4

Navigation in Modern Android Development

In Android development, navigation is the interaction that allows your Android application users to navigate to, from, and back out from the different screens within your app, an action that is very vital in the mobile ecosystem.

Jetpack navigation has simplified navigation between screens, and in this chapter, we will learn how to implement navigation with a simple view click, from the bottom navigation bar, which is most commonly used, by navigating with arguments, and more.

In this chapter, we'll cover the following recipes:

- Implementing a bottom navigation bar using navigation destinations
- Navigating to a new screen in Compose
- Navigating with arguments
- Creating deep links for destinations
- Writing tests for navigation

Technical requirements

The complete source code for this chapter can be found at **https://github.com/PacktPublishing/Modern-Android-13-Development-Cookbook/tree/main/chapter_four**.

Implementing a bottom navigation bar using navigation destinations

In Android development, having a bottom navigation bar is very common; it helps inform your users that there are different sections in your application. In addition, other apps opt to include a navigation drawer activity, which holds a profile and additional information about the application.

An excellent example of an app that utilizes both – a navigation drawer and bottom navigation – is Twitter. It is also important to mention that some companies prefer to have a top navigation bar as a preference. In addition, others such as Google Play Store have both bottom and drawer navigation.

Getting ready

Create a new Android project with your preferred editor or Android Studio, or you can use any project from previous recipes.

How to do it...

In this recipe, we are going to create a new project and call it **BottomNavigationBarSample**:

1. After creating our new empty **Activity** **BottomNavigationBarSample** project, we will start by adding the required navigation dependency in **build.gradle**, and then sync the project:

```
implementation 'android.navigation:navigation-compose:2.5.2'
```

2. As noticed in a previous new project, when you create a new project, there is code that comes with it, the **Greeting()** function; you can go ahead and delete that code.
3. After deleting that code, let us go ahead and create a **sealed** class in the main package directory and call it **Destination.kt**, where we will define our **route** string, **icon: Int**, and **title: String** for our bottom navigation items:

```
sealed class Destination(val route: String, val icon: Int, val title: String) {...}
```

Strictly speaking, we might not need the **sealed** class, but it is a nicer way to implement navigation. A **sealed** class in Kotlin represents a restricted class hierarchy that provides more control over inheritance. Alternatively, you can think of it as a class that, in its value, can have one of the types from a limited set, but it cannot have any other types.

4. Inside the **sealed** class, now let's go ahead and create our destinations. For our sample, we will assume we are creating a budgeting app. Hence, the destinations we can have are **Transactions**, **Budgets**, **Tasks**, and **Settings**. See the next step on how to get the

icons; in addition, you will need to import them. For good practice, you can extract the **`String`** resource and save it in the **`String`** XML file. You can try this as a small exercise:

```
sealed class Destination(val route: String, val icon: Int, val title: String) {
    object Transaction : Destination(
        route = "transactions", icon =
            R.drawable.ic_baseline_wallet,
            title = "Transactions"
    )
    object Budgets : Destination(
        route = "budget", icon =
            R.drawable.ic_baseline_budget,
            title = "Budget"
    )
    object Tasks : Destination(route = "tasks", icon =
        R.drawable.ic_add_task, title = "Tasks")
    object Settings : Destination(
        route = "settings", icon =
            R.drawable.ic_settings,
            title = "Settings"
    )
    companion object {
        val toList = listOf(Transaction, Budgets,
        Tasks, Settings)
    }
}
```

5. For the icons, you can access them easily but clicking on the resource folder (**`res`**), then navigating to **Vector Assets** | **Clip Art**, which will launch and bring up free icons that you can use, as shown in *Figure 4.1*:

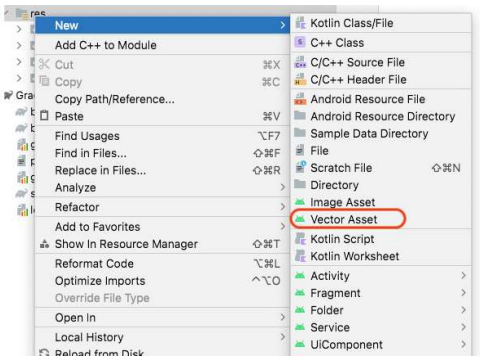

Figure 4.1 – How to access Vector Asset

6. You can also upload an SVG file and access it through **Asset Studio**. For more icons, you can check out this link: **https://fonts.google.com/icons**.

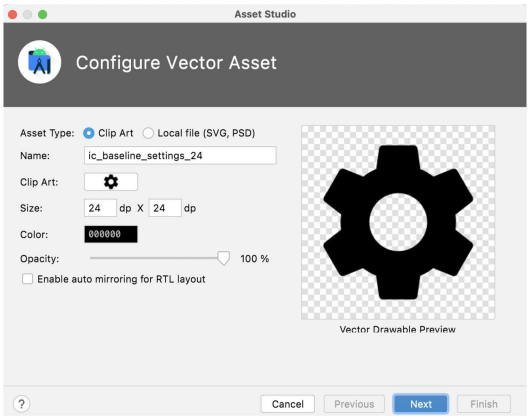

Figure 4.2 – Asset Studio

7. Now, for the destinations we just added, let's go ahead and add dummy text to verify that, indeed, when we navigate, we are on the right screen. Create a new file and name it **`AppContent.kt`**. Inside **`AppContent`**, we will add our **`Transaction`** function, which will be our home screen, where new users enter the app the first time; then later, they can navigate to other screens:

```
@Composable
fun Transaction(){
    Column(
        modifier = Modifier
            .fillMaxSize()
            .wrapContentSize(Alignment.Center)
    ) {
    ...
    }
}
```

8. Go ahead and add the remaining three screens, **`Task`**, **`Budget`**, and **`Settings`**, using the same composable pattern.

9. We now need to create a bottom navigation bar composable and tell the **`Composable`** function how to react when clicked, and also restore the state when re-selecting a previously selected item:

```
@Composable
fun BottomNavigationBar(navController: NavController, appItems: List<Destination>) {
    BottomNavigation(
        backgroundColor = colorResource(id =
            R.color.purple_700),
            contentColor = Color.White
    ) {
    ...
    }
}
```

10. Now, let's go to **`MainActivity`** and create **`NavHost`** and a few composable functions, **`AppScreen()`**, **`AppNavigation()`**, and **`BottomNavigationBar()`**. Each nav controller must be associated with a single nav host composable because it connects the controller with a nav graph that helps specify the composable directions:

```
@Composable
fun AppNavigation(navController: NavHostController) {
    NavHost(navController, startDestination =
    Destination.Transaction.route) {
        composable(Destination.Transaction.route) {
            Transaction()
        }
        composable(Destination.Budgets.route) {
            Budget()
        }
        composable(Destination.Tasks.route) {
            Tasks()
        }
        composable(Destination.Settings.route) {
            Settings()
        }
```

```
        }
    }
```

11. Finally, let's go ahead and glue everything together by creating another composable function and calling it **`AppScreen()`**. We will call this function inside **`setContent`** in the **`onCreate()`** function:

```
@Composable
fun AppScreen() {
    val navController = rememberNavController()
    Scaffold(
        bottomBar = {
            BottomNavigationBar(navController =
            navController, appItems =
            Destination.toList) },
        content = { padding ->
            Box(modifier = Modifier.padding(padding))
            {
                AppNavigation(navController =
                    navController)
            }
        }
    )
}
```

12. Then, call this created function inside **`set-Content{}`**; the import should be **`import androidx.activity.compose.setContent`**, based on the fact that sometimes it might happen that you import the wrong one. Run the application. You will notice a screen with four tabs, and when you select a tab, the selected one gets highlighted, as shown in *Figure 4.3*:

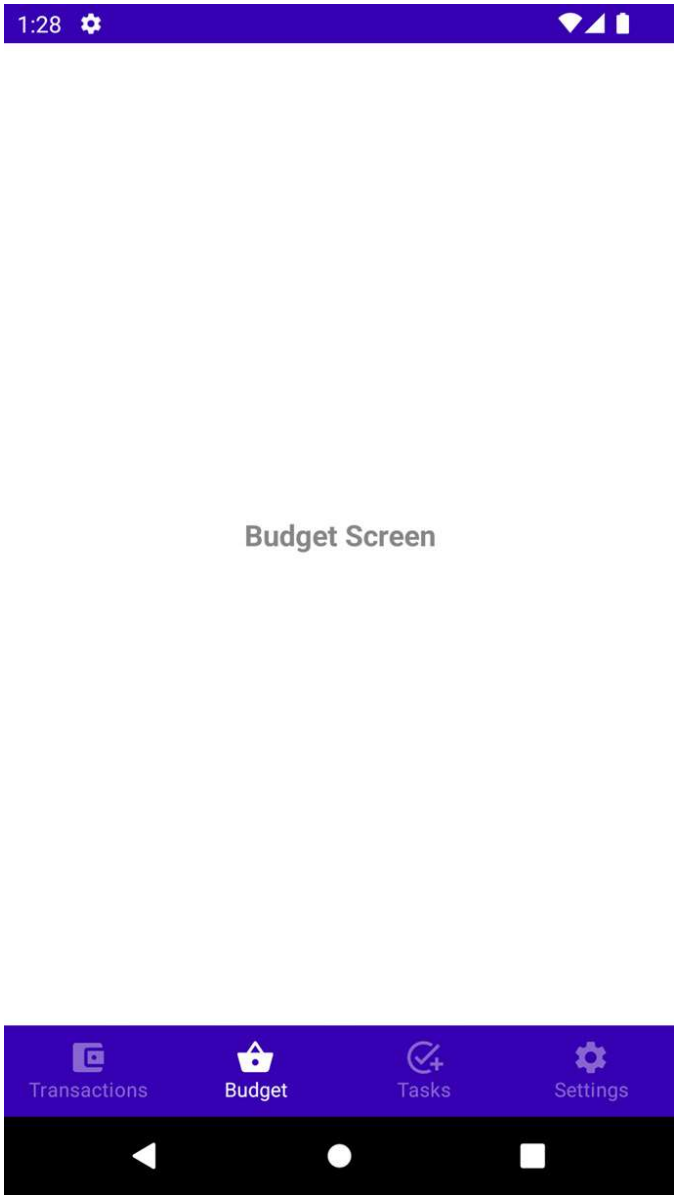

Figure 4.3 – The bottom navigation bar

How it works...

In Compose, navigation has a crucial term called **route**. The key is the string that defines the pathway to your composable. The key basically is the source of truth – or think of it as a deep link that takes you to a specific destination, and each destination should have a unique route.

Furthermore, each destination should consist of a unique key route. In our example, we added icons and a title. The icons, as seen in *Figure*

4.3, show what the bottom navigation entails, and the title describes the specific screen we are browsing at that exact time. In addition, these are optional and only needed for some routes.

`NavController()` is the main API for our navigation component, and it keeps track of every back stack entry for the composables that make up the screens in our application and the state of each screen. We created this using **`rememberNavController`**: as we mentioned in the previous chapter, **`remember`**, as the name suggests, remembers the value; in this instance, we are remembering **`NavController`**:

```
val navController = rememberNavController()
```

`NavHost()`, on the other hand, requires the **`NavController()`** previously created through **`rememberNavController()`** and the destination route of the entry point of our graph. In addition, **`rememberNavController()`** returns **`NavHostController`**, which is a subclass of **`NavController()`** that offers some additional APIs that **`NavHost`** requires.

This is very similar to how Android developers build navigation before composing fragments. The steps include creating a bottom navigation menu with the **`menu`** items, as shown in the following code block:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/transaction_home"
        android:icon="@drawable/card"
        android:title="@string/transactions"/>
```

```
        <item
             android:id="@+id/budget_home"
             android:icon="@drawable/ic_shopping_basket_black_
                 24dp"
             android:title="@string/budgets"
        />
    ...
    </menu>
```

Then, we create another resource in the **navigation** package that points to the screens (fragments):

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    app:startDestination="@+id/transaction_home">
    <fragment
        android:id="@+id/transaction_home"
        android:name="com.fragments.TransactionsFragment"
        android:label="@string/title_transcation"
        tools:layout="@layout/fragment_transactions" >
        <action
            android:id="@+id/action_transaction_home_to_
                budget_home"
            app:destination="@id/budget_home" />
    </fragment>
    <fragment
        android:id="@+id/budget_home"
        android:name="com.fragments.BudgetsFragment"
        android:label="@string/title_budget"
        tools:layout="@layout/fragment_budget" >
        <action
            android:id="@+id/action_budget_home_to_tasks_
                home"
            app:destination="@id/tasks_home" />
    </fragment>
</navigation>
```

Navigating to a new screen in Compose

We will build a register screen prompt on our login page for registering first-time users of our application. This is a standard pattern because we need to save the user's credentials so that the next time they log in to our application, we just log them in without registering again.

Getting ready

You should have completed the previous recipe, *Implementing a bottom navigation bar using navigation destinations*, before getting started with this one.

How to do it...

In this recipe, we will need to use our **`SampleLogin`** project and add a new screen that users can navigate to if it is their first time using the application. This is a typical use case in many applications:

1. Open your **`SampleLogin`** project, create a new **`sealed`** class, and call it **`Destination`**. To also ensure we maintain great packaging, add this class to **`util`**. Also, just like the bottom bar, we will have a route, but this time, we do not need any icons or titles:

```
sealed class Destination (val route: String){
    object Main: Destination("main_route")
    object LoginScreen: Destination("login_screen")
    object RegisterScreen:
        Destination("register_screen")
}
```

2. After creating the destinations, we now need to go ahead and add clickable text in **`LoginContent`** to ask users whether it is their first time using the application. They should

click **Register**. Then, we can navigate to **`RegisterContent`**. You can open the project by checking out the *Technical requirements* section if you need to refer to any step:

```
@Composable
fun LoginContent(
    ...
    onRegister: () -> Unit
) {
ClickableText(
    modifier = Modifier.padding(top = 12.dp),
    text = AnnotatedString(stringResource(id =
        R.string.register)),
    onClick = { onRegister.invoke() },
    style = TextStyle(
       colorResource(id = R.color.purple_700),
       fontSize = 16.sp
    )
)
```

3. Now, when you click on **`ClickableText`**, our clickable text is text that you can click, and it will help users navigate to the registration screen via **First-time user? Sign UP**. Once you click on this, it should navigate to a different screen where users can now sign up, as shown in *Figure 4.4*:

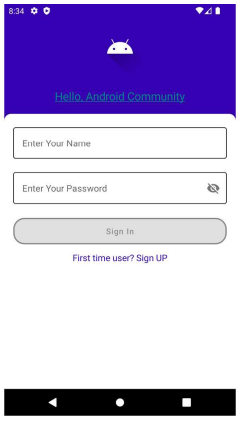

Figure 4.4 – A new Register screen

4. For the **Register** screen, you can get the entire code in the *Technical requirements* sec-

tion. We will reuse the user input fields that we created and just change the text:

```
@Composable
fun PasswordInputField(
    text: String
) {
    OutlinedTextField(
        label = { Text(text = text) },
...
}
```

5. In **`MainActivity`**, we will have a **`Navigation()`** function, as follows:

```
@Composable
fun Navigation(navController: NavHostController) {
    NavHost(navController, startDestination =
    Destination.LoginScreen.route) {
        composable(Destination.LoginScreen.route) {
            LoginContentScreen(loginViewModel =
                hiltViewModel(),
            onRegisterNavigateTo = {
                navController.navigate(
                    Destination.RegisterScreen.route)
            })
        }
        composable(Destination.RegisterScreen.route) {
            RegisterContentScreen(registerViewModel =
                hiltViewModel())
        }
    }
}
```

6. In **`PasswordInputField`**, we will name each input appropriately for reusability:

```
PasswordInputField(
    text = stringResource(id = R.string.password),
    authState = uiState,
    onValueChanged = onPasswordUpdated,
    passwordToggleVisibility =
        passwordToggleVisibility)
```

7. Moreover, you can also navigate to the previous **Sign in** screen by clicking on the hard-

ware **Back** button.

8. Finally, in **`setContent`**, we will need to update the code to accommodate the new navigation:

```
@AndroidEntryPoint
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?)
    {
        super.onCreate(savedInstanceState)
        setContent {
            SampleLoginTheme {
                // A surface container using the
                   'background' color from the theme
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color =
                       MaterialTheme.colors.background
                ) {
                    val navController =
                        rememberNavController()
                    Navigation(navController =
                        navController)
                }
            }
        }
    }
}
```

Run the code and click on the **Sign up** text, and you should now be taken to a new screen.

How it works...

You will notice that we have just created a different destination entry point, where **`ClickableText`** is used to navigate to the newly created screen. Furthermore, in order to navigate to a composable destination in the nav graph, you must use **`navController.navigate(Destination.RegisterScreen.route)`**,

and as mentioned earlier, the string represents the destination route.

In addition, **`navigate()`** adds our destination to the back stack by default, but if we need to modify the behavior, we can easily do that by adding additional nav options to our **`navigate()`** call.

Suppose you want to work with animations when navigating. In that case, you can easily do that by using the Accompanist library – **https://github.com/google/accompanist** – which offers a group of libraries that aim to supplement Jetpack Compose with features that are required mainly by developers and are not available yet.

You can utilize **`enterTransition`**, which explicitly specifies the animation that runs when you navigate to a particular destination, whereas **`exitTransition`** does the opposite:

```
AnimatedNavHost(
    modifier = Modifier
        .padding(padding),
    navController = navController,
    startDestination = Destination.LoginScreen.route,
    route = Destination.LoginScreen.route,
    enterTransition = { fadeIn(animationSpec = tween(2000)) },
    exitTransition = { fadeOut(animationSpec = tween(200))
    }
)
```

You can also use **`popEnterTransition`**, which specifies the animation that runs when the destination re-enters the screen after going through **`popBackStack()`**, or **`popExitTranstion`**, which does the opposite.

IMPORTANT NOTE

It is crucial to note a good practice that becomes relevant for hoisting the state is when you expose events from your composable functions to callers in your application that know how to handle that logic properly. In addition, under-the-hood navigation is all state-managed.

See also

For more about **`AnimatedNavHost`**, you can find details here: **https://google.github.io/accompanist/navigation-animation/**.

Navigating with arguments

Passing data between destinations is very vital in Android development. The new Jetpack navigation allows developers to attach data to a navigation operation by defining an argument for a destination. Readers will learn how to pass data between destinations using arguments.

A good use case is, say, you load an API with data and want to show more description on the data you just displayed; you can navigate with unique arguments to the next screen.

Getting ready

We will explore the most common interview project requirement, which is to fetch data

from an API and display one screen and add an additional screen for extra points.

Let's assume the API is the GitHub API, and you want to display all organizations. Then, you want to navigate to another screen and see the number of repositories each company has.

How to do it...

For this recipe, we will look at an example of navigating with arguments as a concept since there is little more to do other than create the basic arguments to pass to utilize an already-built project – **`SampleLogin`**:

1. Let's go ahead and create **`SearchScreen`**, and this screen will have just a search function, **`EditText`**, and a column to display the data returned from an API:

```
SearchScreen(
    viewModel = hiltViewModel(),
    navigateToRepositoryScreen = { orgName ->
        navController.navigate(
            Destination.BrowseRepositoryScreen.route +
            "/" + orgName
        )
    }
)
```

2. And now, when setting up navigation to **`BrowseRepository`**, you will need to add the following code. This piece of code is for passing a mandatory data parameter from one screen to another, but adding the example of passing an optional argument; a default value will help the user:

```
composable(
    route = Destination.BrowseRepositoryScreen.route +
            "/{org_name}",
```

```
            arguments = listOf(navArgument("org_name") { type
                = NavType.StringType }),
            enterTransition = { scaleIn(tween(700)) },
            exitTransition = { scaleOut(tween(700)) },
        ) {
            BrowseRepositoryScreen(
                viewModel = hiltViewModel(),
            )
        }
```

We also use the **`enter`** and **`exit`** transition for animations. In this recipe, we have just touched on the concept of navigating with arguments, which can be applied to many projects.

How it works...

When you want to pass an argument to the destination, which is something that might be required, you need to explicitly attach it to the route when initiating the **`navigate`** function call, as you can see in the following code snippet:

```
navController.navigate(Destination.BrowseScreen.route + "/" + orgName)
```

We added an argument placeholder to our route, similar to how we added arguments to a deep link when using the base navigation library.

There is also a list of what the navigation library supports; if you have a different use case, you can look into this document: **https://developer.android.com/guide/navigation/navigation-pass-data#supported_argument_types**.

The Navigation library supports the following argument types:

Type	app:argType syntax	Support for default values	Handled by routes	Nullable
Integer	app:argType="integer"	Yes	Yes	No
Float	app:argType="float"	Yes	Yes	No
Long	app:argType="long"	Yes - Default values must always end with an 'L' suffix (e.g. "123L").	Yes	No
Boolean	app:argType="boolean"	Yes - "true" or "false"	Yes	No
String	app:argType="string"	Yes	Yes	Yes
Resource Reference	app:argType="reference"	Yes - Default values must be in the form of "@resourceType/resourceName" (e.g. "@style/myCustomStyle") or "0"	Yes	No
Custom Parcelable	app:argType="<type>", where <type> is the fully-qualified class name of the `Parcelable`	Supports a default value of "@null". Does not support other default values.	No	Yes
Custom Serializable	app:argType="<type>", where <type> is the fully-qualified class name of the `Serializable`	Supports a default value of "@null". Does not support other default values.	No	Yes
Custom Enum	app:argType="<type>", where <type> is the fully-qualified name of the enum	Yes - Default values must match the unqualified name (e.g. "SUCCESS" to match MyEnum.SUCCESS).	No	No

Figure 4.5 – Navigation support argument type (credit: developers.android.com)

There's more...

There is more to learn about navigation and, for a more thorough way to look into how you can navigate with arguments, retrieve complex data when navigating, and add additional arguments in depth, you can read more here: **https://developer.android.com/jetpack/compose/navigation**.

Creating deep links for destinations

In Modern Android Development, deep links are very vital. A link that helps you navigate directly to a specific destination within an app is called a **deep link**. The `Navigation` component lets you create two types of deep links: **explicit** and **implicit**.

Compose navigation supports implicit deep links, which can be part of your Composable functions. It is also fair to mention there is no huge difference between how you would handle these using XML layouts.

Getting ready

Since we don't have a deep link use case in our application, in this recipe, we will look into how we can utilize the knowledge by learning how to implement implicit deep links.

How to do it...

You can match deep links using a **Uniform Resource Locator** (**URI**), intent actions, or **Multipurpose Internet Mail Extensions** (**MIME**) types. Furthermore, you can easily specify multiple types that match for a deep link single but remember that the URI argument comparison is always prioritized, followed by the intent action, then the MIME type.

Compose has made it easier for developers to work with deep links. The `composable` function accepts a parameter list of `NavDeepLinks`, which can be easily created using the `navDeepLink` method:

1. We will start by making the deep link externally available by adding the appropriate intent filter to our `AndroidManifest.xml` file:

```
<activity>
    <intent-filter>
      ...
      <data android:scheme="https"
      android:host="www.yourcompanieslink.com" />
```

```
        </intent-filter>
    </activity>
```

2. Now in our **`composable`** function, we can use the **`deepLinks`** parameter, specify the list of **`navDeepLink`**, and then pass the URI pattern:

```
val uri = "www.yourcompanieslink.com"
composable(deepLinks = listOf(navDeepLink { uriPattern = "$uri/{id}" }))
{...}
```

3. Please note that navigation will automatically deep-link into that composable when another application triggers the deep link.

Many applications still use **`launchMode`** when navigating. This is when using the Navigation Jetpack component, as seen in the following code snippet:

```
override fun onNewIntent(intent: Intent) {
    super.onNewIntent(intent)
    navigationController.handleDeepLink(intent)
}
```

4. Finally, you can also utilize **`deepLinkPendingIntent`** like any other **`PendingIntent`** to start your Android application at the deep link destination.

IMPORTANT NOTE

When triggering an implicit deep link, the back stack state depends on when the implicit intent was launched with **`Intent.FLAG_ACTIVITY_NEW_TASK`**. *Furthermore, if the flag is set, the back stack task is cleared and then replaced with the intended deep link destination.*

How it works...

In Android development, a deep link refers to a specific destination for the application. For instance, when you invoke the deep link, the link opens up your application's corresponding destination when a user clicks on the specified link.

This refers to where the link is meant to lead once clicked. The explicit deep link is a single instance that uses **`PendingIntent`** to take users to a specific location within your application. A good use case is when using notifications or app widgets.

There's more...

There is more to learn about deep links; for instance, how to create an explicit deep link. You can learn more about deep links here: **https://developer.android.com/training/app-links/deep-linking**.

Writing tests for navigation

Now that we have created a new screen for our **`SampleLogin`** project, we need to fix the broken test and add new tests for the UI package. If you can recall, in *Chapter 3*, *Handling the UI State in Jetpack Compose and Using Hilt*, we did unit tests and not UI tests. This means after adding all the **`ViewModel`** instances, our UI tests are now broken. In this recipe, we will fix the failing tests and add a navigation test.

Getting ready

In this recipe, you do not need to create any new project; use the already-created project, **SampleLogin**.

How to do it...

You can apply these concepts to test the bottom navigation bar we created. Hence, we will not be writing tests for the **BottomNavigationBarSample** project. Open **SampleLogin** and navigate to the **androidTest** package. We will add tests here for the new **RegisterScreen()** a Composable function, and also fix the broken tests:

1. Let's open the **LoginContentTest** class. Now, let's move the **LoginContent** class to a helper class that we will create to help us with testing the UI logic:

```
@Composable
fun contentLoginForTest(
    uiState: AuthenticationState =
    AuthenticationState(),
    onUsernameUpdated : (String) -> Unit = {},
    onPasswordUpdated :(String) -> Unit = {},
    onLogin : () -> Unit = {},
    passwordToggleVisibility: (Boolean) -> Unit = {},
    onRegisterNavigateTo: () -> Unit = {}
) {
    LoginContent(
        uiState = uiState,
        onUsernameUpdated = onUsernameUpdated,
        onPasswordUpdated = onPasswordUpdated,
        onLogin = onLogin,
        passwordToggleVisibility =
            passwordToggleVisibility,
        onRegister = onRegisterNavigateTo
```

```
        )
    }
```

2. Inside the **`LoginContentTest`** class, now, we will replace **`LoginContent`** with the newly created **`contentLoginForTest()`** function inside the **`initCompose`** function:

```
private fun initCompose() {
    composeRuleTest.activity.setContent {
        SampleLoginTheme {
            contentLoginForTest()
            launchRegisterScreenWithNavGraph()
        }
    }
}
```

3. Now that we have fixed the tests, we can now add a **`test`** tag for our newly created clickable, **`TextView`**:

```
const val REGISTER_USER = "register_user"
```

4. Once that is done, we now need to create **`lateint var NavHostController`**, and a **`launchRegisterScreenWithNavGraph`** function to help us set up the navigation:

```
private fun launchRegisterScreenWithNavGraph() {
    composeRuleTest.activity.setContent {
        SampleLoginTheme {
            navController = rememberNavController()
            NavHost(
                navController = navController,
                startDestination =
                    Destination.LoginScreen.route
            ) {
                composable(Destination.LoginScreen
                .route) {
                LoginContentScreen(
                onRegisterNavigateTo = {
                    navController.navigate(
                        Destination.RegisterScreen
                        .route)
                }, loginViewModel = hiltViewModel())
```

```
                    }
                    composable(
                        Destination.RegisterScreen
                        .route) {
                            RegisterContentScreen(
                                hiltViewModel())
                        }
                }
            }
        }
    }
```

You can call the created function inside the **`initCompose`** function or in the new test function that we will create.

5. Now, let's create a test function and name it **`assertRegisterClickableButtonNavigatesToRegisterScreen()`**. In this test case, we will set our route and then use **`assert`** when the correct **`TextView`** is clicked; we will navigate to the correct destination:

```
@Test
fun assertRegisterClickableButtonNavigatesToRegisterScreen() {
    initCompose()
    composeRuleTest.onNodeWithTag(
        TestTags.LoginContent.REGISTER_USER)
            .performClick(
    )
    val route =
        navController.currentDestination?.route
    assert(route.equals(
        Destination.RegisterScreen.route))
}
```

6. Finally, run the test, and the UI test should pass, as seen in *Figure 4.6*:

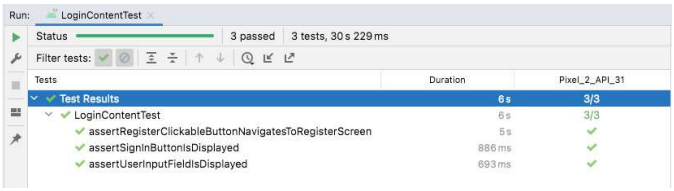

Figure 4.6 – Tests passing

How it works...

We created the **`contentLoginForTest`** that can help us verify our navigation. That is, when a user enters a valid username and password, they can navigate to a home screen. Furthermore, we created **`launchRegister-ScreenWithNavGraph()`**, a helper function that creates the testing graph for our navigation test case.

If you are using **`FragmentScenario`**, there are great tips for testing your navigation that you can see here: **https://developer.android.com/guide/navigation/navigation-testing**.