Chapter 11: Conclusion and Next Steps

This book shows you how to write beautiful, fast, and maintainable Jetpack Compose apps. In Chapters 1 to 3, I introduced you to the fundamentals of Jetpack Compose, explained core techniques and principles, as well as important interfaces, classes, packages, and, of course, composable functions. Chapters 4 to 7 focused on building Compose UIs. You learned how to manage state and navigate to different screens. We also explored the ViewModel and Repository patterns. Chapters 8 to 10 covered advanced topics such as animation, interoperability, testing, and debugging.

This final chapter is all about what you can do next. We'll investigate the near future of Jetpack Compose and explore neighboring platforms, because you can apply your Compose knowledge there, too. The main sections of this chapter are the following:

- Exploring the future
- Migrating to Material You
- Moving beyond Android

We'll start by looking at the next version of Jetpack Compose, 1.1, which was not yet stable when this book went into production. This iteration will bring bug fixes, performance improvements, and new features, for example, **`ExposedDropdownMenuBox()`**, an exposed drop-down menu, and **`NavigationRail()`**. This vertical navigation bar is intended for foldables and large-screen devices.

The second main section, *Migrating to Material You*, introduces you to Material 3 for Compose. This package contains *Material You*, the latest iteration of Google's beautiful design language, to Jetpack Compose apps.

We'll look at some differences between Material 2 and Material 3, for example, the simplified typography and color schemes.

The *Moving beyond Android* section shows you how to use your Jetpack Compose knowledge on other platforms, for example, desktop and the web. I will briefly explain how to bring one of my sample composable functions to desktop.

Technical requirements

This chapter is based on the **`ExposedDropdownMenuBoxDemo`** and **`NavigationRailDemo`** samples. Please refer to the *Technical requirements* section in *Chapter 1*, *Building Your First Compose App*, for information about how to install and set up Android Studio, and how to get the repository accompanying this book.

All the code files for this chapter can be found on GitHub at https://github.com/PacktPublishing/Android-UI-Development-with-Jetpack-Compose/tree/main/chapter_11.

Exploring the future

This book is based on Jetpack Compose 1.0, the first stable version of the library, which was released in July 2021. Just like all other Jetpack components, Google is constantly enhancing and updating Compose. At the time of finishing the manuscript, version 1.1 was in beta. When it becomes stable, I will update the repository accompanying this book to reflect the changes. You can find the latest version of the samples of this book at https://github.com/PacktPublishing/Android-UI-Development-with-Jetpack-Compose.

Jetpack Compose 1.1 will offer bug fixes, new functionality, and performance improvements. New features include the following:

- The Compose compiler will support older versions of the Compose runtime. This allows you to use the latest tooling while still targeting older Compose versions.
- Touch target sizing (UI elements may get extra spacing to make them more accessible).
- **`ImageVector`** caching.
- Support for Android 12 stretch overscroll.

Several previously experimental APIs (for example, **`AnimatedVisibility`**, **`EnterTransition`**, and **`ExitTransition`**) will become stable. Additionally, Jetpack Compose 1.1 will support newer versions of Kotlin. Unfortunately, you will also face some breaking changes. For example, lambdas in **`EnterTransition`** and **`ExitTransition`** factories may be moved to the last position in the parameter list.

Showing exposed drop-down menus

There are also new Material UI elements. For example, **`ExposedDropdownMenuBox()`** shows an exposed drop-down menu, which displays the currently selected menu item above the list of options. The **`ExposedDropdownMenuBoxDemo`** sample illustrates the usage of the composable function (*Figure 11.1*).

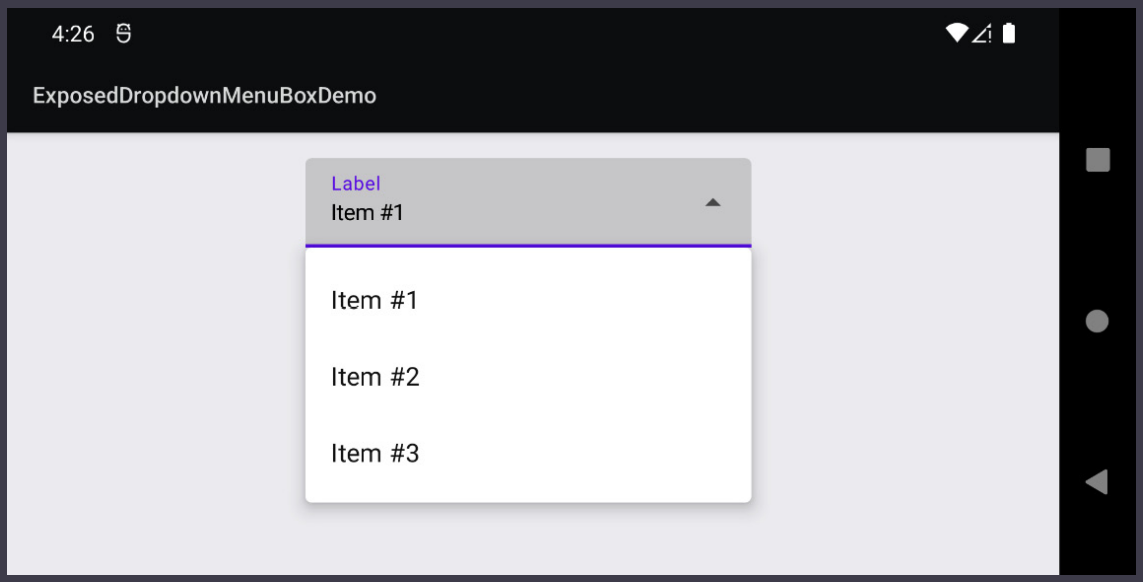

Figure 11.1 – The ExposedDropdownMenuBoxDemo sample

Currently, **`ExposedDropdownMenuBox()`** is marked experimental. Therefore, you must add the **`@ExperimentalMaterialApi`** annotation:

```
@ExperimentalMaterialApi
@Composable
fun ExposedDropdownMenuBoxDemo() {
  val titles = List(3) { i ->
    stringResource(id = R.string.item, i + 1)
  }
  var expanded by remember { mutableStateOf(false) }
  var selectedTxt by remember {
mutableStateOf(titles[0]) }
  Box(
    modifier = Modifier
      .fillMaxSize()
      .padding(16.dp),
    contentAlignment = Alignment.TopCenter
  ) {
    ...
  }
}
```

`ExposedDropdownMenuBoxDemo()` puts **`ExposedDropdownMenuBox()`** in a **`Box()`** and horizontally centers the menu at the top. The menu items are stored in a list (**`titles`**). The **`expanded`** state reflects the visibility of the menu items. **`selectedTxt`** represents the currently selected text. Here's how they are used:

```
ExposedDropdownMenuBox(expanded = expanded,
  onExpandedChange = {
    expanded = !expanded
  }) {
  TextField(value = selectedTxt,
    onValueChange = { },
    readOnly = true,
    label = {
```

```
        Text(text = stringResource(id =
  R.string.label))
      },
      trailingIcon = {
        ExposedDropdownMenuDefaults.TrailingIcon(
          expanded = expanded
        )
      }
    )
    ExposedDropdownMenu(expanded = expanded,
      onDismissRequest = {
        expanded = false
      }) {
      for (title in titles) {
        DropdownMenuItem(onClick = {
          expanded = false
          selectedTxt = title
        }) {
          Text(text = title)
        }
      }
    }
  }
```

ExposedDropdownMenuBox() has two children, read-only **TextField()** and **ExposedDropdownMenu()**. The text field shows **selectedTxt**. As **readOnly** is set to **true**, the **onValueChange** block can be empty. **expanded** controls the trailing icon, which reflects the visibility of the menu items. The **onExpandedChange** lambda expression passed to **ExposedDropdownMenuBox()** is executed when the user clicks on the exposed drop-down menu. Usually, you will negate **expanded**.

ExposedDropdownMenu() has at least one **DropdownMenuItem()** as its content. Typically, you will want to hide the menu (**expanded = false**) and update the text field (**selectedTxt = title**). The **onDismissRequest** block passed to

`ExposedDropdownMenu()` should also close the menu, but not update the text field.

So, `ExposedDropdownMenuBox()` is a very compact way of showing a selection of items and allowing the user to choose one. In the following section, I show you another Material UI element that debuts in Compose 1.1. `NavigationRail()` presents top-level navigation destinations vertically.

Using NavigationRail()

Compose offers several ways to navigate to top-level destinations within your app. For example, you can place a navigation bar at the bottom of the screen using `BottomNavigation()`. I show you how to use it in the *Adding navigation* section of *Chapter 6*, *Putting Pieces Together*. Jetpack Compose 1.1 includes another UI element for top-level navigation. `NavigationRail()` implements the **navigation rail** interaction pattern, a vertical navigation bar especially for large screens such as tablets and open foldables.

If the screen is not big enough, or the foldable is closed, a standard bottom navigation bar should be displayed instead. The `NavigationRailDemo` sample shows how to achieve this. In *Figure 11.2*, you can see the app in portrait mode.

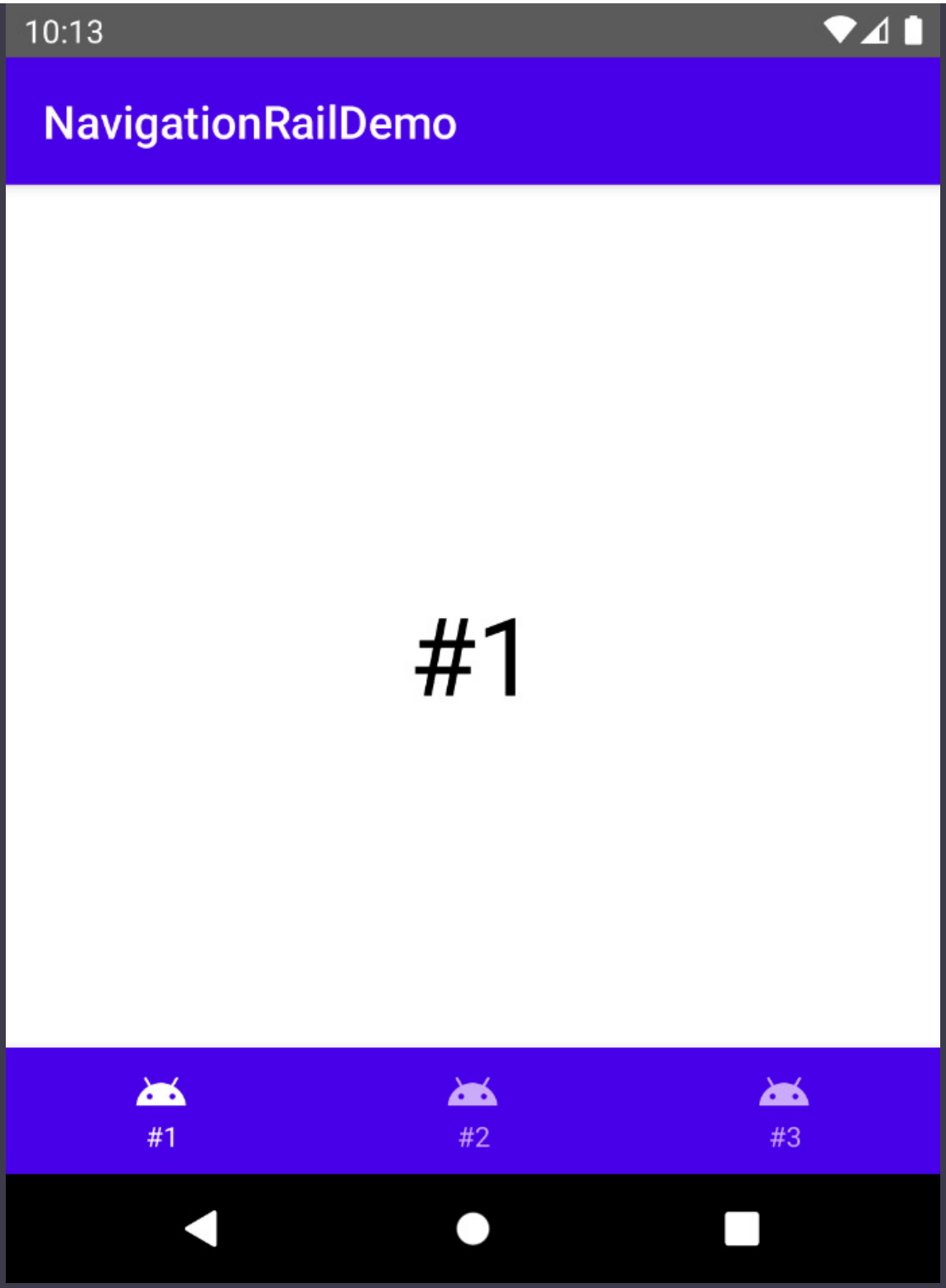

Figure 11.2 – The NavigationRailDemo sample in portrait mode

To continue, an elaborate approach would be to use the Jetpack **`WindowManager`** library, however this is beyond the scope of the book. Instead, we will use **`NavigationRailDemo ( )`** for the sake of simplicity, which determines whether the navigation rail should be used by simply

comparing the current width of the screen with the minimum size (600 density-independent pixels):

```
@Composable
fun NavigationRailDemo() {
  val showNavigationRail =
          LocalConfiguration.current.screenWidthDp >=
600
  val index = rememberSaveable { mutableStateOf(0) }
  Scaffold(topBar = {
    TopAppBar(title = {
      Text(text = stringResource(id =
R.string.app_name))
    })
  },
    bottomBar = {
      if (!showNavigationRail)
        BottomBar(index)
    }) {
    Content(showNavigationRail, index)
  }
}
```

`Scaffold()` receives bottom bars through the **`bottomBar`** lambda expression. If the navigation rail should not be shown (**`showNavigationRail`** is **`false`**), my **`BottomBar()`** composable is invoked. Otherwise, no bottom bar is added. The currently active screen is stored in a mutable **`Int`** state (**`index`**). It is passed to **`BottomBar()`** and **`Content()`**. Next, let's briefly revisit how **`BottomNavigation()`** works by looking at my **`BottomBar()`** composable:

```
@Composable
fun BottomBar(index: MutableState<Int>) {
  BottomNavigation {
    for (i in 0..2)
      BottomNavigationItem(selected = i ==
index.value,
```

```
          onClick = { index.value = i },
          icon = {
            Icon(
              painter = painterResource(id =
                  R.drawable.ic_baseline_android_24),
              contentDescription = null
            )
          },
          label = {
            MyText(index = i)
          }
        )
    }
  }
```

The content of **`BottomNavigation()`** consists of several **`BottomNavigationItem()`** elements with an icon, a label, and an **`onClick`** block. My implementation just updates the **`index`** state, which is also used inside **`Content()`**. This composable displays the navigation rail if needed, and the main content (screen), which is just a box with text centered inside:

```
  @Composable
  fun Content(showNavigationRail: Boolean, index:
     MutableState<Int>) {
    Row(
      modifier = Modifier.fillMaxSize()
    ) {
      if (showNavigationRail) {
        NavigationRail {
          for (i in 0..2)
            NavigationRailItem(selected = i ==
  index.value,
              onClick = {
                index.value = i
              },
```

```
            icon = {
              Icon(
                painter = painterResource(id =
                    R.drawable.ic_baseline_android_24
),
                contentDescription = null
              )
            },
            label = {
              MyText(index = i)
            })
      }
    }
    Box(
      modifier = Modifier
        .fillMaxSize()
        .background(color =
MaterialTheme.colors.surface),
      contentAlignment = Alignment.Center
    ) {
      MyText(
        index = index.value,
        style = MaterialTheme.typography.h3
      )
    }
  }
}
```

The navigation rail and the screen are arranged horizontally in **`Row()`**. Like **`BottomNavigation()`**, **`NavigationRail()`** gets one or more child elements that represent the navigation destinations. The children (**`NavigationRailItem()`**) have a label, an icon, and an **`onClick`** block. *Figure 11.3* shows the **`NavigationRailDemo`** sample in landscape mode.

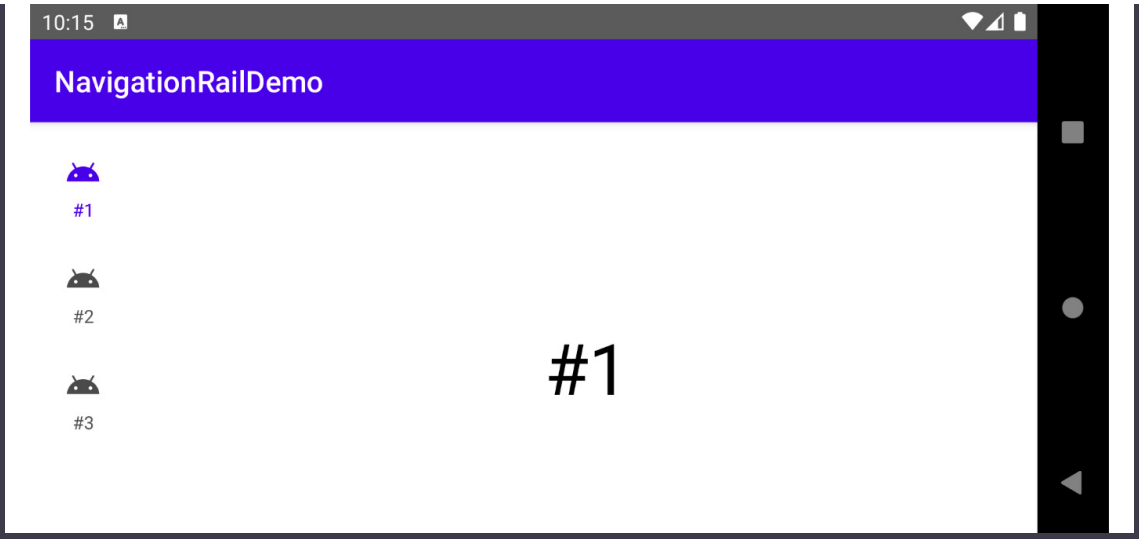

Figure 11.3 – The NavigationRailDemo sample in landscape mode

While Jetpack Compose 1.1 will add some Material UI elements and polish the existing ones, it still implements *Material Design* as present in previous Android versions, including 11 (sometimes referred to as Material 2). *Material You,* which debuted with Android 12, will be available for Compose, too. However, it is not an in-place update of the existing packages but comes as a new library. In the following section, we look at Material 3 for Jetpack Compose, which was in early alpha at the time this chapter was written.

NOTE

You may be wondering what the difference between Material You and Material 3 is. I am referring to Material 3 as the latest version of the Material Design specification, whereas Material You is the implementation on Android 12.

Migrating to Material You

Material You is the latest iteration of Google's design language Material Design. It was announced during Google I/O 2021 and was first available on Pixel smartphones running Android 12. Eventually, it will be rolled out to other devices, form factors, and frameworks. Like its predecessors, Material You is based on typography, animation, and layers. But it empha-

sizes personalization: depending on the platform, *Material You* implementations may use color palettes derived from the system wallpaper.

Looking at some differences between Material 2 and Material 3 for Compose

To use *Material You* in your Compose app, you must add an implementation dependency to **`androidx.compose.material3:material3`** in the module-level **`build.gradle`** file. The base package for composables, classes, and interfaces changes to **`androidx.compose.material3`**. If you want to migrate an existing Compose app to this new version, you at least need to change imports. Unfortunately, the names of quite a few composable functions will change, too. To get an idea of the differences, I have reimplemented **`NavigationRailDemo`** for *Material You*. The project is named **`NavigationRailDemo_Material3`**. This way, you can easily examine the changes by comparing important files.

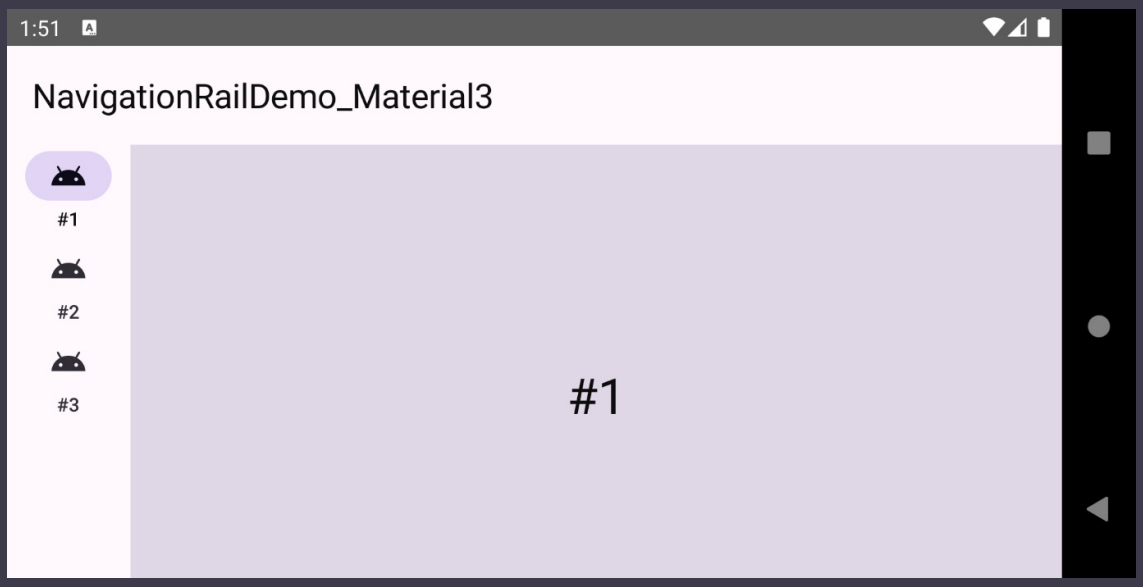

Figure 11.4 – The NavigationRailDemo_Material3 sample in landscape mode

Specifically, **`TopAppBar()`** needs to be replaced by **`SmallTopAppBar()`** or one of its bigger siblings, **`MediumTopAppBar()`** and **`LargeTopAppBar()`**. Other changes include the following:

- **`BottomNavigation()`** will be generalized to **`NavigationBar()`**.
- **`BottomNavigationItem()`** is now called **`NavigationBarItem()`**.
- **`NavigationRailItem()`** remains unchanged.

The last bullet point is interesting: as **`NavigationRailItem()`** elements very much resemble **`NavigationBarItem()`**, I wonder if these two may be generalized in the future.

Several properties that control the visual representation of UI elements will change considerably. For example, Material colors belong to **`MaterialTheme.colorScheme`** instead of the former **`MaterialTheme.colors`**. For more information about colors in Material 3, please refer to the official documentation at https://m3.material.io/styles/color/dynamic-color/overview.

Styled texts may also require some adaptions because the members of the **`Typography`** class will be simplified. For example, instead of **`h1`**, **`h2`**, **`h3`**, and so on, you will use **`headlineLarge`**, **`headlineMedium`**, or **`headlineSmall`**.

This concludes our brief look at the changes regarding Material 3 and the near future of Jetpack Compose. Did you know you can write Compose apps for the web and desktop, too? In the following section, we give it a try.

Moving beyond Android

While Jetpack Compose is the new UI toolkit on Android, its underlying ideas and principles make it attractive for other platforms, too. Let's see why this is the case:

1. The declarative approach was first implemented on the web.
2. SwiftUI, Apple's implementation of a declarative UI framework, works well for iPhones, iPads, watches, and macOS devices.
3. Jetpack Compose UI elements use Material Design, which is designed for different platforms, device categories, and form factors.

Most importantly, core concepts such as state and composable functions are not Android-specific. Therefore, if someone provides the toolchain (for example, the Kotlin compiler and the Compose compiler), any platform capable of showing graphics *may* be able to execute Compose apps. Certainly, there is an awful lot of work to be done.

For example, the Compose UI must be hosted *somewhere*. On Android, activities are used. On the web, this would be a browser window. And on desktop, it would be a window provided by some UI toolkit. Any other functionality (for example, network and file I/O, connectivity, memory management, threading) must be addressed by other libraries or frameworks.

JetBrains, the inventor of Kotlin and IntelliJ, decided to tackle this. In recent years, the company gained a lot of experience in targeting multiple platforms and sharing code among them. For example, with *Kotlin Multiplatform Mobile* you can use a single code base for the business logic of iOS and Android apps. *Compose Multiplatform* aims to simplify and speed up the development of UIs for desktop and the web, and to share UI code among them and Android.

In the following section, I will briefly show how to create a simple Compose for Desktop application using the IntelliJ IDE.

Setting up a sample project

The easiest way to create a Compose for Desktop project is to use the project wizard of the IntelliJ IDE. This requires IntelliJ IDEA Community Edition or Ultimate Edition 2020.3 or later. Setting up IntelliJ is beyond the scope of this book and is not detailed here. *Figure 11.5* shows you how to fill in the project wizard dialog.

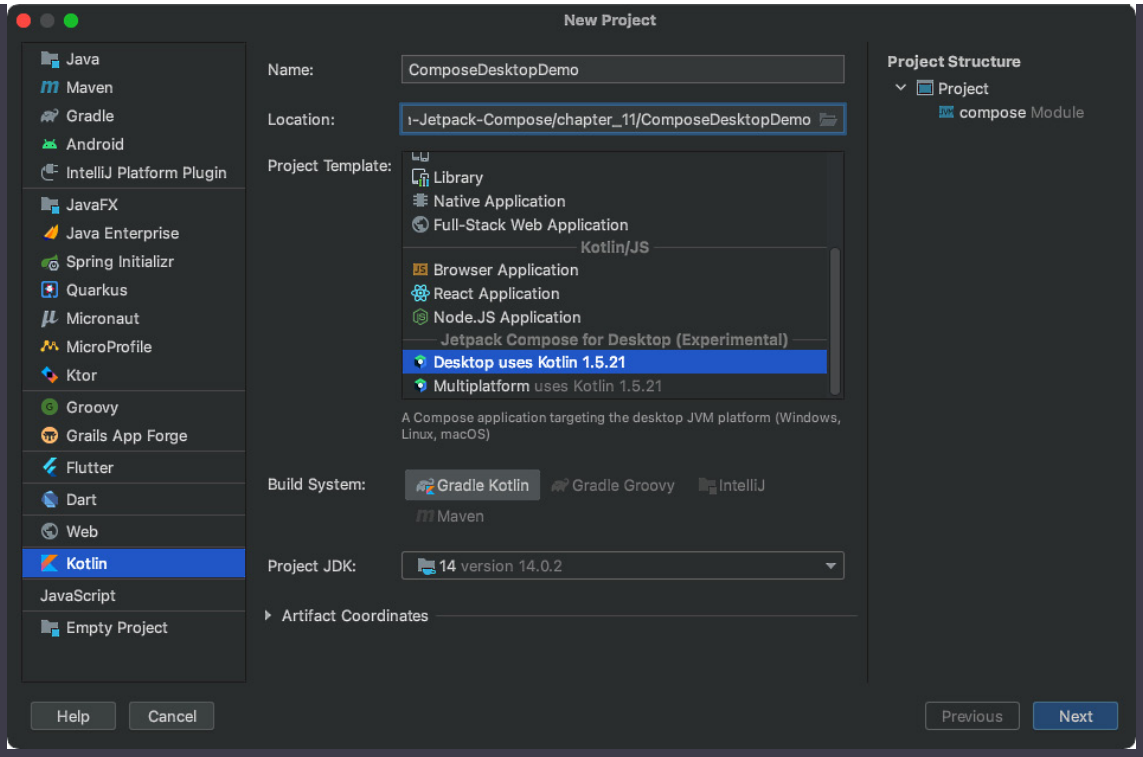

Figure 11.5 – The IntelliJ project wizard

JetBrains maintains a *Getting started with Compose Multiplatform* tutorial on GitHub at https://github.com/JetBrains/compose-jb/blob/master/tutorials/Getting_Started/README.md. Please refer to this for additional information.

The project wizard adds a simple **`Main.kt`** file inside **`src/main/kotlin`**. You can run it from the **Gradle** tool window by double-clicking on **Tasks** | **compose desktop** | **run** (*Figure 11.6*).

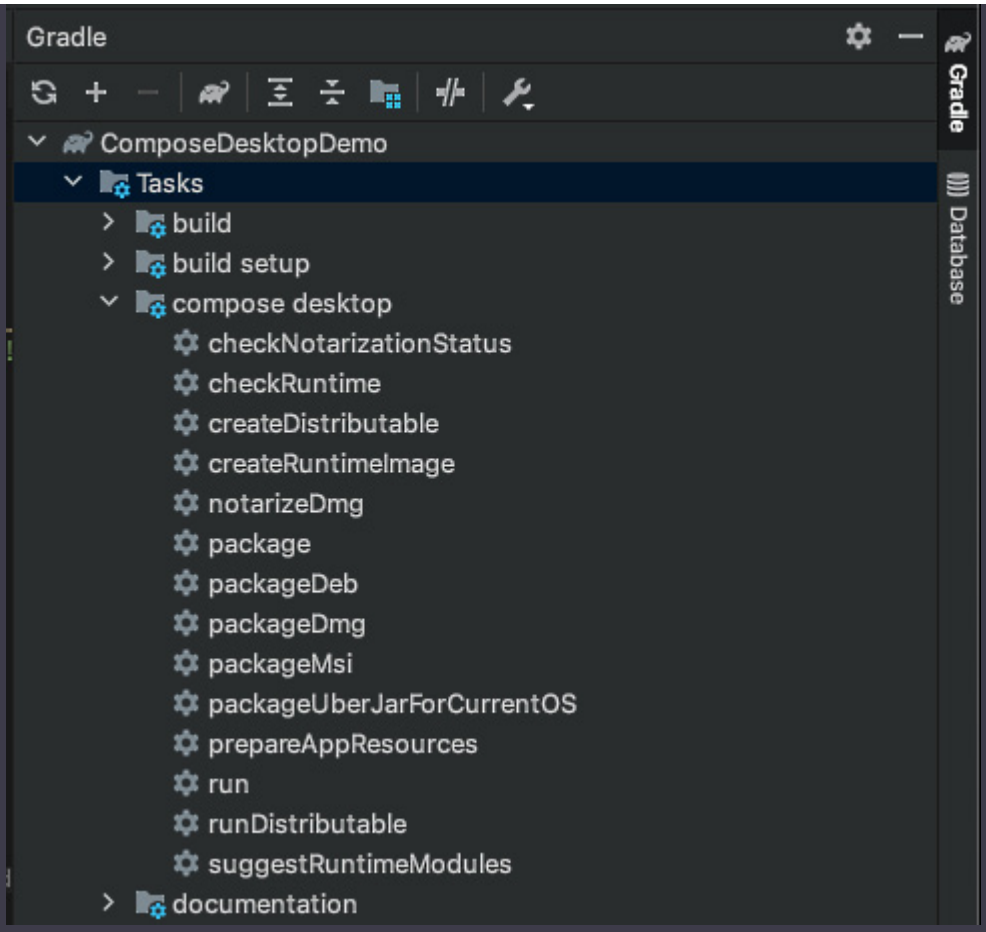

Figure 11.6 – The IntelliJ Gradle tool window

The source code contains a composable called **`App()`**. It is invoked from the **`main()`** function. Let's replace the body of **`App()`** with one of my samples, for example, **`StateChangeDemo()`** from *Chapter 8*, *Working with Animations*:

```
@Composable
@Preview
fun App() {
  var toggled by remember {
    mutableStateOf(false)
  }
  val color = if (toggled)
    Color.White
  else
    Color.Red
  Column(
```

```
      modifier = Modifier
        .fillMaxSize()
        .padding(16.dp),
      horizontalAlignment =
  Alignment.CenterHorizontally
    ) {
      Button(onClick = {
        toggled = !toggled
      }) {
        Text(text = "Toggle")
      }
      Box(
        modifier = Modifier
          .padding(top = 32.dp)
          .background(color = color)
          .size(128.dp)
      )
    }
  }
```

Have you noticed that I changed one line? The original version uses the **`stringResource()`** composable. However, Android resources are not available on desktop so you must replace the invocation with something different. A simple workaround is to hardcode the text. Real-world applications may want to choose a mechanism that supports multiple languages. Compose for Desktop relies on the Java Virtual Machine, so you can use Java's internationalization support.

The app running on macOS is shown in *Figure 11.7*.

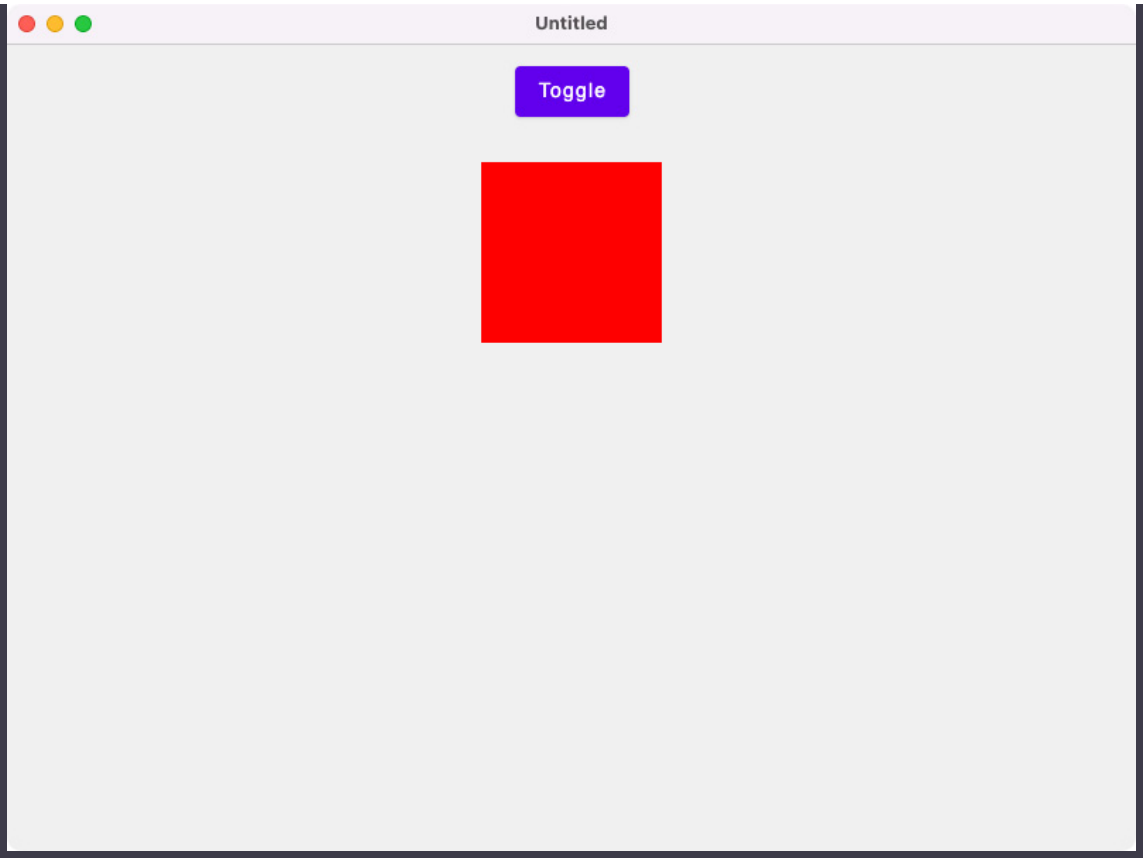

Figure 11.7 – A simple Compose for Desktop app

This concludes our brief look at Compose for Desktop and Compose Multiplatform. To learn more, please visit the product page at https://www.jetbrains.com/de-de/lp/compose-mpp/.

Summary

In this final chapter, we looked at the near future of Jetpack Compose and glimpsed neighboring platforms. Jetpack Compose 1.1 will bring bug fixes, performance improvements, and new features, for example, **`ExposedDropdownMenuBox()`** and **`NavigationRail()`**. Two samples (**`ExposedDropdownMenuBoxDemo`** and **`NavigationRailDemo`**) show you how to use them.

The second main section, *Migrating to Material You*, introduced you to Material 3 for Compose. This package brings *Material You*, the latest iteration of Google's beautiful design language, to Jetpack Compose apps. We

looked at some differences between Material 2 and Material 3, for example, the simplified typography and color schemes.

Moving beyond Android showed you how to use your Jetpack Compose knowledge on another platform. I explained how to bring one of my sample composable functions to desktop.

I sincerely hope you enjoyed reading this book. You now have a thorough understanding of the core principles of Jetpack Compose, as well as the important advantages over the traditional Android View system. Using a declarative approach makes writing great-looking apps easier than ever. I can't wait to see which beautiful ideas you are going to turn into code.