Chapter 5: Adding Navigation in Compose With Jetpack Navigation

In this chapter, we'll focus on a core Jetpack library, the Navigation component. This library is essential to us since it allows us to easily navigate between application screens.

So far, we have only created a screen in our Restaurants application, where we displayed a list of diners. It's time to step up the game and add another screen to our application!

In the first section, *Introducing the Jetpack Navigation component*, we will explore the basic concepts and elements of the Navigation component. In the second section, *Creating a new Compose-based screen*, we will create a new screen to display the details of a specific restaurant and realize that we don't know how to navigate to it.

In the third section, *Implementing navigation with Jetpack Navigation*, we will add the Navigation component to the Restaurants application and use it to navigate to the second screen. Finally, in the *Adding support for deep links* section, we will create a deep link to our newly created screen and make sure that our application knows how to handle it.

To summarize, in this chapter we're going to cover the following main topics:

- Introducing the Jetpack Navigation component
- Creating a new Compose-based screen
- Implementing navigation with Jetpack Navigation
- Adding support for deep links

Before jumping in, let's set up the technical requirements for this chapter.

Technical requirements

Building Compose-based Android projects with Jetpack Navigation usually requires your day-to-day tools. However, to follow along smoothly, make sure you have the following:

- The Arctic Fox 2020.3.1 version of Android Studio. You can also use a newer Android Studio version or even Canary builds, but note that the IDE interface and other generated code files might differ from the ones used throughout this book.
- Kotlin 1.6.10 or newer plugin installed in Android Studio
- The Restaurants app code from the previous chapter

The starting point for this chapter is represented by the Restaurants application developed in *Chapter 4*, *Handling Async Operations with Coroutines*. If you haven't followed the implementation from the previous chapter, access the starting point for this chapter by navigating to the **`Chapter_04`** directory of the repository and importing the Android project named **`chapter_4_restaurants_app`**.

To access the solution code for this chapter, navigate to the **`Chapter_05`** directory: https://github.com/PacktPublishing/Kickstart-Modern-Android-Development-with-Jetpack-and-Kotlin/tree/main/Chapter_05/chapter_5_restaurants_app.

Introducing the Jetpack Navigation component

The **Navigation** component is Jetpack's solution to navigation within Android apps. This library allows you to easily implement navigation between the screens of your application.

To promote a predictable user experience and consistent manner of handling app flows, the Navigation component adheres to a set of principles. The two most important principles are as follows:

- The application has a fixed start **destination** (screen) – this allows the application behavior to be predictable because the app will always present this destination first, no matter where it is being launched from.

In our Restaurants application, we plan to set the start destination as our existing screen with the list of restaurants (represented by the `RestaurantsScreen()` composable function). In other words, this is the first screen that the user will always see when launching the app from the Android launcher screen.

- The navigation state is defined as a stack of destinations, often called the **back stack**. When the app is initially started, the stack will contain the app's start destination – let's call this *Screen A*. If you navigate from *Screen A* to *Screen B*, *B* will be added on top of the stack. This applies when navigating to *Screen C* too. To better understand how the back stack works, let's try to illustrate it in such a scenario:

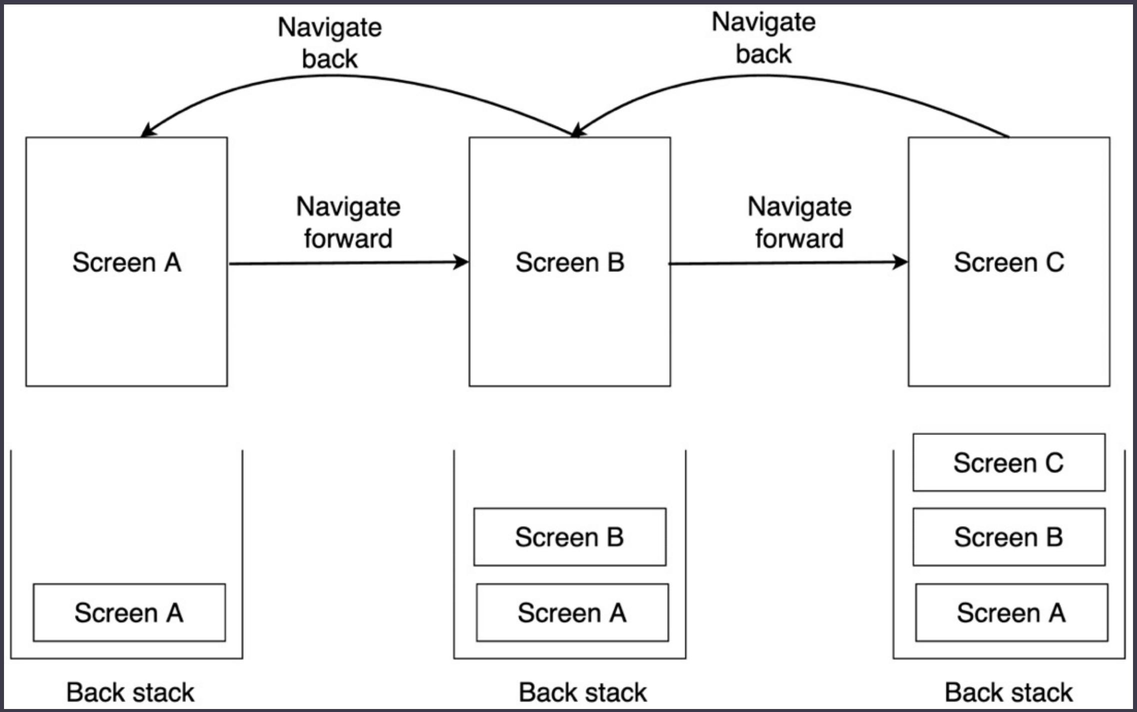


Figure 5.1 – Back stack evolution of screens while the user navigates within the app

At the top of the back stack, you will always have the current screen the user is at right now. When navigating back to the previous screen, the top of the back stack is popped, just as we did in *Figure 5.1*, where navigating from *Screen C* to *Screen B* resulted in the pop of *Screen C* from the back stack.

All these operations are always done at the top of the stack, while the back of the stack will always contain the fixed start destination – in our case, *Screen A*.

The Navigation component takes care of handling the back stack operations behind the scenes for us.

NOTE

Initially, the Navigation component specialized in offering navigation mainly between `Fragment` *components. Today, the library also supports Compose and the navigation between composable functions.*

Apart from following clear principles when it comes to UI navigation, the Navigation component has three main constituent elements:

- **Navigation graph**: The core source of information related to navigation within your app. In the navigation graph, you define all the destinations as well as the possible paths that the user can take throughout the app to achieve different tasks.
- **NavHost**: A container composable function that will display the composable destinations. As the user navigates between different destinations, the content of the navigation host is swapped and recomposed.
- **NavController**: A stateful object that handles the navigation between composable screens and is, therefore, in charge of propagating updates inside the destinations back stack. The navigation controller sets

the correct destinations to `NavHost` as the user starts navigating between screens.

Now, when you implement the Navigation component in your Compose-based Android app, you will gain a lot of benefits. The following lists some examples:

- You don't need to handle the complexity of navigation between composable functions. The library does that out of the box for you.
- You don't need to handle *Up* or *Back* actions on your own. If you press the system's **Back** button, the library will automatically pop the current destination from the back stack and send the user to the previous destination.
- You benefit from scoped `ViewModel` components to a specific Navigation graph or destination. This means that the `ViewModel` instance used by a composable destination will live for as long as the composable screen does.
- You don't need to implement deep links from scratch. Deep links allow you to directly navigate to a specific destination within the app without having to traverse the entire path of screens that get you there. We will see how they work in the *Adding support for deep links* section of this chapter.

Now that we have a basic overview of the elements and advantages of using Jetpack Navigation, it's time to create a new screen so we can implement navigation in our Restaurants application.

Creating a new Compose-based screen

Real-world applications are required to display a lot of content, so one screen probably won't suffice. So far, our Restaurants application features a simple screen where all the restaurants that we receive from our remote database are displayed.

Let's practice all the skills we've learned so far by creating a new screen that will display the details of a particular restaurant. The plan is that when users press on a particular restaurant from the list inside our **`RestaurantsScreen()`** composable screen, we should take them to a new details screen for that particular restaurant.

Yet to perform navigation between two screens, we need first to build the second screen. Unlike with the first composable screen, it's time to change our tactic and build it from top to bottom. Let's build this second feature first by defining the network request, then executing it inside its own **`ViewModel`**, and finally creating the composable UI that will consume the data, as follows:

- Defining the HTTP request for the contents of a restaurant
- Getting the contents of a specific restaurant
- Building the restaurant details screen

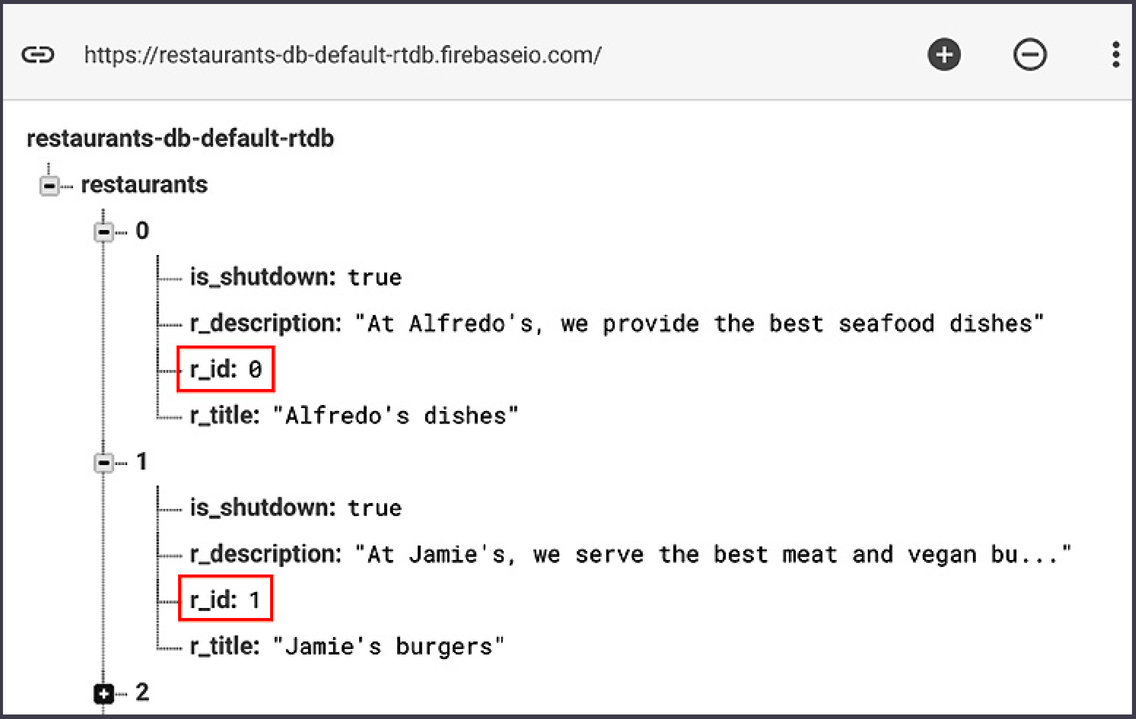
Let's start!

Defining the HTTP request for the contents of a restaurant

We need to know how to obtain the data for our new restaurant details screen. Instead of relying on the previously retrieved data (the list of restaurants), we want to make every screen in our application as independent as possible. This way, we design our application to easily support deep links and we better defend ourselves from events such as a system-initiated process death.

That's why we will build this new screen so that it gets its own content. In other words, in the new screen, we will get the details for a particular restaurant from the same database where we've obtained the list of restaurants. But how will we do that?

Remember that the restaurants within our Firebase database have a unique **`Integer`** identifier field called **`r_id`**, as shown in the following screenshot:



```
https://restaurants-db-default-rtdb.firebaseio.com/
restaurants-db-default-rtdb
  restaurants
    0
      is_shutdown: true
      r_description: "At Alfredo's, we provide the best seafood dishes"
      r_id: 0
      r_title: "Alfredo's dishes"
    1
      is_shutdown: true
      r_description: "At Jamie's, we serve the best meat and vegan bu..."
      r_id: 1
      r_title: "Jamie's burgers"
    2
```

Figure 5.2 – Identifying the unique identifier field for restaurants in Firebase

We can use this field to get the details of one specific restaurant. And since **`r_id`** is mapped to the **`id: Int`** field of the **`Restaurant`** object, this means that when the user presses on a restaurant in our **`RestaurantsScreen`** composable, we can forward the **`id`** value to the second screen.

In the second screen, we will execute an API request to our Firebase REST API and pass the value of the unique ID of the restaurant within our app that corresponds to the **`r_id`** identifier of the restaurant inside the remote database.

The Firebase REST API has us covered for such cases. If we want to get the details of one element from the restaurants JSON content, we must append two query parameters to the same URL used to retrieve the entire restaurants list:

- `orderBy=r_id` to instruct Firebase to filter the elements by their `r_id` field.
- `equalTo=2` to let Firebase know the value of the `r_id` field of the restaurant element that we're looking for – in this case `2`.

To practice, place in your browser address bar the Firebase URL that you've used to get the restaurants until now and append the previous two query parameters as follows:

```
https://restaurants-db-default-
rtdb.firebaseio.com/restaurants.json?
orderBy="r_id"&equalTo=2
```

If you access your link, the response will, unfortunately, look like this:

```
{ "error" : "Index not defined, add \".indexOn\":
\"r_id\", for path \"/restaurants\", to the rules" }
```

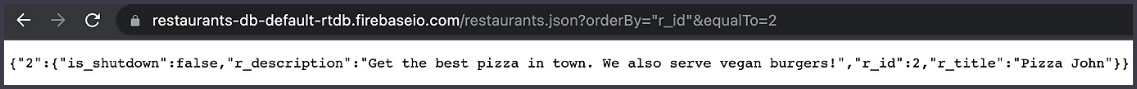
Firebase needs some additional configuration so that we can get the details of only one element within the list, so let's do that now:

1. Navigate to your Firebase console and log into your Google account by accessing this link: https://console.firebase.google.com/.
2. From the list of Firebase projects, select the one you've previously created to store the restaurants.
3. In the left menu, expand the **Build** tab, search for **Realtime Database**, and then select it.
4. Move away from the preselected **Data** tab and select the **Rules** tab.
5. We need to allow Firebase to index the restaurants based on their `r_id` field, so update the write `rules` as follows:

```
{
    "rules": {
        ".read": "true",
        ".write": "true",
        "restaurants": {
          ".indexOn": ["r_id"]
         }
    }
}
```

By doing so, we've instructed Firebase that the JSON array content located at the **`restaurants`** node can be indexed and accessed individually.

6. Now, try to access the URL with the details for the restaurant with the **`id`** field value of **`2`** again:



restaurants-db-default-rtdb.firebaseio.com/restaurants.json?orderBy="r_id"&equalTo=2

```
{"2":{"is_shutdown":false,"r_description":"Get the best pizza in town. We also serve vegan burgers!","r_id":2,"r_title":"Pizza John"}}
```

Figure 5.3 – Obtaining the response from Firebase for one restaurant JSON object

NOTE:

To see the structure of the JSON response in a more readable manner in your browser, you can add the **`&print=pretty`** *query parameter at the end of the request URL.*

Success! We've obtained the details of the restaurant with the **`r_id`** field's value of **`2`**.

Now, let's implement this request in our app:

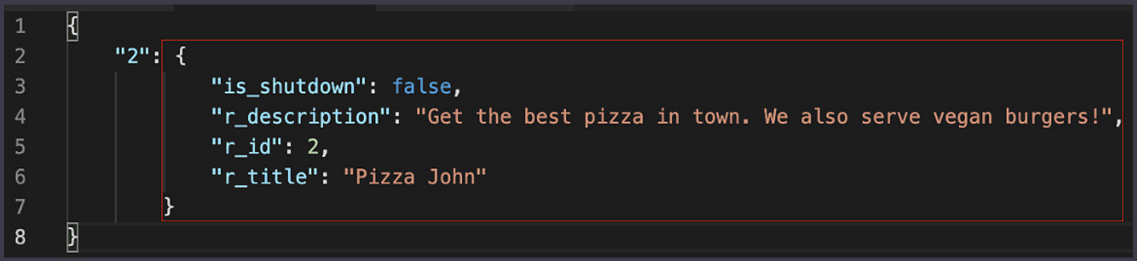
1. First, inside the **`RestaurantsApiService`** interface, define a **`suspend`** function called **`getRestaurant()`**, which will serve as another **`@GET`** HTTP method that will get the details of one restaurant:

```
interface RestaurantsApiService {
    […]
    @GET("restaurants.json?orderBy=\"r_id\"")
    suspend fun getRestaurant(
      @Query("equalTo") id: Int): Unit
}
```

Let's break down the code we've just added to our second HTTP method:

- The HTTP call defined by this method is an async job that takes some time to finish, so we've marked the method as a suspending function by adding the **`suspend`** keyword.
- Inside the **`@GET`** annotation, we specified not only that we want to access the **`restaurants.json`** JSON content, but this time we also hard-coded the **`orderBy`** query parameter and specified the **`r_id`** value so that we filter the elements by the value of their **`r_id`** key.
- This method receives one essential parameter – **`id: Int`** that represents the unique ID of the restaurant corresponding to the **`r_id`** field in the database. To tell Retrofit that this method parameter is a query parameter in the required HTTP call, we've annotated it with the **`@Query`** annotation and passed the **`"equalTo"`** value.

Yet our HTTP call defined by our **`getRestaurant()`** method is missing something crucial: the response type. We've set **`Unit`** as the response type, but we need to receive a proper response object. To see what to expect, let's have a closer look at the response we received earlier on inside our browser:



```
{
    "2": {
        "is_shutdown": false,
        "r_description": "Get the best pizza in town. We also serve vegan burgers!",
        "r_id": 2,
        "r_title": "Pizza John"
    }
}
```

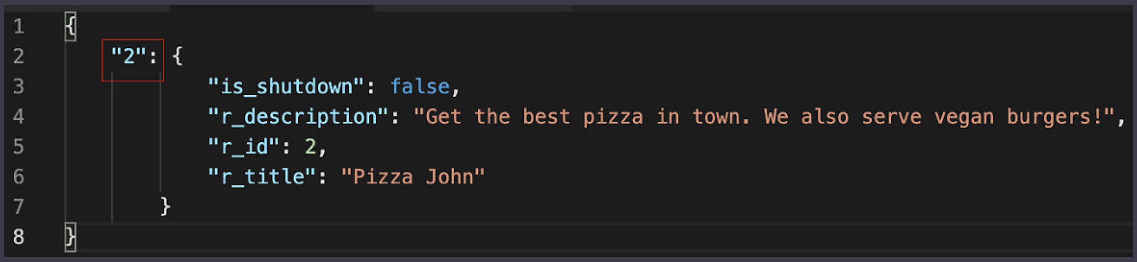
Figure 5.4 – The JSON response structure of the restaurant object

If we look at these fields, **`is_shutdown`**, **`r_description`**, **`r_id`**, and **`r_title`**, we can easily identify the response JSON object as the same JSON object that we receive in the existing HTTP request that gets all the restaurants.

And since we've mapped such a JSON object in the past to our **`Restaurant`** data class using the **`@Serialized`** annotations, we could very well say our new **`getRestaurant()`** HTTP call will receive a simple **`Restaurant`** object as a response.

We wouldn't be far from the truth, yet this response wouldn't be fully correct.

If we look closer at the previous JSON response, we notice that the restaurant JSON object is a value object that corresponds to a **String** key with the value of **2**:



```
{
    "2": {
            "is_shutdown": false,
            "r_description": "Get the best pizza in town. We also serve vegan burgers!",
            "r_id": 2,
            "r_title": "Pizza John"
        }
}
```

Figure 5.5 –Identifying the key field for the restaurant object

This key corresponds to an internal index generated by Firebase that represents the order number in which the corresponding restaurant was added to the database. This response structure isn't typical for most REST API responses, yet Firebase has this quirk of wrapping your JSON object in a key that is unknown at compile time.

2. To get around this, inside the **RestaurantsApiService** interface, update the **getRestaurant()** method to return a **Map** object with an unknown **String** key and a **Restaurant** data type as the value:

```
interface RestaurantsApiService {
    …
    @GET("restaurants.json?orderBy=\"r_id\"")
    suspend fun getRestaurant(@Query("equalTo") id:
Int)
      : Map<String, Restaurant>
}
```

Great work! We have our app ready to execute a second network request that obtains the details about a specific restaurant, so it's time to call this request.

Getting the contents of a specific restaurant

Now that we know how to obtain the details about a specific restaurant, it's time to execute our newly defined network request.

Our existing **RestaurantsScreen** composable delegates the responsibility of requesting the list of restaurants that must be displayed to a **ViewModel** class, so let's create another **ViewModel** so that our second screen can do the same:

1. Create a new file by left-clicking the application package, selecting **New**, and then **Kotlin Class/File**. Enter **RestaurantDetailsViewModel** as the name and select **File** as the type. Inside the newly created file, add the following code:

```
class RestaurantDetailsViewModel(): ViewModel() {
    private var restInterface:
RestaurantsApiService
    init {
        val retrofit: Retrofit = Retrofit.Builder()
            .addConverterFactory(GsonConverterFacto
ry
                .create())
            .baseUrl("your-firebase-base-url")
            .build()
        restInterface = retrofit.create(
            RestaurantsApiService::class.java)
    }
}
```

In the preceding snippet, we've created a **ViewModel** class where we instantiated a Retrofit client of type **RestaurantsApiService**, just like we did in the **RestaurantsViewModel** class.

The block of code that initializes a Retrofit client is indeed duplicated in both our **ViewModel** classes, but don't worry because you will be able to fix this during Chapter 9, *Implementing Dependency Injection with Jetpack Hilt*.

NOTE

Remember to pass your Firebase database URL to the **baseUrl()** *method. This URL should be identical to the one used in the* **RestaurantsViewModel** *class and should correspond to your Firebase Realtime Database project.*

2. Inside the newly created **ViewModel**, create a **getRemoteRestaurant()** method that receives an **id** parameter and takes care of executing the network request to get the details of a specific restaurant:

```
class RestaurantDetailsViewModel() : ViewModel() {
    private var restInterface:
RestaurantsApiService
     init { […] }
    private suspend fun getRemoteRestaurant(id:
Int):
            Restaurant {
        return withContext(Dispatchers.IO) {
            val responseMap = restInterface
                .getRestaurant(id)
            return@withContext
responseMap.values.first()
        }
    }
}
```

Let's break down what happens inside the **getRemoteRestaurant()** method:

- It receives an **id** parameter corresponding to the restaurant whose details we need and returns the specific **Restaurant** object.

- It is marked by the **suspend** keyword since the job of executing a network request is a suspending work that shouldn't block the main thread.
- It is wrapped in a **withContext()** block that specifies the **Dispatchers.IO** dispatcher since the suspending work should be run on the specialized IO thread.
- It executes the network request to obtain the details of a restaurant by calling the **getRestaurant()** suspending function on **restInterface** while passing **id** of the specific restaurant.
- Finally, it obtains **Map<String, Restaurant>** from the REST API. To unwrap this and obtain the restaurant, we call the **values()** function of **Map** and get the first **Restaurant** object with the **.first()** extension function.

NOTE:

The **first()** *extension function is called on the* **Collection<Restaurant>** *object returned by the* **values()** *function of* **Map**. *With this extension function, we are obtaining the first element, that is, the* **Restaurant** *object we're interested in. However, the* **first()** *extension function can throw a* **NoSuchElementException** *if for some reason we query for a non-existent restaurant. In production, you should cover this case as well by catching such an exception.*

3. Since **RestaurantDetailsViewModel** will hold the state of the restaurant details screen, add a **MutableState** object that will hold a **Restaurant** object and initialize it with a **null** value until we finish executing the network request that retrieves it:

```
class RestaurantDetailsViewModel(): ViewModel() {
    private var restInterface:
RestaurantsApiService
    val state = mutableStateOf<Restaurant?>(null)
     […]
}
```

4. Inside the **init** block of **RestaurantDetailsViewModel**, below the instantiation of the Retrofit client, launch a coroutine with the help of the **viewModelScope** builder:

```
init {
    […]
    restInterface = retrofit.create(…)
    viewModelScope.launch {
        val restaurant = getRemoteRestaurant(2)
        state.value = restaurant
    }
}
```

We needed to launch a coroutine because the job of getting a **Restaurant** object from our remote Firebase API would have blocked the main thread. We've used the built-in **viewModelScope** coroutine builder to make sure that the launched coroutine will live as long as the **RestaurantDetailsViewModel** instance does. Inside the coroutine, we did the following:

1. We first called the suspending **getRemoteRestaurants()** function and passed a hardcoded value of **2** as the **id** of the restaurant. At this time, **RestaurantsViewModel** has no idea what's the **id** of the restaurant that it's looking for – we will fix this soon when we perform the navigation.
2. We stored the obtained **Restaurant** inside the **restaurant** variable and passed it to the **state** variable of the **RestaurantDetailsViewModel** class so that the UI will be recomposed with the freshly received restaurant content.

We've executed the network request to obtain the details about a restaurant and prepared the state so that a Compose-based screen can display its contents. Let's build the new screen up next.

Building the restaurant details screen

We need to create a new composable screen that will display the details about a specific restaurant:

1. Create a new file inside the application package called **RestaurantDetailsScreen** and create the **RestaurantDetailsScreen** composable:

```
@Composable
fun RestaurantDetailsScreen() {
    val viewModel: RestaurantDetailsViewModel =
        viewModel()
    val item = viewModel.state.value
    if (item != null) {
        // composables
    }
}
```

Inside of it, we've instantiated its corresponding **ViewModel** and accessed the **State** object, just like we previously did in the **RestaurantsScreen** composable. The **State** object holds the **Restaurant** object, which we're storing inside the **item** variable. If **item** is not **null**, we will display the details about the restaurant by passing a composable hierarchy.

2. Since we plan to reuse some composable functions from the first screen, head back inside the **RestaurantsScreen.kt** file and mark the **RestaurantIcon** and **RestaurantDetails** composables as public for use by removing their **private** keywords.
3. Add a new parameter to the **RestaurantDetails** composable called **horizontalAlignment** and pass it to the column's **horizontalAlignment** parameter:

```
@Composable
fun RestaurantDetails(
    … ,
    modifier: Modifier,
    horizontalAlignment: Alignment.Horizontal
```

```
                                          =
Alignment.Start
) {
    Column(
        modifier = modifier,
        horizontalAlignment = horizontalAlignment
    ) { ... }
}
```

By doing so, we can control how the **Column** children are horizontally aligned so we can change this behavior in the new screen. Since we want **Column** to position its children horizontally to the left by default (so that its effect in the **RestaurantsScreen** composable won't differ), we passed **Alignment.Start** as the default value.

4. Inside the **RestaurantDetailsScreen** composable, add a **Column** instance that contains **RestaurantIcon**, **RestaurantDetails**, and **Text** composables, all positioned vertically and centered horizontally:

```
@Composable
fun RestaurantDetailsScreen() {
    val viewModel: RestaurantDetailsViewModel =
        viewModel()
    val item = viewModel.state.value
    if (item != null) {
        Column(
            horizontalAlignment =
                Alignment.CenterHorizontally,
            modifier =
                Modifier.fillMaxSize().padding(16.d
p)
        ) {
            RestaurantIcon(
                Icons.Filled.Place,
                Modifier.padding(
                    top = 32.dp,
                    bottom = 32.dp
                )
            )
            RestaurantDetails(
                item.title,
                item.description,
                Modifier.padding(bottom = 32.dp),
                Alignment.CenterHorizontally)
            Text("More info coming soon!")
        }
    }
}
```

To prove how simple it is to reuse composables, we've passed the same **RestaurantIcon** and **RestaurantDetails** composables used in the first screen to our **Column**. We've configured them with different **Modifier** objects and additionally passed **Alignment.centerHorizontally** to the **RestaurantDetails** composable's new alignment parameter added previously.

5. To test that everything works fine, and our new screen renders the details of the hardcoded restaurant with an **id** value of **2**, navigate back to **MainActivity** and inside the **setContent** method, replace the **RestaurantsScreen** composable with **RestaurantDetailsScreen**:

```
setContent {
    RestaurantsAppTheme {
        //RestaurantsScreen()
        RestaurantDetailsScreen()
    }
}
```

6. Run the application and we get the following screenshot:

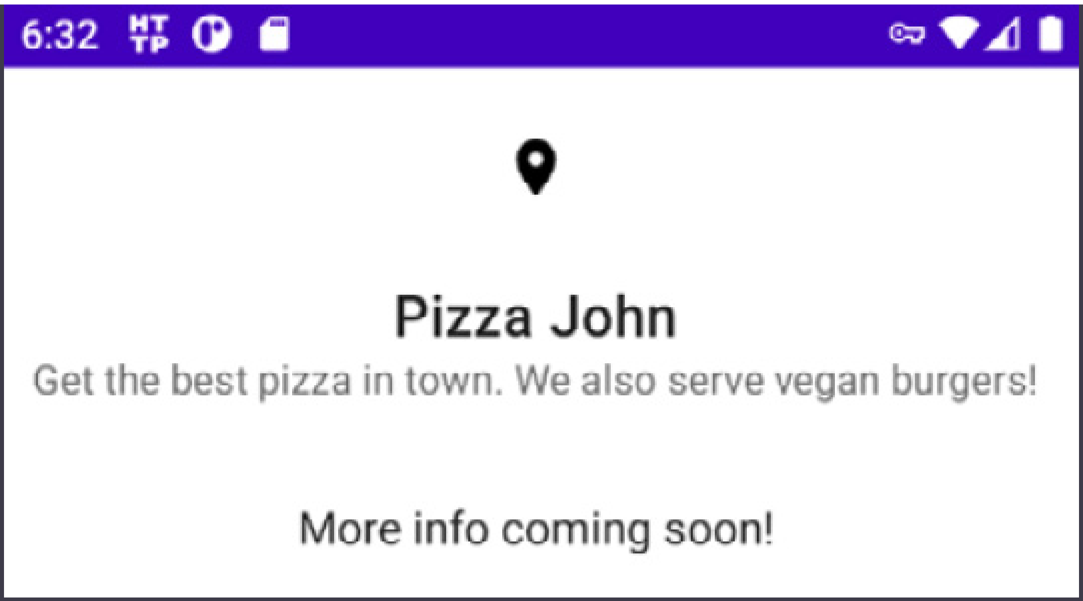


Figure 5.6 – Displaying the RestaurantDetailsScreen() composable

Awesome! We have now created our second screen, the restaurant details screen. We can now start thinking about the navigation between our two screens.

Implementing navigation with Jetpack Navigation

Navigation within apps represents those interactions that allow the user to navigate back and forth between several screens.

In our Restaurants application, we now have two screens, and we want to navigate from the first one to the second one. In the first screen, we display a list of restaurants and when the users press on one restaurant item from the list, we want to take them to the second screen, the details screen:

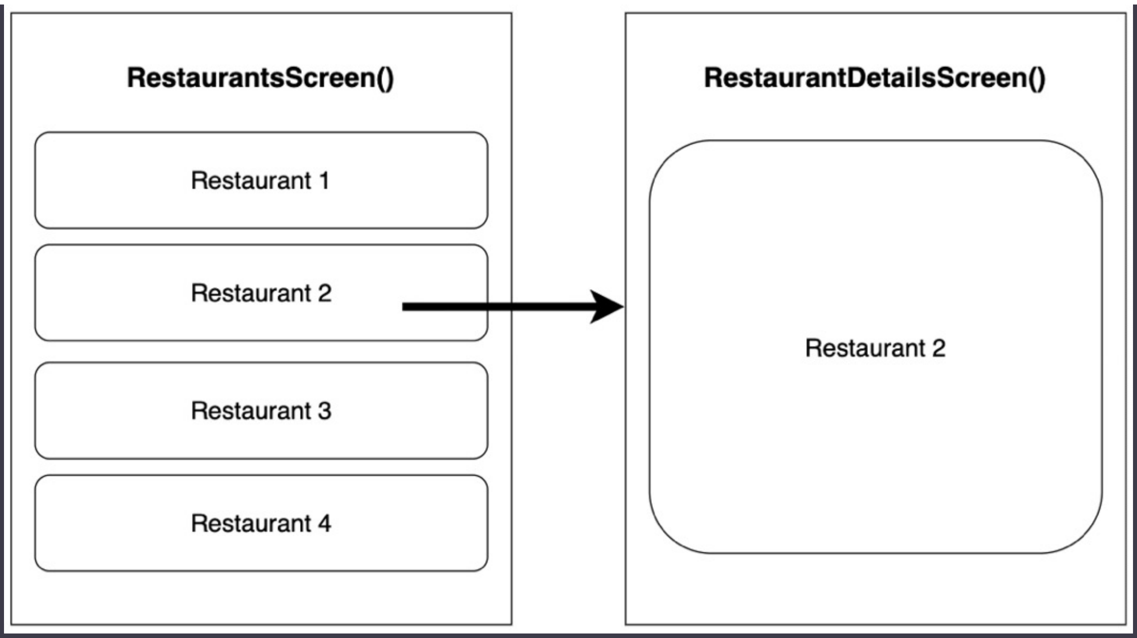


Figure 5.7 – Navigation from list screen to details screen

Basically, we want to perform a simple navigation action from the `RestaurantsScreen` composable to the `RestaurantDetailsScreen` composable. To achieve a simple navigation action, we need to implement a navigation library that will not only allow us to transition from the first screen to the second screen but should also allow us to return to the previous screen with the press of the **Back** button.

As we already know, the Jetpack Navigation component comes to our rescue as it will help us implement such a behavior! Let's start with the following steps:

1. Inside the `build.gradle` file in the app module, add the dependency for the Navigation component with Compose inside the dependencies block:

```
implementation "androidx.navigation:navigation-
compose:2.4.2"
```

After updating the `build.gradle` file, make sure to sync your project with its Gradle files. You can do that by pressing on the **File** menu option and then by selecting **Sync Project with Gradle Files**.

2. Inside the **`MainActivity`** class, create a new empty composable function called **`RestaurantsApp()`**:

```
@Composable
private fun RestaurantsApp() {
}
```

This composable function will act as the parent composable function of our Restaurants application. Here, all the screens of the app will be defined.

3. Inside the **`onCreate()`** method, replace the **`RestaurantsDetailsScreen()`** composable that is passed to the **`setContent`** method with the **`RestaurantsApp()`** composable:

```
setContent {
    RestaurantsAppTheme {
        RestaurantsApp()
    }
}
```

4. Inside the **`RestaurantsApp()`** composable function, instantiate **`NavController`** via the **`rememberNavController()`** method:

```
@Composable
private fun RestaurantsApp() {
    val navController = rememberNavController()
}
```

The **`NavController`** object handles the navigation between composable screens – it operates on the back stack of composable destinations. This means that across recompositions, it must keep the current state of the navigation stack. For that to happen, it must be a stateful object – that's why we used the **`rememberNavController`** syntax, which is similar to the **`remember`** block we've been using when defining **`State`** objects.

5. Next up, we need to create a **`NavHost`** container composable that will display the composable destinations. Every time a navigation action

between composables is done, the content within **`NavHost`** is recomposed automatically.

Add a **`NavHost`** composable and pass both, the **`NavController`** instance created previously and an empty **`String`** to the **`startDestination`** parameter:

```
import androidx.navigation.compose.NavHost
[…]
@Composable
private fun RestaurantsApp() {
    val navController = rememberNavController()
    NavHost(navController, startDestination = "") {
    }
}
```

Among other parameters, **`NavHost`** specifies three mandatory parameters:

- A **`navController: NavHostController`** object that is associated with a single **`NavHost`** composable. **`NavHost`** links **`NavController`** with a navigation graph that defines the possible destinations of the application. In our case, we've passed the **`navController`** variable to this parameter.
- A **`startDestination: String`** object that defines the entry-point **route** of the navigation graph. The route is **`String`**, which defines the path to a specific destination (composable screen). Every destination should have a unique route. In our case, since we haven't defined any routes, we've passed an empty **`String`** to **`startDestination`**.
- The **`builder: NavGraphBuilder.() -> Unit`** trailing lambda parameter, which uses the lambda syntax from the Navigation Kotlin DSL (just like **`LazyColumn`** or **`LazyRow`** did with their own DSL) to construct a navigation graph. In here, we should define routes and set corresponding composables, yet so far we've set an empty body **`{ }`** function to the trailing lambda parameter.

6. To build the navigation graph, we must make use of the **`builder`** parameter and instead of passing only an empty function, inside of it, we need to start adding routes that specify composable destinations.

To do that, make use of the DSL function called **`composable()`** where you can provide a route string to the **`route`** parameter and a composable function corresponding to the desired destination to the trailing lambda **`content`** parameter:

```
@Composable
private fun RestaurantsApp() {
    val navController = rememberNavController()
    NavHost(
        navController,
        startDestination = "restaurants"
    ) {
        composable(route = "restaurants") {
            RestaurantsScreen()
        }
    }
}
```

Through the **`composable()`** DSL function, we've created a route with the value of **`"restaurants"`** that navigates to the **`RestaurantsScreen()`** composable.

Additionally, we've passed the same route to the **`startDestination`** parameter of **`NavHost`**, thereby making our **`RestaurantsScreen()`** composable the unique entry point of our application.

7. By calling the **`composable()`** DSL function again inside the navigation graph builder, add another route that points to the **`RestaurantDetailsScreen()`** destination and that derives from the **`"restaurants"`** route by appending the **`{restaurant_id}`** argument placeholder:

```
NavHost(navController, startDestination = "...") {
    composable(route = "restaurants") { … }
    composable(route =
"restaurants/{restaurant_id}") {
        RestaurantDetailsScreen()
        }
    }
```

We want to navigate from the **`"restaurants"`** route to this new route that points to the **`RestaurantDetailsSreen()`** composable, so the **`{restaurant_id}`** placeholder will take the **`id`** value of the restaurant to which we are trying to navigate.

In practice, this route branches off the **`"restaurants"`** route, and while being structured similarly to a URL (because of the **`"/"`** element that delimitates a new path), we can say that this route can have multiple values, depending on **`id`** of the restaurant we're looking to navigate to. For example, this route can have values at runtime such as **`"restaurants/0"`** or **`"restaurants/2"`**.

8. Inside the navigation graph, we've defined the routes and their corresponding destinations, but we haven't really performed the actual navigation between the two screens. To do that, we first need to have a trigger or callback that notifies us when the user pressed on a restaurant item within the restaurant list, so we can navigate to the restaurant details screen.

Inside the **`RestaurantsScreen.kt`** file, modify the **`RestaurantItem`** composable to expose an **`onItemClick`** callback function that provides us with **`id`** of the restaurant that is clicked, and also call it when the entire restaurant's **`Card`** is pressed on:

```
@Composable
fun RestaurantItem(item: Restaurant,
                   onClick: (id: Int) -> Unit,
                   onItemClick: (id: Int) -> Unit) {
    val icon = …
    Card(elevation = 4.dp,
         modifier = Modifier
            .padding(8.dp)
```

```
              .clickable { onItemClick(item.id) }) { …
  }
  }
```

9. To prevent confusion, refactor the **RestaurantItem** composable by renaming the old **onClick** parameter to a more suggestive name, such as **onFavoriteClick**:

```
@Composable
fun RestaurantItem(item: Restaurant,
                   onFavoriteClick: (id: Int) ->
Unit,
                   onItemClick: (id: Int) -> Unit)
{
    val icon = …
    Card(…) {
        Row(…) {
            …
            RestaurantIcon(icon,
Modifier.weight(0.15f))
            {
                onFavoriteClick(item.id)
            }
        }
    }
}
```

10. Inside the **RestaurantsScreen()** composable, add a similar **onItemClick** callback function as a parameter, and call it when the **onItemClick** callback comes from the **RestaurantItem** composable:

```
@Composable
fun RestaurantsScreen(onItemClick: (id: Int) ->
Unit = { }) {
    val viewModel: RestaurantsViewModel =
viewModel()
    LazyColumn(...) {
        items(viewModel.state.value) { restaurant -
>
            RestaurantItem(
                restaurant,
                onFavoriteClick =
                  { id ->
viewModel.toggleFavorite(id) },
                onItemClick = { id ->
onItemClick(id) })
        }
    }
}
```

Additionally, we've changed the **onClick** parameter name of the **RestaurantItem** composable call to match its signature of **onFavoriteClick**.

What we are essentially doing is propagating events through callbacks from child composables to parent composables.

11. Inside **NavHost**, update the **RestaurantsScreen()** composable destination to listen for navigation callbacks and then, inside the callback, trigger the navigation between composables by calling the **navigate()** method, which expects **route** as a parameter:

```
@Composable
private fun RestaurantsApp() {
    val navController = rememberNavController()
    NavHost(navController, startDestination =
"...") {
        composable(route = "restaurants") {
            RestaurantsScreen { id ->
                navController.navigate("restaurants
/$id")
            }
        }
        composable(
            route = "restaurants/{restaurant_id}"
        ) {
            RestaurantDetailsScreen()
        }
    }
}
```

Inside the new trailing lambda function of **`RestaurantsScreen`**, we now receive the **`id`** value of the restaurant we need to navigate to. To trigger the navigation, we called the **`navigate()`** method, and to its **`route`** parameter, we passed the **`"restaurants/$id"`** string to match the route of our other composable destination, **`RestaurantDetailsScreen()`**.

12. Try running the application and verify the following.

When the app is launched, the **`RestaurantsScreen()`** composable is composed and displayed. In other words, you are at the **`"restaurants"`** route because we've set this route as **`startDestination`** for our navigation graph. On the navigation back stack, this destination will be added:

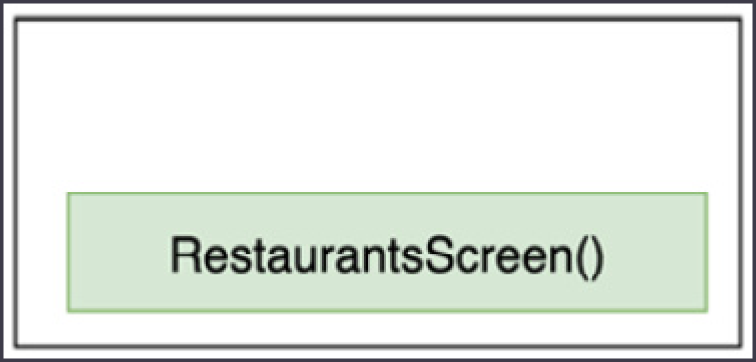


Figure 5.8 – Back stack with the start destination

When pressing on one of the restaurants on the list, navigation is triggered and you arrive at the **`RestaurantDetailsScreen()`** composable destination. On top of the navigation back stack this destination will be added:

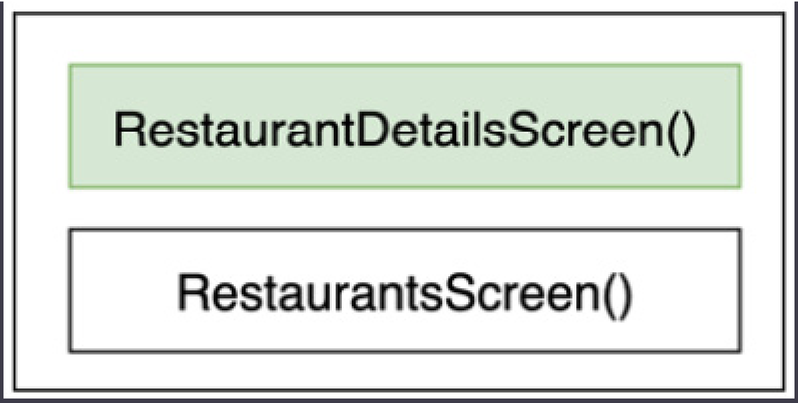


Figure 5.9 – Back stack after navigating to another destination

When pressing the system's **Back** button while being at the `RestaurantDetailsScreen()` destination, you are sent back to the existing destination in the back stack, `RestaurantsScreen()`. This means that on the back stack, the top destination is popped, and only the root destination remains:

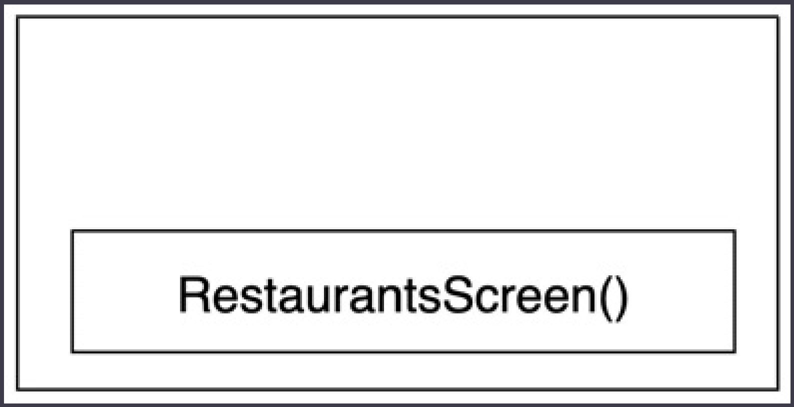


Figure 5.10 – Back stack after returning to start destination

The navigation works, but if you noticed, it always points to the same restaurant. This happens because of two reasons:

- While we defined the `{restaurant_id}` placeholder argument in the route that points to `RestaurantDetailsScreen()`, we didn't define this argument inside the DSL `composable()` function as a navigation argument, so the Navigation component has no idea how to send it to the route's composable destination.
- Inside `RestaurantDetailsViewModel`, we've hardcoded the id of the restaurant to the value of `2`.

We want the user to see details about the restaurant that is pressed on, so let's fix these issues and pass the ID of the restaurant dynamically.

13. For the **`RestaurantDetailsScreen()`** destination, apart from **`route`**, add the **`arguments`** parameter that expects a list of **`NamedNavArgument`** objects, and pass such an argument using the **`navArgument`** function:

```
NavHost(navController, startDestination = "..."){
    composable(route = "restaurants") { … }
    composable(
        route = "restaurants/{restaurant_id}",
        arguments =
            listOf(navArgument("restaurant_id") {
                type = NavType.IntType
            })
    ) { RestaurantDetailsScreen() }
}
```

This argument specifies the same **`"restaurant_id"`** key that we've added as a place holder within **`route`** and allows the Navigation library to expose this argument to the destination composable. Additionally, the **`navArgument`** function exposes **`NavArgumentBuilder`**, where we specified the type of the argument to be **`IntType`**.

To obtain the argument's value inside the **`RestaurantDetailsScreen()`** destination, the **`composable()`** DSL function exposes a **`NavBackStackEntry`** object that allows us to get the value as follows:

```
composable(…) { navStackEntry ->
    val id =
        navStackEntry.arguments?.getInt("restaurant_i
d")
    RestaurantDetailsScreen()
}
```

Yet our **`RestaurantDetailsScreen()`** destination doesn't expect the **`id`** of a restaurant, but **`RestaurantDetailsViewModel`** does, so we will not perform the previous changes where we access **`navStackEntry`**; instead, we will do something similar in the **`ViewModel`** soon enough.

14. Behind the scenes, the Navigation component saves the navigation arguments stored in **`NavStackEntry`** into **`SavedStateHandle`**, which our VM exposes. This means that we can take advantage of that, and instead of obtaining the ID of the restaurant inside the **`RestaurantDetailsScreen()`** composable, we can directly obtain it in **`RestaurantDetailsViewModel`**.

First, add the **`SavedStateHandle`** parameter to the **`RestaurantDetailsViewModel`** constructor, just like we did within **`RestaurantsViewModel`**:

```
class RestaurantDetailsViewModel(
    private val stateHandle: SavedStateHandle
) : ViewModel() {
    […]
    init { […]  }
    private suspend fun getRemoteRestaurant(id: Int)
{
        […]
    }
}
```

15. Inside of the **`init { }`** block of **`ViewModel`**, below the instantiation of the Retrofit client, store the ID of the restaurant inside a new **`id`** variable while obtaining it dynamically from the **`SavedStateHandle`** object, and then pass it to the **`getRemoteRestaurant()`** method call:

```
class RestaurantDetailsViewModel(private val
stateHandle: SavedStateHandle): ViewModel() {
    …
    init {
        val retrofit: Retrofit =
Retrofit[…].build()
```

```
        restInterface = […]
        val id = stateHandle.get<Int>
("restaurant_id")
            ?: 0
        viewModelScope.launch {
            val restaurant =
getRemoteRestaurant(id)
            state.value = restaurant
        }
    }
    …
}
```

We've instructed **navArgument** that the argument is of type **Int**, so we've obtained it as an **Int** value from **stateHandle** and passed the same **"restaurant_id"** key that we've used to define **navArgument**.

This approach will protect us from system-initiated process death scenarios as well. The user could navigate to the **RestaurantDetailsScreen()** destination of a restaurant with an **id** value of **2**, and then minimize the app for a while. In the meantime, the system could decide to kill the process of the app to free up memory, so when the user resumes the app, the system would restore it and provide us with a **SavedStateHandle** object that contains the ID of the restaurant with the value of **2**.

In conclusion, the app would know to obtain the details of the restaurant the user initially navigated to, so the application behaves correctly for this edge case.

16. Run the app again and verify that this time when pressing on one restaurant item in the **RestaurantsScreen()**start destination, the details about this restaurant are displayed in the second destination, **RestaurantDetailsScreen()**.
    NOTE

We used the Navigation component with destinations that are composable functions. Inside these composables, we instantiate **`ViewModel`** *objects. Since these composables are in a back stack of destinations, their* **`ViewModel`** *objects become scoped to the lifetime of the composables. In other words, with the addition of the Navigation component, the* **`ViewModel`** *objects have the same lifetime as the composable screen that they are attached to.*

Perfect! Now our Restaurants app has two screens that you can navigate between whenever the user presses on a restaurant from within our list. It's time to explore another type of navigation event.

Adding support for deep links

Deep links allow you to redirect users to specific parts of your application without having them go through all the intermediary screens. This technique is especially useful for marketing campaigns because it can boost user engagement while also providing a good user experience.

Deep links are usually incorporated within URI schemes or custom schemes. This allows you to configure anything from an image advertisement, text advertisement, or even a QR code that when clicked or scanned redirects you to a specific page of the app. If your app is configured to know how to handle such schemes, the user will be able to open that particular link with your application.

For example, say that for our Restaurants application, we start a marketing campaign where we include some advertisements on the internet that showcase some special restaurants. We configure the advertisements to be clickable and to redirect to the following link, which contains the ID of the advertised restaurant, such as **`2`**:
`https://www.restaurantsapp.details.com/2`.

This URI will not work when loaded into a browser application (because there is no such website), yet we can configure our app to know how to interpret it as a deep link.

When a user is browsing a search engine and presses on a campaign advertisement for one of our restaurants, the app should know how to handle these actions and should allow the user to be redirected to our application:

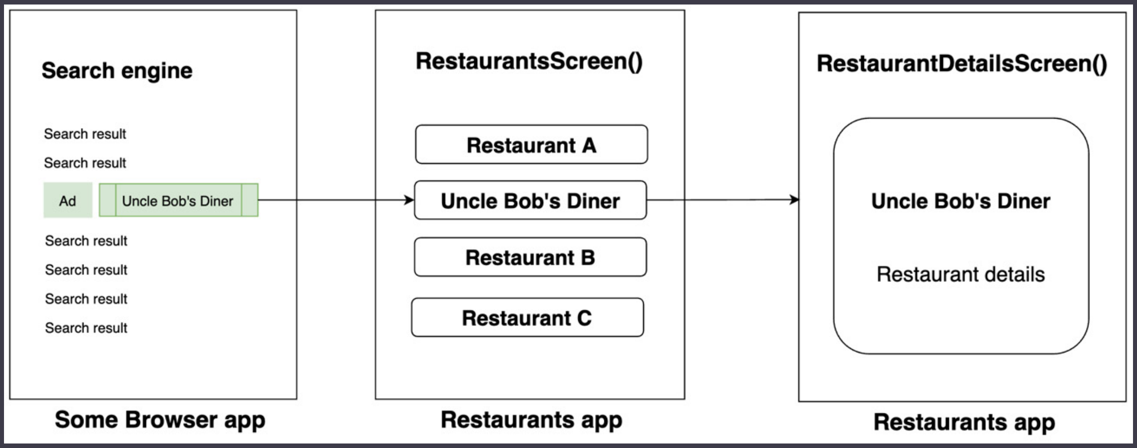


Figure 5.11 – Inefficient redirect to our Restaurants app

Our application has as the start destination the **`RestaurantsScreen()`** composable, so the user should manually find the restaurant that was initially presented on the advertisement, and press on it to navigate to the **`RestaurantDetailsScreen()`** destination.

This is obviously a bad practice because we don't want the user to perform manual navigations within our app to get to the advertised restaurant. Imagine if other apps required the user to navigate not through one or two screens as per our application, but more screens – this would result in a bad user experience and the campaign would be ineffective.

Deep links, however, allow you to automatically redirect the user to your desired destination:

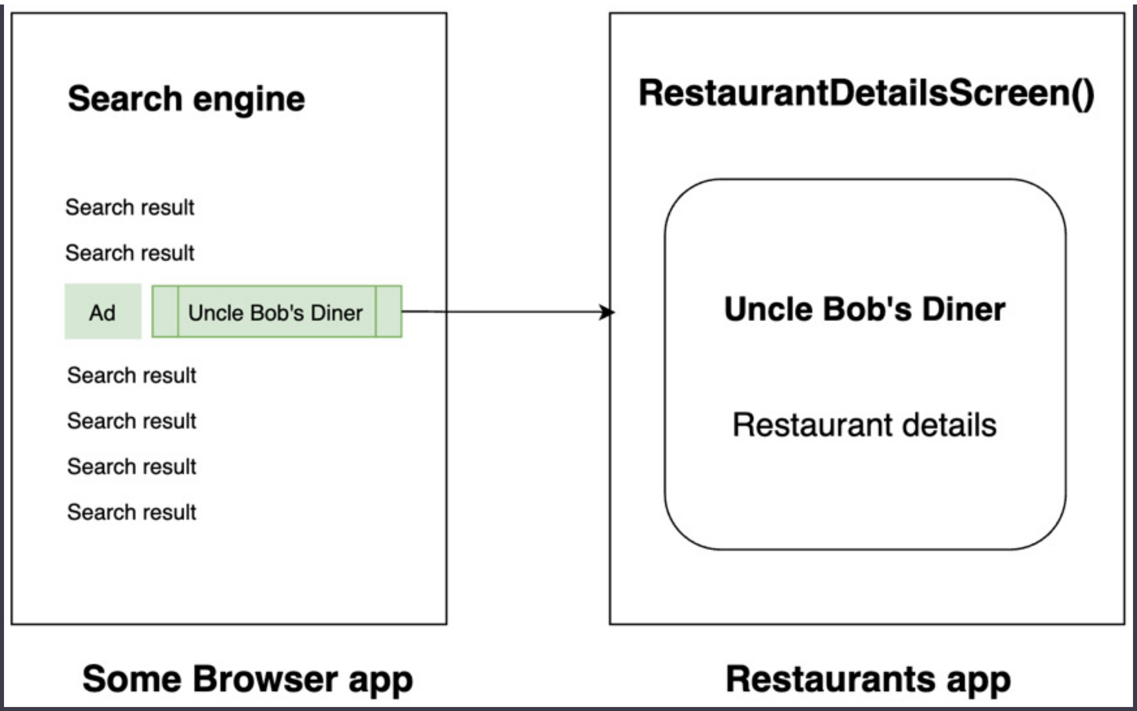


Figure 5.12 – Direct deep link to the screen of interest

By redirecting the user directly to the screen of interest, we improve the user experience and expect our advertising campaign to perform better.

Let's implement such a deep link in our Restaurants application with the help of the Navigation component library:

1. Inside the **`RestaurantDetailsScreen()`** DSL **`composable()`** function, apart from **`route`** and **`arguments`**, add another parameter called **`deepLinks`** that expects a list of **`NavDeepLink`** objects, and pass such an argument using the **`navDeepLink`** function:

```
NavHost(navController, startDestination =
"restaurants")
{
  composable(route = "restaurants") {…}
  composable(
    route = "restaurants/{restaurant_id}",
    arguments = listOf(
      navArgument("restaurant_id") {…}
    ),
    deepLinks = listOf(navDeepLink {
      uriPattern =
    "www.restaurantsapp.details.com/{restaurant_id}
"
      })
     ) { RestaurantDetailsScreen() }
}
```

The **navDeepLink** function expects in turn a **NavDeepLinkDslBuilder** extension function that exposes its own DSL. We've set the **uriPattern** DSL variable to expect our custom URI of **www.restaurantsapp.details.com** but also added our placeholder **"restaurant_id"** argument that will allow the Navigation component to parse and provide us with the ID of the restaurant from the deep link.

Right now, our application knows how to handle a deep link, but only internally.

2. To make our deep link available externally, inside the **AndroidManifest.xml** file, add the following **<intent-filter>** element within our **MainActivity**'s **<activity>** element:

```
<application … >
    <activity
        android:name=".MainActivity"
        […] >
        <intent-filter>
            <action android:name="[…].action.MAIN"
/>
            <category android:name="[…].LAUNCHER"
/>
        </intent-filter>
        <intent-filter>
          <data
                android:host="www.restaurantsapp.
                    details.com"
                android:scheme="https" />
            <action android:name="android.intent.
                action.VIEW" />
            <category android:name="android.intent.
                category.DEFAULT" />
            <category android:name="android.intent.
                category.BROWSABLE" />
        </intent-filter>
    </activity>
</application>
```

Let's break up what we've just added inside the new `<intent-filter>` element:

- A `<data>` element that specifies the following:
- The `host` parameter as the deep link URI that we've set previously in our navigation graph. This is the URI that our ads should link to.
- The `scheme` parameter of the deep link as `https`. Every `<data>` element should define a scheme so that the URI is recognized.
- A `<category>` element of `BROWSABLE` that is required for the intent filter to be accessed from web browser apps.
- A `<category>` element of `DEFAULT` that makes the app intercept the deep link's intents implicitly. Without it, the app could be started only if the deep link intent specified the application component name.

To test the deep link, we need to simulate a deep link action. Let's imagine that we want to test a deep link that points to a restaurant that has the ID with the value of `2`. The deep link would look like this: `https://www.restaurantsapp.details.com/2`.

Since we don't have any advertisements that refer to our deep link, we have two options:

- Create a QR code with this URL and then scan it with our device.
- Launch an intent from the command line that simulates the deep link.

Let's go with the second option.

3. Build the project and run the application on an emulator or physical device. This step is needed so that the installed application knows how to respond to our deep link.
4. Close the app or minimize it, but make sure you leave your emulator or device connected to Android Studio.
5. Open the terminal inside Android Studio, paste the following command and enter it:

```
$ adb shell am start -W -a
android.intent.action.VIEW -d
"https://www.restaurantsapp.details.com/2"
```

6. The emulator/device that you have connected to Android Studio should now prompt a disambiguation dialog asking you what app you'd like to open the deep link with:

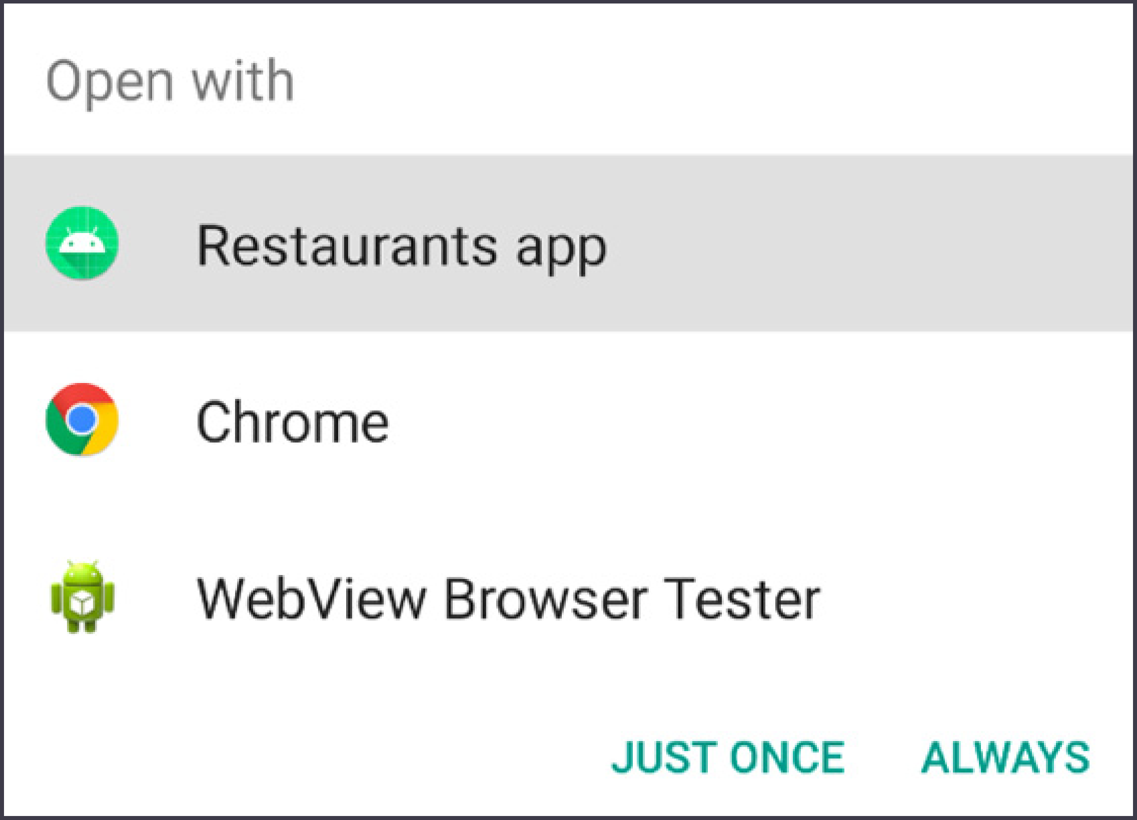


Figure 5.13 – Disambiguation dialog displayed when launching a deep link

Our application is one of those apps and this means that it has been correctly configured to intercept our deep links.

7. Select **Restaurants app** (or whatever you called your app) and press **JUST ONCE**. The application should open our `RestaurantDetailsScreen()` destination and show the details of the desired restaurant.

Optionally, you can try pressing the system's **Back** button. The Navigation component application knows automatically how to send the user back to the `RestaurantsScreen` composable.

Now that we've also successfully added deep link functionality to our Restaurant application, it's time to wrap this chapter up.

Summary

In this chapter, we learned how to navigate between screens within our Restaurants application. We did that easily with the help of the Jetpack Navigation component library.

We started off by learning the basics of the Jetpack Navigation library and understood how easy our life becomes when having to handle navigation back stacks. Afterward, we created a new screen, implemented the Navigation library, and explored how seamless it is to add navigation between composables. Finally, we added support for deep links and made sure to test such a deep link within our app.

Next up, it's time to focus on improving the quality and architecture of our Restaurants application.