail") {
DetailScreen(...) } }
```

Each `composable` block defines a screen and its unique route name.

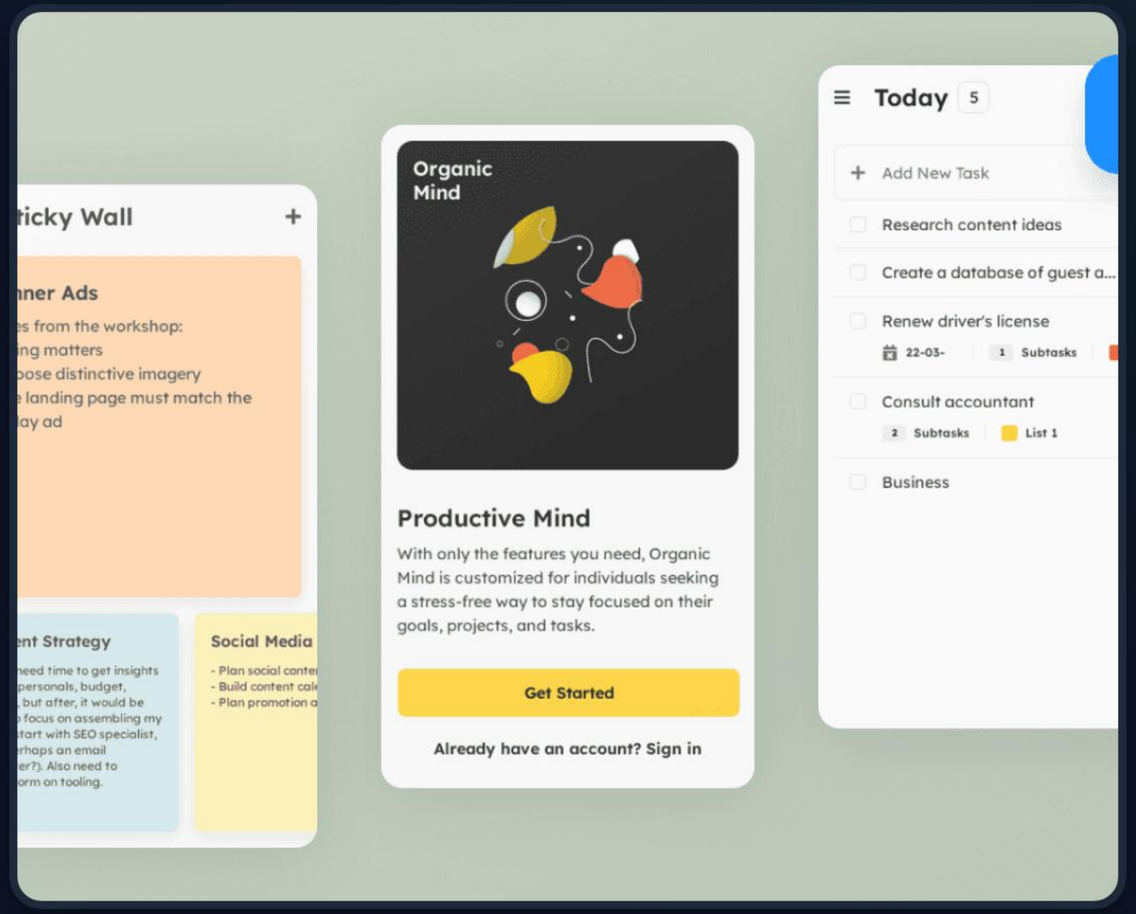
Routes map strings to UI.

Passing Data

Use simple string concatenation to pass arguments.

```
// 1. Navigate with ID
navController.navigate("detail/42") // 2. Define
Route with argument composable("detail/{id}") {
backStackEntry → val id =
backStackEntry.arguments?.getString("id")
DetailScreen(id) }
```

Think of it like a URL: `website.com/detail/42`



Why Architecture?

Why we stop putting logic inside the UI.

Bugs

Rotation destroys the UI. If your data lives in the UI, it gets lost.

Testing

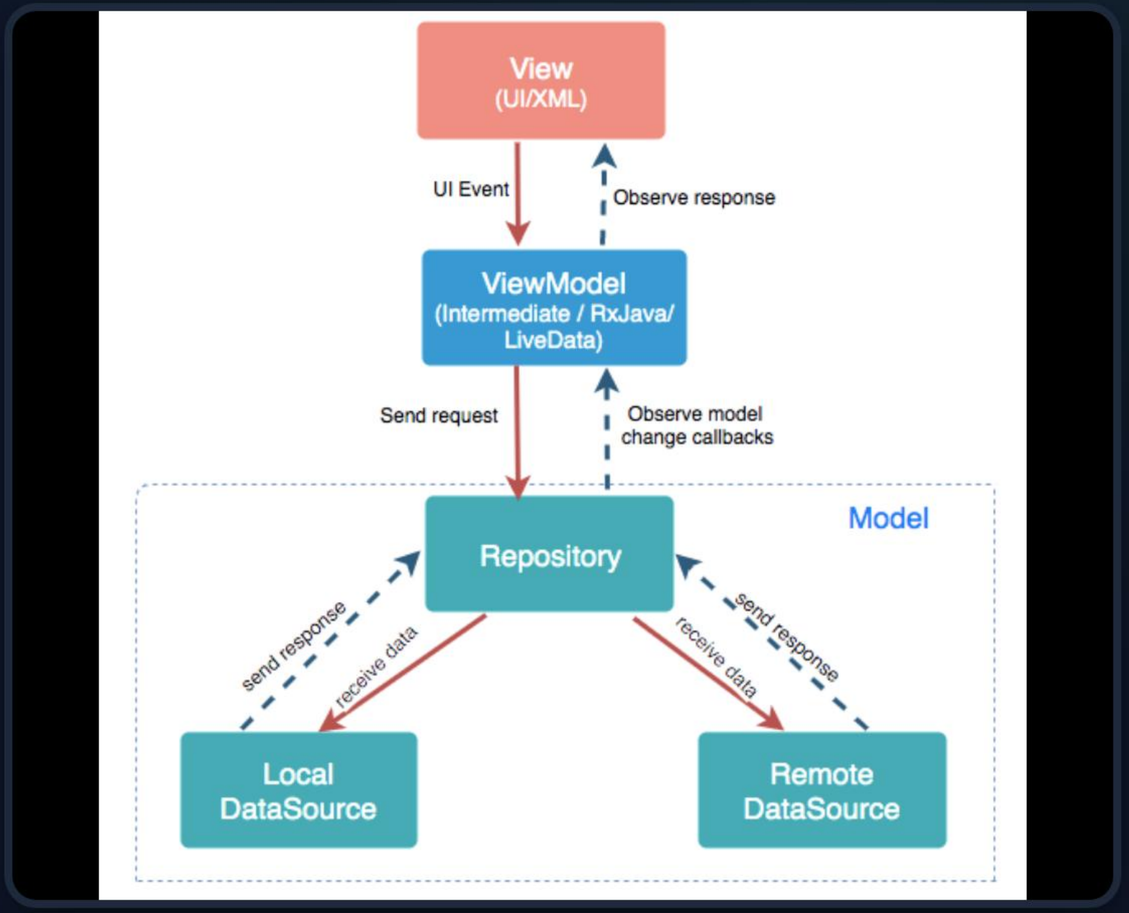
You can't write unit tests for a UI button click easily. You can test a function.

Separation

Keep your UI "dumb". It should only display data, not calculate it.

The MVVM Pattern

- **Model:** The Data (Database, API, Objects).
- **View:** The UI (Composable Functions). Displays state.
- **ViewModel:** The Brain. Holds state, survives rotation, talks to Model.

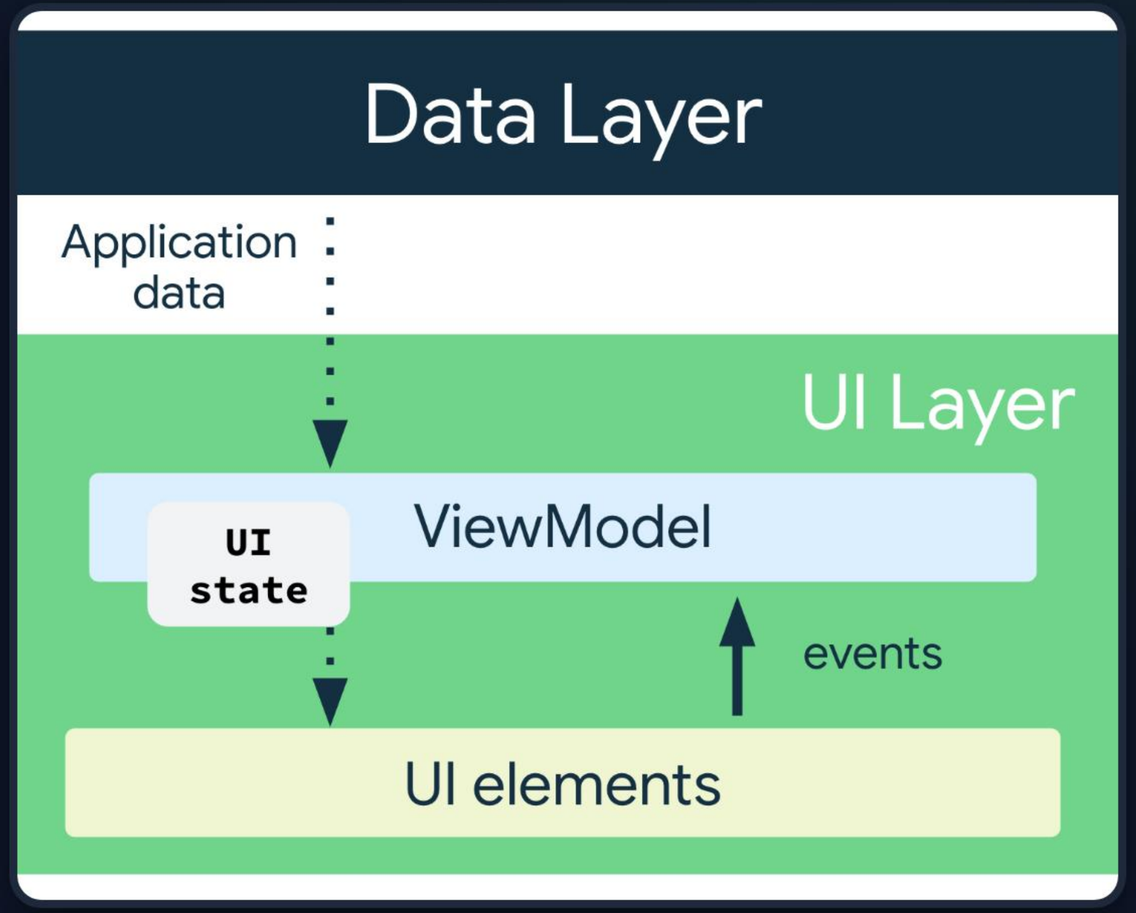


The ViewModel

A class that extends `ViewModel`.

```
class TaskViewModel : ViewModel() { // State lives here, safe from rotation var tasks = mutableStateListOf() private set fun addTask(name: String) { tasks.add(Task(name)) } }
```

The UI calls functions like `addTask()`, and the state updates automatically.

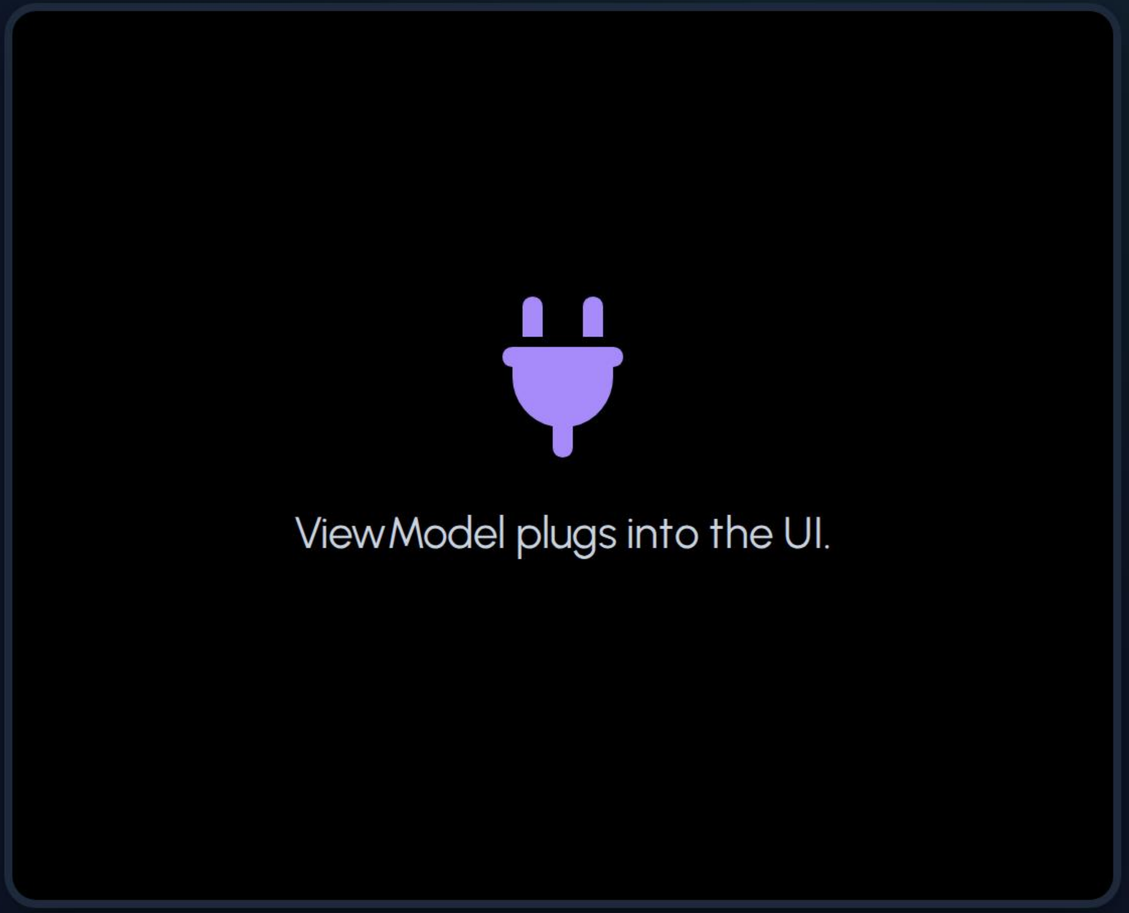


Connecting to UI

Inject the ViewModel into your screen.

```
@Composable fun TaskListScreen( viewModel:
TaskViewModel = viewModel() ) { val tasks =
viewModel.tasks LazyColumn { items(tasks) { task →
TaskRow(task) } } }
```

When `viewModel.tasks` changes, this screen recomposes automatically.

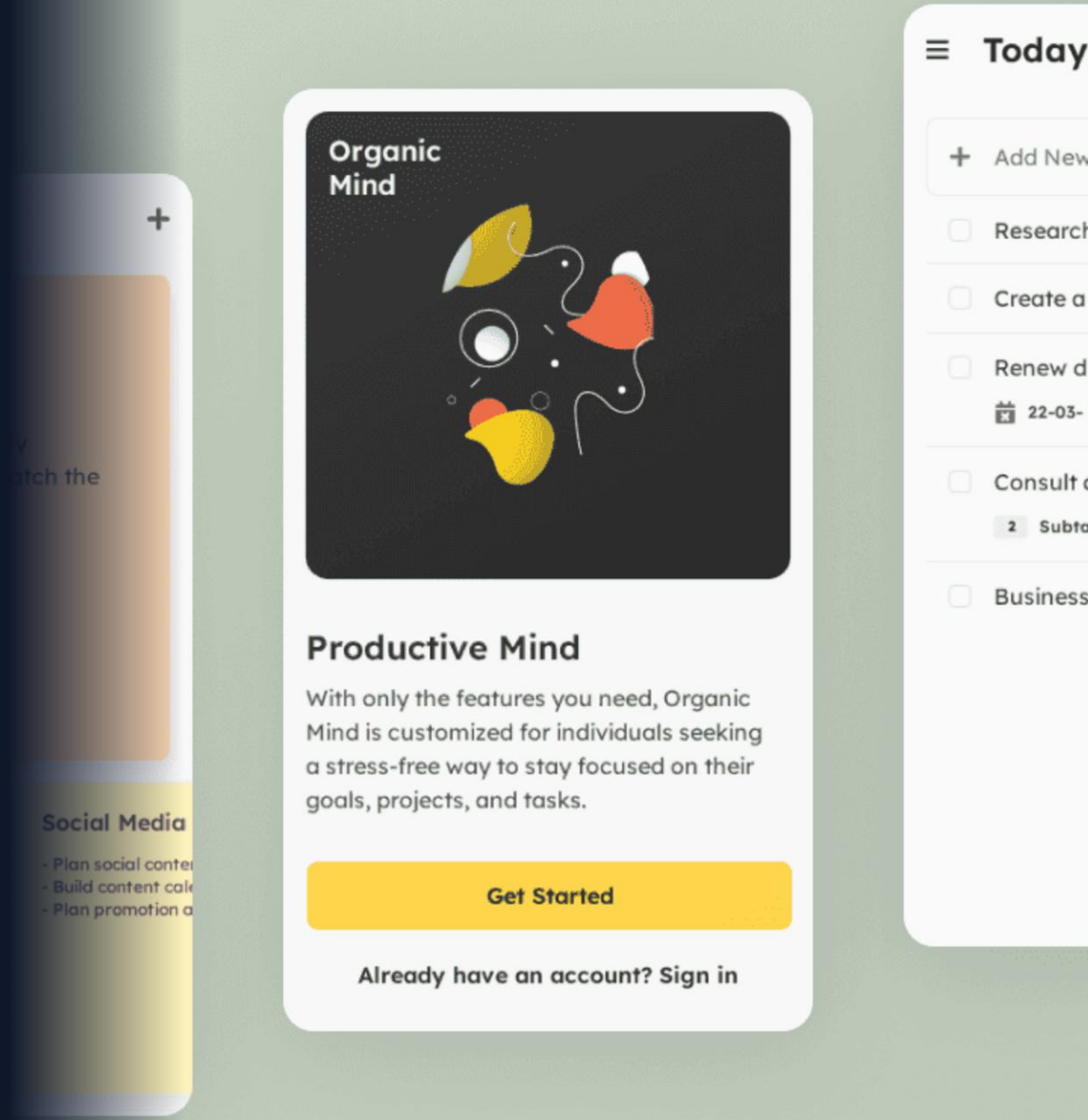


ViewModel plugs into the UI.

Lab: Refactor Task App

We will convert our single-screen app into a robust, navigable application.

- **Step 1:** Create `TaskViewModel`
- **Step 2:** Split UI into `ListScreen` & `DetailScreen`
- **Step 3:** Implement `NavHost`



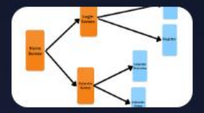
Questions?

Let's refactor our code!

Image Sources

https://uizard.io/static/ad6ed79a8d0a71b0ae07d216a95824f1/a8e47/e33b6844ef5e56ea7e36dc6b9e9b520bf8fa69c0-1440x835.png

Source: uizard.io

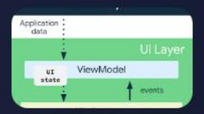


https://miro.medium.com/v2/resize:fit:1400/1*Xhpe8-nz8GhH0CylOhUpCw.png

Source: medium.com

https://miro.medium.com/1*gcB9uaPClZfHQRPQtOjHBA.png

Source: medium.com



https://developer.android.com/static/codelabs/basic-android-kotlin-compose-viewmodel-and-state/img/61eb7bcdcff42227.png

Source: developer.android.com