ChatGPT

Uber RIBs: Simple-Modern Prototype With Jetpack Compose

What We're Gonna Build?

We are gonna build a simple master-detail app that benefits from Uber's RIBs architectural pattern (with Jetpack Compose for the UI). This means structuring our app using **RIBs** (which stands for **Router**, **Interactor**, **Builder**), instead of a traditional MVVM or MVP approach. The goal is to demonstrate how RIBs provides a highly modular and scalable architecture for even a simple feature, and how it integrates with modern UI toolkits like Jetpack Compose.

Why RIBs?

If you take a look at big tech companies over the several years, you will see a pattern in their mobile approach: introducing their own architectural framework. For example:

- **Square** has many of them (often used together) to create a robust architecture: *Molecule*, *Zipline*, *Redwood* (to name a few).
- **Slack** has **Circuit** (a Compose-driven app architecture [1]).
- **Bumble** open-sourced **Appyx** (a navigation and architecture framework).
- **Uber** has **RIBs** (Router-Interactor-Builder, Uber's cross-platform mobile architecture framework [2]).

Aaand Uber has this RIBs. When I realized this type of approach across almost each and every big tech company, of course I questioned the reason why. There are those buzzwords to support the idea of creating whole new architecture frameworks like **scalability**, **maintainability**, etc. etc. But what's really interesting is the same words are being used for simple and plain MVVM, yet somehow these companies are obviously avoiding implementing plain MVVM in their enormous-humongous projects that serve hundreds of millions of users around the world. In my point of view I will explain the reason with one idiom: **Domino Effect** .

I know it seems unrelated, but my focus point is that when you put multiple engineers on the same project and expect them to:

- create the resources in the exact same way,
- put every single file into the exact path they must be (e.g. put UI components that will be used across the project into some kind of core/common package, and feature-specific ones into the related feature's own package),
- search the entire large codebase to see if there is a component or class defined that does the same thing you are about to implement (to avoid duplications),
- put only and only business logic into ViewModels (so that UI changes in the future won't interfere with the ViewModel logic),
- and vice versa: do **not** create multiple sources of truth for UI state,
- ... and so on and so forth.

The list goes on, but you can sense the butterfly effect here that will cause some kind of architectural horror movie or "rewrite from scratch" headaches if that list is not being applied by each and every single engineer working on your codebase. So it is obvious that providing a well-scoped framework for the team is crucial. RIBs helps developers do exactly this: very well-scoped, pluggable mini app-like features that do exactly (and only) what they need to do, with clear boundaries. In fact, Uber created RIBs to address the challenges of large-scale apps with many nested states and a large number of engineers collaborating on the codebase [2]. The RIBs framework emphasizes strong **modularity** and isolation of features, which allows hundreds of engineers to work in parallel on the same app without stepping on each other's toes [3] [4].

There are trade-offs, of course. RIBs introduces more components and up-front planning than simpler patterns. A smaller engineering organization may not need this level of structure, since RIBs can seem verbose compared to MVVM or MVP [5]. However, for Uber's scale (with millions of users and hundreds of developers), these extra rules and patterns prevent the "Domino Effect" of small inconsistencies snowballing into huge problems. In essence, **business logic drives the app** in RIBs, not the view hierarchy. Unlike MVVM where UI components (Activities/Fragments) often dictate structure, a RIB does not even need to have a view for each unit. This means the app's hierarchy is driven by business logic states rather than the UI tree [6]. (This is likely why those same buzzwords of *scalability* and *maintainability* actually *are* achieved in practice with RIBs rather than plain MVVM in these large apps.)

Essentials of RIBs Architecture

As the name suggests, there are three **essential components** when using RIBs:

- **Router:** Manages navigation and the lifecycle of some components. It typically decides which child RIBs to attach or detach in response to events, taking some overhead away from the interactor (so the interactor can focus on business logic).
- **Interactor:** Contains pure business logic. It responds to UI events, performs data fetching or state updates, and decides when to tell the Router to navigate (attach/detach child RIBs).
- **Builder:** Brings RIB components together and builds them. It declares and satisfies necessary dependencies for the RIB (often via a Dependency Injection system). We will use Uber's **Motif** DI library for this, as it helps implement the Builder’s job in the RIB framework.

Those three make up the core of a RIB (Router, Interactor, Builder). There are also a couple of **non-mandatory components** in the RIB pattern:

- **View:** It's just a view (the UI). In our case this will be a Jetpack Compose UI (or it could be an XML layout in a traditional view system). Not every RIB needs its own View; some RIBs can be pure logic.
- **Presenter:** Handles complex UI mappings and view-model transformations. Often you can omit a Presenter if the UI is simple enough; in that case either the Interactor or the View itself can handle any minor state mapping. We will use a `ComposePresenter` class provided by the framework to integrate Compose UI as the view layer for a RIB.

To make this more concrete, let's take a look at a snippet from our app's **Main RIB's Router** 👇 👇 :

```
class MainRouter(
    // (Optional) A View is not mandatory for RIBs, but we use one here.
    view: ComposeView,
    interactor: MainInteractor,
    private val parentView: ViewGroup,
    private val childContent: ChildContent,
) : BasicViewRouter<ComposeView, MainInteractor>(view, interactor) {

    // Keeps references to child routers because we will use attachChild/
detachChild
    // functions, and their signatures require a reference to the child's
router.
    private var listRouter: ListRouter? = null
    var detailRouter: DetailRouter? = null
        private set

    override fun willAttach() {
        super.willAttach()
        // When this RIB is attached, add its view to the parent container
        parentView.addView(view)
        // Set the Jetpack Compose UI content for this RIB's view
        view.setContent { MainView(childContent) }
    }

    override fun willDetach() {
        parentView.removeView(view)
        super.willDetach()
    }

    // ...
}
```

In the code above, the **MainRouter** is a `BasicViewRouter` (a base class provided by the RIBs framework for routers that have a View). We pass in a Jetpack `ComposeView` (which will host our Compose UI) and the `MainInteractor`. The router holds references to its child RIB routers (`listRouter` and `detailRouter`) so that it can attach or detach those children when needed. In `willAttach()`, we explicitly add our ComposeView to the parent container and set its content to the `MainView` composable function. (This is a Compose-specific way to provide a UI, instead of using an XML layout.) By doing this in the Router, we keep the Interactor focused only on business logic. The Router thus manages the **navigation** aspects (which child RIB is shown), while delegating user interactions and data to the Interactor.

Uber Motif (Dependency Injection for RIBs)

Motif is a simple Dependency Injection (DI) library from Uber. It used to depend on generating Dagger code, but currently it does not. It is optimized for nested scopes, which means it plays a strong supportive role when implementing RIBs. RIBs architecture naturally forms a tree of scopes (each RIB is a scope that can have child scopes), and Motif helps fulfill the Builder duties in the RIB framework by providing a way to declare dependencies and instantiate RIB components for each scope. (In fact, Motif is described as "an abstraction on top of Dagger offering simpler APIs for nested scopes" [7] .)

Here is our **Main RIB's Builder** interface defined using Motif 👇 :

```
@motif.Scope
interface MainScope {
    // This is an access method to be called from parent RIBs.
    // (Remember, attaching and detaching a child is done via its router,
    // so we expose the router through the scope.)
    fun router(): MainRouter

    @motif.Objects
    abstract class Objects {
        // Provide the MainRouter, MainInteractor, Presenter, etc.
        abstract fun router(): MainRouter
        abstract fun interactor(): MainInteractor
        abstract fun presenter(): EmptyPresenter
        abstract fun childContent(): MainRouter.ChildContent

        fun view(parentViewGroup: ViewGroup): ComposeView {
            // Create a ComposeView when building this RIB, using the parent's
context
            return ComposeView(parentViewGroup.context)
        }

        // Exposed dependencies are visible to all children under this scope.
        @Expose
        abstract fun itemsStream(): ItemsStream

        // This is a callback for one of the children (List RIB in this case) to
call.
        // It's defined here because the decider of this action must be Main
RIB.
        // (Neither of the children know about each other,
        // so List RIB cannot directly navigate to Detail - it must call back
up.)
        @Expose
        abstract fun onItemClickedListener(interactor: MainInteractor):
ListInteractor.Listener
    }
}
```

In the above code, we define a `MainScope` (using `@motif.Scope`). The nested `@motif.Objects` class declares what objects are provided as part of this scope. This is where we **construct our RIB's pieces** and declare dependencies. For example, `router()` provides the `MainRouter` instance, `interactor()` provides the `MainInteractor`, and so on. The `view()` method shows how we create a `ComposeView` for the RIB using a parent ViewGroup passed in (the parent will be the Root's view container). We also **expose** two things to child scopes: an `ItemsStream` (which presumably holds the list of items to display, coming from a repository), and an `onItemClickedListener` callback. The `@Expose` annotation means these dependencies can be seen by child RIBs (so the List RIB can get `ItemsStream` and the callback listener from its parent Main RIB scope).

Motif generates an implementation of `MainScope` behind the scenes. We don't call `new MainRouter(...)` directly; instead, we'll use a `ScopeFactory.create(MainScope::class.java, deps)` to get an instance of this scope (supplying any external dependencies it needs). This ensures that all the objects declared in `Objects` (router, interactor, etc.) are created and wired together properly, with the dependencies satisfied.

Implementing the Root RIB

Now that you have a little bit of idea of the framework, let's start implementing our RIB hierarchy. We have parent and child RIBs, and our architecture will pass necessary callbacks, dependencies, and UI events between those RIBs. Everything in its own place. No mess. We will implement our **Root RIB** first, as it is the entry point to the application (and the highest scope in our hierarchy, analogous to an "Application scope" in a traditional app).

RootScope

First, let's define the **RootScope**. This Motif scope will contain the Root RIB's objects, and also declare child scopes that can be attached. The RootScope is the largest scope and can contain child scopes for major sections of the app. Here's the RootScope interface:

```
@motif.Scope
interface RootScope {
    fun router(): RootRouter

    // We declare the MainScope here so that the Root RIB can attach the Main
RIB.
    // Essentially, this gives us access to MainScope's factory method.
    fun mainScope(
        parentViewGroup: ViewGroup,
        productsOfflineFirstRepository: ProductsOfflineFirstRepository
    ): MainScope

    fun dataScope(): DataScope

    @motif.Objects
    abstract class Objects {
        abstract fun router(): RootRouter
        abstract fun interactor(): RootInteractor
        abstract fun presenter(): EmptyPresenter

        fun view(parentViewGroup: ViewGroup): RootView {
            // The RootView is a simple FrameLayout that will serve as
            // a container for child views.
            return RootView(parentViewGroup.context)
        }

        @Expose
        fun context(activity: RibActivity): Context {
            // Expose the application context to the scope (could be used by
child scopes)
            return activity.applicationContext
        }
    }
}
```

A few things to notice here: The `RootScope` exposes a method `mainScope(...)` which takes a parent `ViewGroup` and a `productsOfflineFirstRepository`. This is how the Root RIB will create (and attach) the Main RIB. The `mainScope` function returns a `MainScope` (which we defined earlier), and from that we can get the `MainRouter` via `MainScope.router()`. We also have a `dataScope()` which presumably provides data-layer dependencies (like a repository) that might be used by child scopes (perhaps `DataScope` is where `ProductsOfflineFirstRepository` comes from). The `Objects` class declares how to build RootRouter, RootInteractor, etc., similar to what we saw in MainScope. We also expose a `context` (the application Context) for any child that might need it.

RootRouter

Now let's implement the **RootRouter**, which is the Router for the Root RIB. Its job is to attach the Main RIB when the app starts, and possibly handle detaching it or other global navigation events (like a back press that should exit the app). Here's our RootRouter:

```
class RootRouter(
    view: RootView,
    interactor: RootInteractor,
    private val scope: RootScope,
) : BasicViewRouter<RootView, RootInteractor>(view, interactor) {
    var mainRouter: MainRouter? = null
        private set

    override fun willAttach() {
        // When the Root RIB is attached, we immediately attach the Main RIB.
        attachMain()
    }

    override fun willDetach() {
        // Clean up by detaching the Main RIB when Root is torn down.
        detachMain()
    }
    private fun attachMain() {
        if (mainRouter == null) {
            // Use the RootScope to create the MainScope (and thus MainRouter)
            mainRouter = scope.mainScope(view,
scope.dataScope().productRepository()).router()
                .also { attachChild(it) }
        }
    }

    private fun detachMain() {
        mainRouter?.let {
            detachChild(it)
            mainRouter = null
        }
    }

    fun detachDetail() {
        // This method is called to handle a back press when Detail RIB is
active.
        mainRouter?.detailRouter?.let {
            mainRouter?.detachDetail()  // tell MainRouter to detach its Detail
child
            mainRouter?.attachList()    // and re-attach the List child to go
back to list
        }
    }
}
```

In `RootRouter`, we override `willAttach()` to automatically attach the Main RIB as soon as Root attaches (so the Main screen appears when the app launches). The `attachMain()` method uses the `scope.mainScope(...)` we defined to get a new **MainScope**. Notice we pass `view` (which is the RootView, a container) and `scope.dataScope().productRepository()` (pulling a product repository from a DataScope, likely providing data) into `mainScope()`. This returns a `MainScope`, from which we call `.router()` to get the `MainRouter` instance. We then call `attachChild(it)` to attach the MainRouter as a child of RootRouter. This is how RIBs nests: Root attaches Main as a child RIB. We keep a reference to the `mainRouter` so we can address it later (like detaching or calling functions on it).

The `detachDetail()` function in RootRouter is interesting. You may wonder: why is RootRouter concerned with "Detail screen"? This is because in RIBs, navigation is handled in a **reactive** way with explicit attach/detach calls (there is no automatic back stack). The developer is responsible for handling back presses. In our app, the RootInteractor will intercept the back button. If the Detail RIB is currently attached (meaning the user is on the detail screen), RootInteractor will call `router.detachDetail()`. That function (defined in MainRouter, as we'll see) will detach the Detail RIB. After that, we call `mainRouter?.attachList()` to re-attach the List RIB, effectively going back to the list view. In summary, RootRouter's `detachDetail()` is a convenience to delegate back-navigation to the MainRouter (since MainRouter knows how to remove its Detail child and bring back the list).

Note: In RIBs architecture, there's no built-in back stack like in traditional Android Fragments. The navigation state is represented by which RIBs are attached. Handling back presses means detaching the appropriate RIB manually in code. Uber's engineers describe this as reactive navigation with a tree of RIBs instead of an OS-managed backstack [8].

Figure: A RIBs tree diagram illustrating how different RIB components (like "Root", "Logged Out", "Logged In", etc.) can be structured in a hierarchical state tree [8]. In our case, the Root RIB attaches the Main RIB as its child; the Main RIB can attach either the List or Detail RIB as children, and so on, forming a tree. The app's active state corresponds to which RIBs are currently attached.

RootView and RootInteractor

The **RootView** is trivial in this case: it's just a `FrameLayout` that we use as a container for the UI of child RIBs. We created it in the RootScope (`RootView(parentViewGroup.context)`). There's not much more to it, so we won't dwell on it.

The **RootInteractor** is the Interactor for the Root RIB. Its main responsibility in our simple app is to handle the back button logic we discussed:

```
class RootInteractor(presenter: EmptyPresenter)
    : BasicInteractor<EmptyPresenter, RootRouter>(presenter) {

    override fun handleBackPress(): Boolean {
        // If a Detail RIB is attached, handle back press by detaching Detail.
        val isDetailAttached = router.mainRouter?.detailRouter != null
        if (isDetailAttached) {
            router.detachDetail()
            return true  // indicate that we handled the back press
        }
        return false  // no special handling, propagate back press (e.g. exit
app)
    }
}
```

RIBs provides the `handleBackPress()` hook. Here, we simply check if the Detail RIB is currently attached (i.e., `detailRouter` in the MainRouter is not null). If so, we call `router.detachDetail()` (which will pop the detail screen and show the list again), and return `true` to consume the back press. If no detail is open, we return false, which allows the back press to bubble up (which might close the app if at root). This is how we implement a custom back navigation logic using RIBs.

RootActivity

Finally, special to the Root RIB is the **RibActivity**. This is provided by the RIBs framework (likely as part of the integration on Android). We need to override a method `createRouter()` to provide the entry point to our RIB framework. Essentially, this is where we create the RootRouter (via the RootScope):

```
class RootActivity : RibActivity() {
    override fun createRouter(parentViewGroup: ViewGroup): ViewRouter<*, *> {
        return ScopeFactory.create(Parent::class.java)
            .rootScope(this, findViewById(android.R.id.content))
            .router()
    }

    @motif.Scope
    interface Parent : Creatable<NoDependencies> {
        fun rootScope(@Expose activity: RibActivity, parentViewGroup:
ViewGroup): RootScope
    }
}
```

In `createRouter`, we use `ScopeFactory.create(Parent::class.java)` to get a factory (or scope) that can create our RootScope. We call `rootScope(this, findViewById(android.R.id.content)).router()` on it. Let's break that down: The `Parent` interface is a Motif scope interface defined inside RootActivity (for convenience). It basically says: if we have an Activity (RibActivity) and a root `ViewGroup`, we can create a `RootScope`. The Activity passes itself (`this`) and the content view (`android.R.id.content` is the root ViewGroup of the activity window) into the RootScope. From the RootScope, we call `.router()` to get the RootRouter instance. That is returned to the RibActivity which will handle attaching it.

This `RootActivity` is where the Root RIB is actually launched. After this point, the RootRouter's `willAttach` runs, which attaches the Main RIB, and the app UI is on screen.

Implementing the Main RIB

Now that we provided a valid entry point to our application (the Root RIB), next we introduce the **Main RIB**. The Main RIB will control the two child RIBs in our feature: the List RIB and the Detail RIB. It acts as a mediator, providing necessary application data to its children and dispatching user events (like navigation) to the right place.

We already saw the `MainScope` above in the Motif section, and a glimpse of `MainRouter`. Let's go deeper into how the Main RIB works.

MainRouter

Here is the complete `MainRouter` code again, now with its child-attachment methods:

```
class MainRouter(
    view: ComposeView,
    interactor: MainInteractor,
    private val parentView: ViewGroup,
    private val childContent: ChildContent,
) : BasicViewRouter<ComposeView, MainInteractor>(view, interactor) {

    private var listRouter: ListRouter? = null
    var detailRouter: DetailRouter? = null
        private set

    override fun willAttach() {
        super.willAttach()
        parentView.addView(view)
        // Direct UI setup - no presenter needed for simple container:
        view.setContent {
            MainView(childContent)
        }
    }

    internal fun attachList() {
        // Prepare dependencies needed by the List RIB
        val listScopeDependencies = object : ListScopeDependencies {
            override fun slot(): MutableState<@Composable () -> Unit> {
                return childContent.fullScreenSlot
            }
            override fun onItemClickedListener(): ListInteractor.Listener {
                // The MainInteractor implements ListInteractor.Listener to
handle item clicks
                return interactor
            }
            override fun itemStream(): ItemsStream {
                return interactor.itemsStream
            }
        }
        // Create the ListScope using the dependencies, then get its router
        listRouter = ScopeFactory.create(ListScope::class.java,
listScopeDependencies).router()
        listRouter?.let { attachChild(it) }
    }

    internal fun attachDetail(product: Product) {
        // Prepare dependencies for Detail RIB
        val detailScopeDependencies = object : DetailScopeDependencies {
            override fun slot(): MutableState<@Composable () -> Unit> {
                return childContent.fullScreenSlot
            }
            override fun product(): Product {
                return product
            }
        }
        detailRouter = ScopeFactory.create(DetailScope::class.java,
detailScopeDependencies).router()
        detailRouter?.let { attachChild(it) }
    }

    internal fun detachList() {
        listRouter?.let {
            detachChild(it)
            listRouter = null
        }
    }

    internal fun detachDetail() {
        detailRouter?.let {
            detachChild(it)
            detailRouter = null
        }
    }

    // ChildContent holds a Compose slot that children use to render their UI in
MainView.
    class ChildContent {
        internal var fullScreenSlot: MutableState<(@Composable () -> Unit)> =
mutableStateOf({})
    }
}
```

MainRouter responsibilities: The MainRouter manages which child (List or Detail) is currently attached to it. We have two key methods: `attachList()` and `attachDetail(product)` which attach the respective child RIBs.

- In `attachList()`: we create a `listScopeDependencies` object that implements `ListScopeDependencies`. This is providing the dependencies that the List RIB (scope) requires:
- A `slot()` which is a `MutableState<() -> Unit>` – this is essentially a Compose slot where the list UI will be rendered. Our `MainView` composable uses this `childContent.fullScreenSlot` to display either the list or detail UI.
- An `onItemClickedListener()` which returns a `ListInteractor.Listener`. We return `interactor` here because our `MainInteractor` implements that interface (so MainInteractor will handle item clicks from the list).
- An `itemStream()` which provides the stream of items (products) to display, coming from MainInteractor.

Then we call `ScopeFactory.create(ListScope::class.java, listScopeDependencies)` to create the ListScope with those dependencies. We grab its router via `.router()` and store it in `listRouter`. Finally, `attachChild(it)` is called to attach the ListRouter as a child of MainRouter. This will mount the List RIB's UI into the `childContent.fullScreenSlot` we provided (as you'll see in List RIB implementation).

- In `attachDetail(product)`: similarly, we create `detailScopeDependencies` for the Detail RIB:
- The same `slot()` (we want the detail screen to occupy the same Compose slot in the UI that the list was using).
- A `product()` which supplies the specific product data the detail screen should show.

We create the DetailScope, get its router, assign to `detailRouter`, and attach it as child.

We also have `detachList()` and `detachDetail()`, which simply detach and nullify the respective child routers. These are called when we navigate away from a screen. For example, when an item is clicked, we likely detach the List and attach the Detail (to avoid overlapping UIs). And when the user hits back, we detach Detail and re-attach List.

One other piece is `ChildContent`: this is a small inner class holding `fullScreenSlot`. This slot is a Compose `MutableState` that represents the content to show in the MainView. Basically, `MainView` (a composable) will observe `childContent.fullScreenSlot` and render whatever composable function it contains (either the List UI or the Detail UI). By providing the same slot to both List and Detail RIBs, we ensure they render in the same place in the UI. This is a pattern to integrate Compose with RIBs: instead of each RIB having its own separate Android View container, we use a composable content slot to swap UIs.

MainInteractor

The **MainInteractor** is where the business logic for the Main RIB resides. It fetches the data (the list of products), populates the `ItemsStream`, and handles the event when an item is clicked (to trigger navigation to detail). Our MainInteractor also implements `ListInteractor.Listener` so it can receive the click callback from the List RIB. Let's see the code:

```kotlin
class MainInteractor(
    presenter: EmptyPresenter,
    private val productsOfflineFirstRepository: ProductsOfflineFirstRepository,
    val itemsStream: ItemsStream,
) : BasicInteractor<EmptyPresenter, MainRouter>(presenter),
ListInteractor.Listener {

    override fun didBecomeActive(savedInstanceState: Bundle?) {
        super.didBecomeActive(savedInstanceState)
        // When Main RIB becomes active, start loading products data.
        coroutineScope.launch {
            productsOfflineFirstRepository.getAggregatedProducts()
                .retry {
                    delay(RETRY_INTERVAL)
                    true  // keep retrying on failure every 3 seconds
                }
                .onEach {
                    itemsStream.update(it)  // update the stream with new list
                }
                .launchIn(coroutineScope)
        }
        // Observe the itemsStream: once we have any items, attach the List RIB.
        itemsStream
            .observe()
            .onEach { items ->
                if (items.isNotEmpty()) {
                    router.attachList()
                }
            }
            .launchIn(coroutineScope)
    }

    override fun onItemClicked(id: Int) {
        // This is called when List RIB's view triggers an item click.
        coroutineScope.launch {
            val product = productsOfflineFirstRepository.getProductById(id)
            router.detachList()
            router.attachDetail(product)
        }
    }

    companion object {
        const val RETRY_INTERVAL = 3000L
    }
}
```

Let's break down what happens in `didBecomeActive`: - We launch a coroutine to fetch products from the repository. The repository might attempt a network call or load from cache (hence "OfflineFirst"). We call `getAggregatedProducts()`, then use a `.retry` operator with a delay to keep retrying every 3 seconds if it fails (network error, etc.). Each time we get data, we call `itemsStream.update(it)`. So `itemsStream` is some kind of observable data holder (perhaps a simple wrapper around a `StateFlow` or similar) that the List RIB will use to get the list of products. - Separately, we subscribe to `itemsStream.observe()`. When items come in, if the list is not empty, we call `router.attachList()`. This ensures that as soon as we have data, the List RIB is attached and the user sees the list. The `attachList` we defined in MainRouter will mount the list UI.

Basically, MainInteractor orchestrates the data flow: fetch products -> update stream -> once data available -> show list.

The `onItemClicked(id: Int)` is the callback from the List RIB (because we passed `MainInteractor` as the `ListInteractor.Listener`). When an item is clicked, we again use a coroutine to fetch the full product details by ID (maybe the list only had summary info and we need more details). Once we have the `product`, we call `router.detachList()` and then `router.attachDetail(product)`. This sequence will remove the list UI and add the detail UI.

Notice how **navigation is triggered by the Interactor** (business logic), not by the view directly. This is a key aspect of RIBs: the view (List) simply reports an event (item clicked) to its interactor, which bubbles up to the parent (MainInteractor via the Listener interface). The MainInteractor then decides to navigate (because perhaps it also fetched additional data or had logic to decide which detail to show) and instructs the router accordingly. This keeps our flow unidirectional and decoupled.

With that, we have prepared the foundation of our list-detail mechanism with RIBs. What's left are our **List RIB** and **Detail RIB**, and they will be very simple in comparison to Root/Main.

Implementing the List RIB

The **List RIB** represents the screen that shows a grid or list of product items. This RIB will have a Compose UI that displays the list, and will notify its parent when an item is clicked (via the listener interface).

First, we define the **ListScope** (the DI scope for the List RIB). Since the List RIB is a child of Main, its scope will receive dependencies from the Main RIB (as we saw when calling `ScopeFactory.create(ListScope::class.java, listScopeDependencies)` in MainRouter).

```
interface ListScopeDependencies {
    fun slot(): MutableState<(@Composable () -> Unit)>
    fun onItemClickedListener(): ListInteractor.Listener
    fun itemStream(): ItemsStream
}

@motif.Scope
interface ListScope : Creatable<ListScopeDependencies> {
    fun router(): ListRouter

    @motif.Objects
    abstract class Objects {
        abstract fun router(): ListRouter
        abstract fun interactor(): ListInteractor

        fun presenter(
            stateStream: StateStream<List<ProductViewModel>>,
            eventStream: EventStream<ListEvent>,
        ): ComposePresenter {
            return object : ComposePresenter() {
                override val composable =
                    @Composable {
                        ListView(
                            // Observe state stream to get current list of
ProductViewModel
                            stateStream.observe().collectAsState(initial =
stateStream.current()),
                            eventStream,
                        )
                    }
            }
        }

        fun eventStream() = EventStream<ListEvent>()
        fun stateStream() = StateStream(emptyList<ProductViewModel>())
    }
}
```

A few notes on this: - `ListScopeDependencies` is an interface listing what dependencies the ListScope needs. This matches what we provided in `listScopeDependencies` object earlier: - a `slot()` for UI output, - a `onItemClickedListener()` for callbacks, - an `itemStream()` for the data source. - `ListScope` itself extends `Creatable<ListScopeDependencies>` which means Motif will create it using those dependencies. - In `Objects`, we provide a custom `ComposePresenter`. This is interesting: we create an anonymous `ComposePresenter` where we override `composable` with a Composable function. That composable uses our `ListView` composable, passing it the current state from a `StateStream` and the `EventStream`. Essentially, the **Presenter** here is producing the UI by combining a state stream and an event stream: - `stateStream` holds a list of `ProductViewModel` (the display models for products). - `eventStream` will carry UI events (like item clicks). - `ListView` is a composable (we would implement it to show a grid or list of items, omitted here for brevity). - We also provide an `eventStream()` and `stateStream()` instances. These are like local data pipes for the List RIB: - `stateStream` starts with an empty list. - `eventStream` will emit `ListEvent` (likely an event class with something like `OnItemClick(id: Int)`).

The `ListRouter` is extremely simple in this case:

```
class ListRouter(
    presenter: ComposePresenter,
    interactor: ListInteractor,
    slot: MutableState<(@Composable () -> Unit)>,
) : BasicComposeRouter<ListInteractor>(presenter, interactor, slot)
```

We subclass `BasicComposeRouter`, which likely knows how to take a ComposePresenter and put its UI into the given `slot`. We pass the `slot` (which came from MainRouter's ChildContent) so that when this router is attached, the `presenter.composable` (the `ListView`) will be rendered in that slot.

Now the **ListInteractor**:

```
class ListInteractor(
    presenter: ComposePresenter,
    private val itemsStream: ItemsStream,
    private val eventStream: EventStream<ListEvent>,
    private val stateStream: StateStream<List<ProductViewModel>>,
    private val listener: Listener
) : BasicInteractor<ComposePresenter, ListRouter>(presenter) {

    override fun didBecomeActive(savedInstanceState: Bundle?) {
        super.didBecomeActive(savedInstanceState)
        // Subscribe to UI events from ListView
        eventStream
            .observe()
            .onEach {
                when (it) {
                    is ListEvent.OnItemClick -> {
                        listener.onItemClicked(it.id)
                    }
                }
            }
            .launchIn(coroutineScope)

        // Subscribe to the ItemsStream (from parent) to get product list
updates
        itemsStream
            .observe()
            .onEach { items ->
                // Convert domain Product to ProductViewModel for display
                val productViewModels = items.map {
                    ProductViewModel(
                        id = it.id,
                        title = it.title,
                        description = it.description,
                        price = it.price.toString(),
                        imageUrl = it.thumbnail,
                        rating = it.rating.toString(),
                        category = it.category,
                        reviewCount = it.reviews.size.toString()
                    )
                }
                stateStream.dispatch(productViewModels)
            }
            .launchIn(coroutineScope)
    }

    interface Listener {
        fun onItemClicked(id: Int)
    }
}
```

What happens here: - When the ListInteractor becomes active, it subscribes to `eventStream.observe()`. These events come from the UI (ListView). We expect that the ListView, when an item is clicked, will send a `ListEvent.OnItemClick(id)` into the `eventStream`. The interactor receives it and calls `listener.onItemClicked(it.id)`. Remember, `listener` was provided by MainInteractor (which implements this interface), so this call actually invokes `MainInteractor.onItemClicked(id)`, causing the navigation to detail. - We also subscribe to the parent-provided `itemsStream`. This `itemsStream` is updated by MainInteractor when product data is fetched. On each update, we map the domain `Product` objects to `ProductViewModel` objects that are more convenient for the UI (strings for price, etc.). Then we dispatch this list into our local `stateStream`. Because our ComposePresenter's composable is collecting `stateStream.observe()`, the UI will automatically recompose with the new list of products. This demonstrates a unidirectional data flow: Main provides data -> ListInteractor transforms it -> Presenter/UI displays it.

The ListView composable (not shown in code) would simply display the list of `ProductViewModel` items and on user click, send `ListEvent.OnItemClick(id)` into the `eventStream`. We have omitted the UI implementation since it's not the focus here and there's already a lot of boilerplate (which is one of the disadvantages of RIBs, as you might notice).

Implementing the Detail RIB

Finally, the **Detail RIB** shows details of a single product. This one is even simpler than the List RIB, because it doesn't have complex interactions – it's mostly just displaying data.

We define **DetailScope** similarly:

```
interface DetailScopeDependencies {
    fun slot(): MutableState<(@Composable () -> Unit)>
    fun product(): Product
}

@motif.Scope
interface DetailScope : Creatable<DetailScopeDependencies> {
    fun router(): DetailRouter

    @motif.Objects
    abstract class Objects {
        abstract fun router(): DetailRouter
        abstract fun interactor(): DetailInteractor

        fun presenter(
            stateStream: StateStream<Product>,
        ): ComposePresenter {
            return object : ComposePresenter() {
                override val composable =
                    @Composable {
                        DetailView(
                            stateStream.observe().collectAsState(initial =
stateStream.current()),
                        )
                    }
            }
        }

        fun stateStream(product: Product) = StateStream(product)
    }
}
```

- `DetailScopeDependencies` requires a `slot()` (where to render the UI, provided by Main) and a `product()` (the product to display).
- `DetailScope` provides a `DetailRouter` and `DetailInteractor`.
- The `presenter` here creates a composable that simply takes a `stateStream` of `Product` and passes it to a `DetailView` composable. We initialize the state stream with the `product` (so `stateStream.current()` will have the product). The UI can just show the details of this product. No events to handle here, presumably.
- `stateStream(product)` creates a StateStream holding that product. This is a convenient way to treat the product as state for the Compose UI.

The **DetailRouter** is analogous to ListRouter:

```
class DetailRouter(
    presenter: ComposePresenter,
    interactor: DetailInteractor,
    slot: MutableState<(@Composable () -> Unit)>,
) : BasicComposeRouter<DetailInteractor>(presenter, interactor, slot)
```

It just takes the presenter and interactor and a Compose slot, and the base class will render the `DetailView` into the given slot.

The **DetailInteractor** doesn't have much to do in our case:

```
class DetailInteractor(
    presenter: ComposePresenter,
) : BasicInteractor<ComposePresenter, DetailRouter>(presenter)
```

We don't need to override anything here; there's no special business logic once the detail screen is shown. (If there were actions on the detail screen, like "add to cart" or something, those would be handled in a similar fashion with an EventStream and callbacks, but our prototype keeps it simple.)

And that's it! When MainRouter attaches the Detail RIB (via `attachDetail(product)`), the DetailView will show that product's information. When the user presses back, RootInteractor will call `RootRouter.detachDetail()`, which triggers MainRouter to detach the Detail RIB and re-attach the List RIB, going back to the list.

Conclusion

RIBs in its essence is a very helpful framework to separate layers very strictly. Each RIB is like a mini-app with a well-defined scope of responsibilities, which can be plugged into the larger app without messy couplings. As we saw, business logic (Interactors) are decoupled from navigation logic (Routers), and dependency injection (Builders via Motif) ensures each piece only knows about what it needs. This strict separation and structure greatly benefit large projects with many engineers – it prevents the architecture from devolving into chaos as the team grows [3]. In our simple example, it might feel like overkill, but at scale this approach pays off in terms of maintainability and testability (Uber noted that RIBs allowed hundreds of engineers to work on the app with hundreds of RIB modules in parallel [9]).

However, as in every good thing, RIBs has the disadvantage of having **so much boilerplate**. You probably noticed the amount of code and interfaces we had to write for even a basic master-detail flow. Indeed, RIBs is more verbose than MVVM, and not every project will justify this cost [5]. One can handle this overhead by using code generation tools (RIBs comes with some tooling) or creating file templates, and even AI-assisted code generation, to quickly scaffold the repetitive parts. Once the patterns are set up, adding new features in a RIBs architecture becomes more systematic and less error-prone.

In the end, the decision to use an architecture like RIBs comes down to project scale and team needs. For a solo developer or a small app, plain MVVM might be perfectly fine (and faster to write). But for an app that is expected to grow huge (many features, large team), the **structured, scalable approach** of RIBs (or similar frameworks by other companies) can prevent the *"rewrite headaches"* in the long run. The example we built here is a simple prototype, but it demonstrates the core ideas behind RIBs.

You can access the full source code from this link (for reference and deeper exploration of the implementation). Happy RIBbing!

---

[1] Circuit
https://slackhq.github.io/circuit/

[2] [6] [7] [9] GitHub - uber/RIBs: Uber's cross-platform mobile architecture framework - Android Repository
https://github.com/uber/RIBs

[3] [4] [5] [8] Architecting Uber's New Driver App in RIBs | Uber Blog
https://www.uber.com/blog/driver-app-ribs-architecture/