9

Building for Large Screens

We can all agree now that we live in a world with foldable phones, a technology we never anticipated, due to their growing demand and popularity. Ten years ago, if you had told a developer we would have foldable phones, no one would have believed it due to the ambiguity of screen complexity and the transfer of information.

However, now the devices are here with us. And since some of these devices run on the Android operating system, it's vital to know how we developers will build our applications to cater to foldability, along with the number of Android tablets we're now seeing on the market. The support for large screens seems now mandatory, and in this chapter, we will look at supporting large screens in the new Modern Android Development.

In this chapter, we'll be covering the following recipes:

- Building adaptive layouts in Modern Android Development
- Building adaptive layouts using `ConstraintLayouts`

- Handling large-screen configuration changes and continuity
- Understanding activity embedding
- Material Theming in Compose
- Testing your applications on a foldable device

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_nine**.

Building adaptive layouts in Modern Android Development

When building the UI for your application in Modern Android Development, it is fair to say that you should consider ensuring the application is responsive to different screen sizes, orientations, and form factors. Finally, developers can now remove the lock in portrait mode. In this recipe, we will utilize ideas we learned from previous recipes and build an adaptive app for different screen sizes and orientations.

Getting ready

We will be using the cities application to create a traveler profile, and our screen should be able to change based on different screen sizes and support foldable devices and tablets. To get the entire code, check out the *Technical requirements* section.

How to do it...

For this recipe, we will create a new project, and this time, instead of picking the empty Compose Activity template, we will pick empty Compose Activity (**`Material 3`**). **`Material 3`** seeks to improve our application's look and feel in Android. It includes updated theming, components, and great features, such as Material You personalization using dynamic color:

1. Let's start by creating an Empty Compose Activity (**`Material3`**) project and calling it **`Traveller`**; note that you can call your project anything you wish.

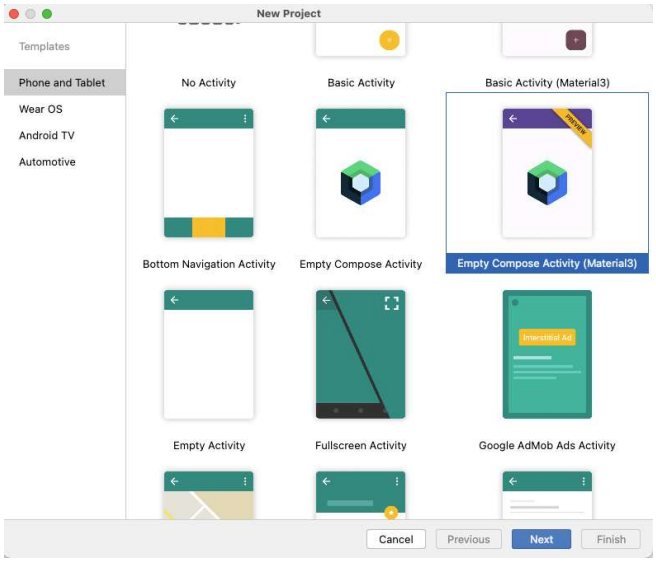

Figure 9.1 – Creating a new Empty Compose Activity (Material3) project

Complex applications utilize responsive UI, and in most cases, ensuring you choose the right navigation type for your application comes in handy. The Material library offers navigation components to developers, such as the bottom navigation, navigation drawer, and navigation rail. You can get the starter code for these in the *Technical requirements* section.

2. Add the following dependency, and check the project for the correct version number, **`1.1.0`**:

```
implementation "androidx.compose.material3:material3-window-size-class:1.1.0"
```

3. When ensuring our code caters to adaptability, we have to remember a responsive UI retains data when a phone is rotated, folded, or unfolded. The most crucial part is ensuring we handle the posture. We will create a function, **`cityPosture`**, that takes **`FoldingFeature`** as input and returns a Boolean:

```
@OptIn(ExperimentalContracts::class)
fun cityPosture(foldFeature: FoldingFeature?): Boolean {
    contract { returns(true) implies (foldFeature !=
        null) }
    return foldFeature?.state ==
        FoldingFeature.State.HALF_OPENED &&
            foldFeature.orientation ==
                FoldingFeature.Orientation.VERTICAL
}
```

We handle the state based on the three provided states. We also annotate it with the experimental class because this API is still experimental, which means it can change in the future and is not very stable.

4. Next, we need to cover **isSeparating**, which listens to the **FLAT** *fully open* and the **isSeparating** Boolean, which calculates whether **FoldingFeature** should be considered, splitting the window into multiple physical areas that users can see as logically separate:

```
@OptIn(ExperimentalContracts::class)
fun separating(foldFeature: FoldingFeature?): Boolean {
    contract { returns(true) implies (foldFeature !=
        null) }
    return foldFeature?.state ==
        FoldingFeature.State.FLAT &&
            foldFeature.isSeparating
}
```

5. We will also create a sealed interface, **DevicePosture**. This is a Jetpack Compose UI component that allows you to detect a device's posture or orientation, such as whether the device is in portrait or landscape mode:

```
sealed interface DevicePosture {
    object NormalPosture : DevicePosture
    data class CityPosture(
        val hingePosition: Rect
    ) : DevicePosture
    data class Separating(
        val hingePosition: Rect,
        var orientation: FoldingFeature.Orientation
    ) : DevicePosture
}
```

6. In our **MainActivity**, we now need to ensure we calculate the window size:

```
val windowSize = calculateWindowSizeClass(activity = this)
```

7. Then, we will ensure we handle all sizes well by creating **postureStateFlow**, which will listen to our **DevicePosture**

and act when **`cityPosture`** is either folded, unfolded, or normal:

```
val postureStateFlow = WindowInfoTracker.getOrCreate(this).windowLayoutInfo(this)
. . .
when {
    cityPosture(foldingFeature) ->
      DevicePosture.CityPosture(foldingFeature.bounds)
    separating(foldingFeature) ->
      DevicePosture.Separating(foldingFeature.bounds,
        foldingFeature.orientation)
    else -> DevicePosture.NormalPosture
}
    }
. . .
    )
```

8. We now need to get set up with a foldable testing virtual device. You can repeat the steps from the first chapter on how to create a virtual device if you need a refresher; otherwise, you should go ahead and create a foldable device. The arrow in *Figure 9.2* shows how you will control the foldable screens.

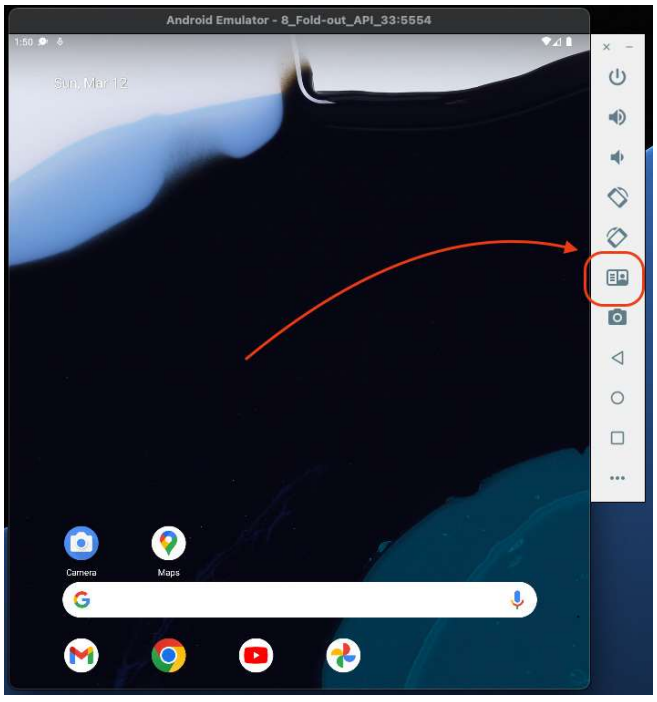

Figure 9.2 – The foldable controls

9. Then, finally, when you run the app, you will see that it changes based on folded and unfolded states, working well. *Figure 9.3* shows when the state is folded.

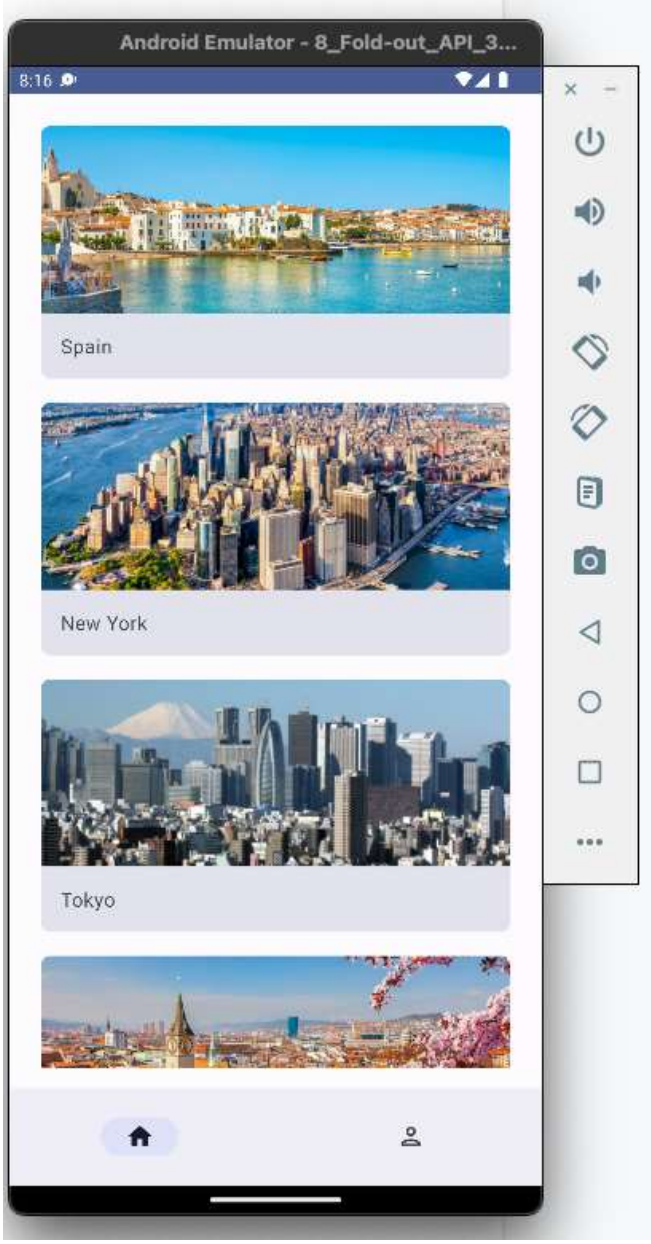

Figure 9.3 – The folded state

10. In *Figure 9.4*, you can see that we changed the bottom navigation and now have our navigation drawer set to the side for more straightforward navigation.

It should be acknowledged that this project is extensive, so we cannot cover all parts of the code. Make sure to utilize the Compose concepts learned in the previous chapter for this section.

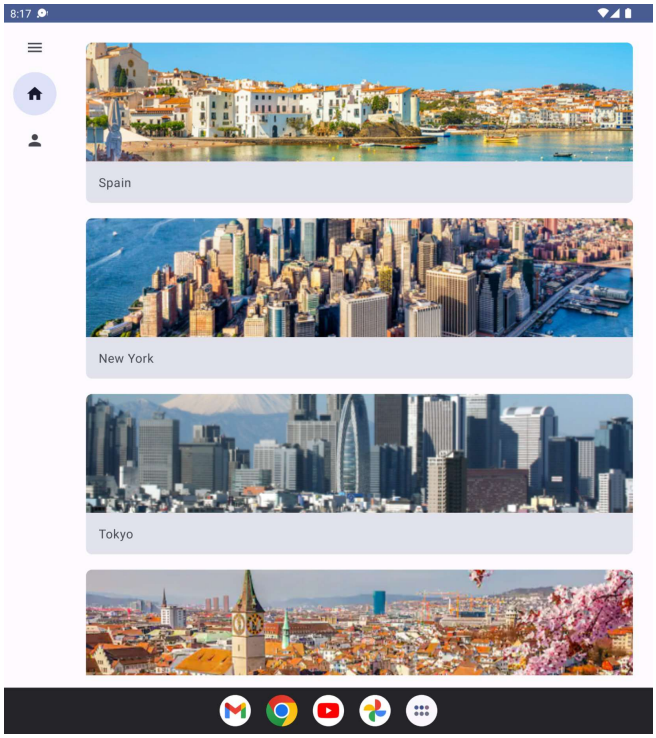

Figure 9.4 – The full-screen state (not folded)

Note that when you expand the navigation drawer, you can see all items, and you should be able to navigate easily.

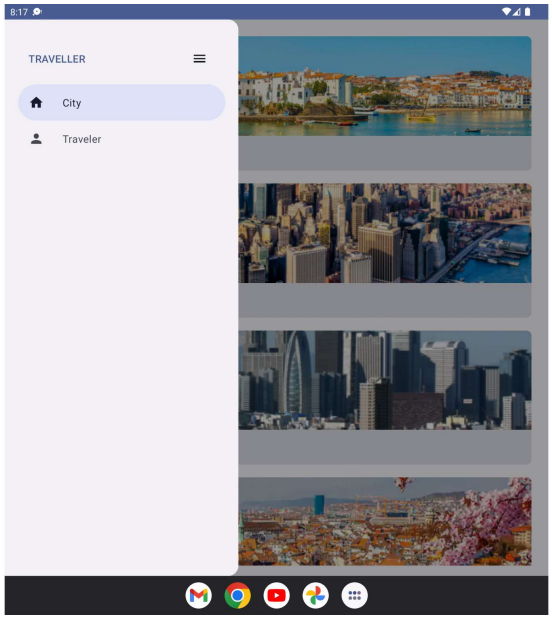

Figure 9.5 – The navigation drawer open

You can also see on the side panel a more descriptive view of your UI, which helps debug issues.

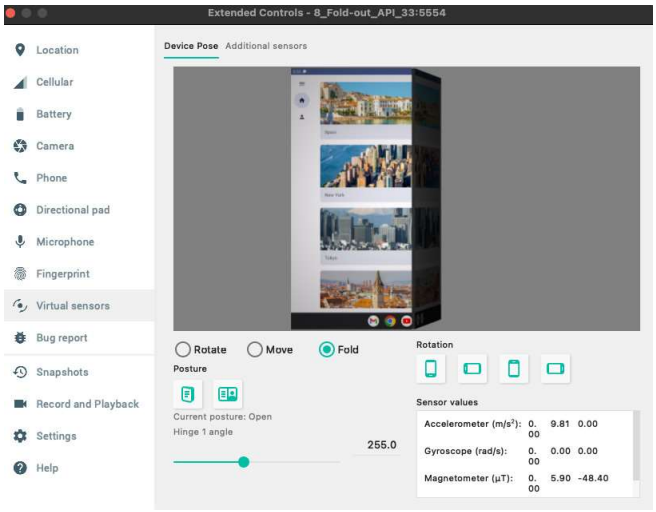

Figure 9.6 – The device pose

IMPORTANT NOTE

The code base for this project is vast and hence, cannot be covered in just one recipe; you can find the full code in the Technical requirements section.

How it works...

We covered the bottom navigation in *Chapter 4*, *Navigation in Modern Android Development*. In this chapter, however, we use it to showcase how your application can change as the screen changes if your application is installed on a foldable device, which is very important in Modern Android Development.

The navigation rail is used for medium-screen sizes, whereas the navigation drawer, just like in the old way of writing applications, is used as a side drawer and is suitable for large-screen devices. **`FoldFeature`** is a built-in Jetpack Compose UI component that allows you to create a folding animation effect when you click on it.

Here are the steps to use **`FoldFeature`** in your Android app. You can also customize **`FoldFeature`** by providing the necessary parameters:

- **`foldableState`**: This state controls the folding and unfolding of **`FoldFeature`**. You can create a **`FoldState`** instance using the **`rememberFoldState()`** function.
- **`foldedConten`**t: Content will be displayed when **`FoldFeature`** is folded.
- **`expandedContent`**: This is the content that will be displayed when **`FoldFeature`** is in its expanded state.
- **`foldingIcon`**: This is the icon that will be displayed to indicate the folding state of **`FoldFeature`**.

A foldable device has the ability to be in various states and postures. The Jetpack **`WindowManager`** library's **`WindowLayoutInfo`** class, which is what we use in our example, provides us with the following details. **`state`** helps describe the folded state the device is in. When the phone is fully opened, then the state is either **`FLAT`** or **`HALF_OPENED`**. We also get to play around with **`orientation`**, which is the orientation of the hinge.

The hinge can be either **`HORIZONTAL`** or **`VERTICAL`**. We have **`occlusionType`**, and this is the value that is **`FULL`** when the hinge hides part of the display. Otherwise, the value is **`NONE`**. Finally, we have **`isSeparating`**, which becomes valid when the hinge creates two logical displays.

Building adaptive layouts using ConstraintLayouts

Jetpack Compose, a declarative UI toolkit to build great UIs, is ideal to implement and design screen layouts that adjust automatically by themselves and render content well across different screen sizes.

This can be useful to consider when building your application, since the chance of it being installed in a foldable device is high. Furthermore, this can range from simple layout adjustments to filling up a foldable space that looks like a tablet.

Getting ready

You need to have read the previous chapters to follow along with this recipe.

How to do it...

For this recipe, we will build a separate composable function to show you how to use **`ConstraintLayout`** in the same project instead of creating a new one:

1. Let's go ahead and open **`Traveller`**. Add a new package and call it **`constraintllayoutexample`**. Inside the package, create a Kotlin file, called **`ConstraintLayoutExample`**, and then add the following dependency to the project:

```
implementation "Androidx.constraintlayout:constraintlayout-compose:1.x.x"
```

2. In our example, we will create a fun **`AndroidCommunity()`** and use **`ConstraintLayout`** to create **`title`**, **`aboutCommunity`**, and **`AndroidImage`** references:

```
@Composable
fun AndroidCommunity() {
    ConstraintLayout {
        val (title, aboutCommunity, AndroidImage) =
            createRefs()
 . . .
}
```

3. **`createRefs()`**, which means *create references*, simply creates a reference for each composable in our **`ConstrainLayout`**.
4. Now, let us go ahead and create our title text, **`aboutCommunity`**, and **`AndroidImage`**:

```
    Text(
        text = stringResource(id =
            R.string.Android_community),
        modifier = Modifier.constrainAs(title) {
            top.linkTo(parent.top)
            start.linkTo(parent.start)
            end.linkTo(parent.end)
        }
            .padding(top = 12.dp),
        style = TextStyle(
            color = Color.Blue,
            fontSize = 24.sp
        )
    )
```

5. Our title text has a modifier that has constraints defined, and if you have used XML before, you may notice that this works exactly how XML works. We provide constraints using the **`constrainAs()`** modifier, which, in our case, takes the references as a parameter and lets us specify its constraints in the body lambda. Hereafter, our constraints are specified using **`linkTo(...)`** or other methods, but in this case, we will use **`linkTo(parent.top)`**.

We can now connect the parts together using a similar style, in addition, ensure you check the *Technical requirements* section for the entire code:

```
Text(
    text = stringResource(id =
        R.string.about_community),
    modifier = Modifier.constrainAs(aboutCommunity) {
        top.linkTo(title.bottom)
        start.linkTo(parent.start)
        end.linkTo(parent.end)
```

```
            width = Dimension.fillToConstraints
        }
            .padding(top = 12.dp, start = 12.dp,
                end = 12.dp),
        style = TextStyle(
            fontSize = 18.sp
        )
    )
```

6. Then, we build the image:

```
    Image(
        painter = painterResource(id =
            R.drawable.Android),
        contentDescription = stringResource(id =
            R.string.Android_image),
        modifier = Modifier.constrainAs(AndroidImage) {
            top.linkTo(aboutCommunity.bottom,
                margin = 16.dp)
            centerHorizontallyTo(parent)
        }
    )
    . . .
```

7. Finally, to run this part of the code, you can run the **`@Preview`** section:

```
    @Preview(showBackground = true)
    @Composable
    fun ShowAndroidCommunity() {
        TravellerTheme() {
            AndroidCommunity()
        }
    }
```

8. When you run the application, it should render and adapt well to the screen sizes. For instance, if the state is full (which means not folded), data should be displayed on the entire screen (see *Figure 9.7*).

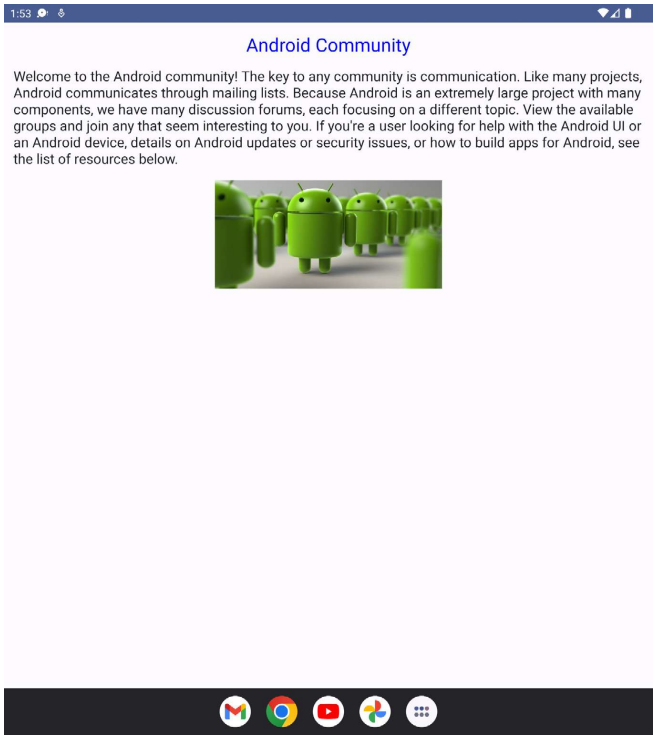

Figure 9.7 – The full screen of the not folded state

9. In *Figure 9.8*, you can see a version of the data when the screen is folded and how it adapts to the specified dimensions.

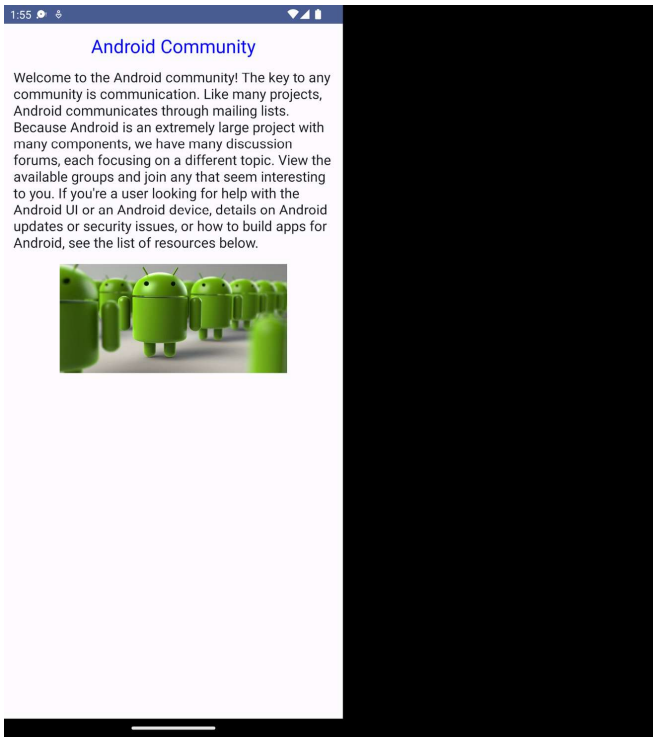

Figure 9.8 – The folded state

How it works…

We use modifiers to adjust the spacing between components and use dimension resources to define the margin between an image and `aboutCommunity`. Our layout will adjust based on the screen size to look good on both small and large screens.

We also use `ConstraintLayout`, which is a layout manager that allows us to create complex layouts with a flat view hierarchy. It also has built-in support for responsive layouts to create different layouts for different screen sizes and orientations.

The best use cases for `ConstraintLayout` include the following:

- When you want to avoid nesting multiple columns and rows; this can include when you want to position your elements on screen for easier code readability
- Utilizing it when you need to use guidelines, chains, or barriers in your positioning

We mentioned modifiers in previous chapters, which are like attributes in XML layouts. They allow us to apply different styles and behaviors to the components in our layout. You can use modifiers to change your component's size, position, and other properties, based on the screen size.

In our example, we use dynamic padding and margins, and you can use them to adjust

the spacing between components based on the screen size. For example, you can use a modifier to add more padding to a component on larger screens.

This allows you to create responsive layouts that adjust based on the screen size.

Handling large-screen configuration changes and continuity

Android devices undergo various configuration changes during their operation. Some of the most notable, or standard, ones include the following:

- **Screen orientation change**: This occurs when a user rotates a device's screen, triggering a configuration change. This is when the device switches from portrait to landscape mode or vice versa.
- **Screen size change**: This is when a user changes the screen size of a device – for example, by plugging or unplugging an external display, triggering a configuration change.
- **Language or locale change**: This is when a user changes the language or locale of a device, triggering a configuration change. This can affect the formatting of text and dates, among other things.
- **Theme change**: This is when a user changes a device's theme, triggering a

configuration change. This can affect the appearance of the UI.

- **Keyboard availability change**: This is when a user attaches or detaches a keyboard from a device, triggering a configuration change. This can affect the layout of the UI, and so on.

In this recipe, we will look at leveraging this knowledge to better handle screen size changes when dealing with foldable devices.

Getting ready

In the first recipe, *Building adaptive layouts in Modern Android Development*, we discussed different state configurations and how to handle them better. In this recipe, we will learn how to use the already provided **`rememberFoldableState`** function in Jetpack Compose to handle screen changes in foldable devices.

How to do it...

Let's use the already created **`Traveller`** project for this example; you will not need to create a new project:

1. To be able to use **`rememberFoldableState`**, we will need to import it into our project:

```
import Androidx.window.layout.FoldableState
```

2. Then, we will create a new **`val/ property foldableState`** and initialize it with our **`rememberFoldableState`**:

```
val foldState = rememberFoldableState()
```

3. Using the **foldState** object, we can get information about foldable devices, make our application respond to the correct state, and display data as needed. The three states available are **STATE_FLAT**, **STATE_HALF_OPENED**, and **STATE_CLOSED**:

```
when (foldState.state) {
    FoldableState.STATE_FLAT -> {
        // Our Device is flat (unfolded)do something
    }
    FoldableState.STATE_HALF_OPENED -> {
        //Our Device is partially folded. Do something
    }
    FoldableState.STATE_CLOSED -> {
        //Our Device is fully folded do something
    }
}
```

4. We can then use this information to adjust our UI accordingly, such as showing or hiding certain elements based on the foldable state or specified position. Also, we can create two different layouts for when the device is folded and when it is unfolded:

```
val isFolded = foldState.state == FoldableState.STATE_CLOSED
if (isFolded) {
    // Create our layout for when the device is folded
} else {
    // Create our layout for when the device is
       unfolded
}
```

And that's it; this will help solve the foldable state if you have a complex UI system that needs better handling.

How it works...

Handling significant screen configuration changes, especially with foldable devices, can be challenging in Android Jetpack Compose. Here are some tips that can help you use the Configuration API. It allows you to get information about a device's screen configuration, such as screen size, orientation, and foldable state. You can use this information to adjust your UI accordingly.

Compose's layout system makes it easy to create responsive UIs that can adapt to different screen sizes and aspect ratios. Use flexible layouts such as columns and rows to create a UI that can scale up or down as needed.

The `rememberFoldableState` function lets you get information about a device's foldable state and adjust your UI accordingly. For example, you can use this function to create two different layouts, one for when the device is folded and one for when it is unfolded.

Testing your app with different screen configurations is also essential to ensure that it works properly. You can use the Android emulator or physical devices to test your app.

Understanding activity embedding

In Jetpack Compose, activity embedding refers to the process of including a composable function within the context of an activity. This allows you to create custom views that can integrate seamlessly with existing Android activities.

To embed a composable function within an activity, you can use the **`setContent`** method of the activity. This method accepts a composable function as a parameter, which can be used to define the activity's layout.

Getting ready

You need to have completed the previous recipes to follow along.

How to do it...

Let's look at an example of embedding a composable function in an activity:

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MyCustomView()
        }
    }
}
@Composable
fun MyCustomView() {
    Text(text = "Hello, Android Community!")
}
```

In this example, the **`setContent`** method embeds the **`MyCustomView`** composable function within **`MainActivity`**. When the activity is

created, the `MyCustomView` function will be called to generate the activity's layout.

How it works...

The `MyCustomView` function is defined as a composable function using the `@Composable` annotation. This allows the function to be called multiple times without causing any side effects. In this case, the function simply displays a `Text` composable with the text `Hello, Android Community!`.

By embedding composable functions within activities, you can create custom views that can be easily integrated into your Android app. This can be especially useful to create reusable components or customize the layout of existing activities.

Material Theming in Compose

Material Theming in Compose is a design system introduced by Google that provides guidelines and principles to design user interfaces. Material Theming helps designers create interfaces that are consistent, easy to use, and visually appealing. Some key features of Material Theming in Jetpack Compose include the following:

- **Color palettes**: A set of predefined color palettes that can be used to create consistent and visually appealing interfaces. Jetpack Compose provides a

`MaterialTheme` composable that allows you to apply a color palette to your entire app.

- **Typography**: A set of typography styles that can create a consistent and easy-to-read interface. Jetpack Compose provides a `Typography` composable that allows you to apply a typography style to your text.
- **Shapes**: A set of shapes that can create consistent and visually appealing components. Jetpack Compose provides a `Shape` composable that allows you to apply a shape to your components.
- **Icons**: A set of icons that can be used to create consistent and recognizable interfaces. Jetpack Compose provides an `Icon` composable that allows you to use Material icons in your app.

By using Material Theming in Jetpack Compose, you can create interfaces that are consistent, easy to use, and visually appealing. Material Theming in Jetpack Compose helps you focus on designing your app's functionality, while the design system takes care of the visual details.

Getting ready

To be able to follow along, you need to have worked on previous recipes.

How to do it...

Many applications still do not use `Material 3`, but if you build a new application from scratch, it is highly recommended you go

with **`Material 3`**. One thing to note is when you create a project, **`Material 3`** does not come pre-installed; this means you need to go ahead and update the Material libraries yourself to **`Material 3`**.

Let's see an example of implementing **`Material 3`** theming in Jetpack Compose for your Android applications:

1. You will need to add the required **`Material 3`** dependencies to your app's **`build.gradle`** file:

```
implementation 'Androidx.compose.material3:material3:1.0.0-alpha14'
```

2. Then, you will need to declare your app theme:

```
@Composable
fun MyAppMaterialTheme(content: @Composable () -> Unit) {
    MaterialTheme(
        colorScheme = /**/,
        typography = /**/,
        shapes = /**/,
        content = content
    )
}
```

3. Finally, you can use your theme in the entire application:

```
@Composable
fun MyApp() {
    MyAppMaterialTheme {}
}
```

In this example, we've used **`Material 3`** colors, typography, and shapes to create a consistent and visually appealing interface. We've also used **`Material 3`** icons to en-

hance the user experience. Finally, we've wrapped our app's content in the `MyAppMaterialTheme` composable to apply the `Material 3` theme.

How it works...

Here's how `Material 3` works in Jetpack Compose. `Material 3` introduces new and updated components, such as `AppBar`, `BottomNavigation`, and `TabBar`, which can be used in Jetpack Compose using the `Androidx.compose.material3` package. These components have updated design and functionality, and they follow the `Material 3` guidelines.

Material 3 also introduces a new theming system that allows for more customization and flexibility – that is, in Jetpack Compose, `Material 3` theming can be applied using the `MaterialTheme3` composable. This composable allows you to customize the color scheme, typography, and shapes of your app, and it also provides new options to customize the elevation and shadows of your components.

The icons are now modern and easily accessible, which is a big plus for us developers. Finally, Material 3 introduces a new typography system that provides updated styles and guidelines for typography in your app. In Jetpack Compose, `Material 3` typography can be applied using the `Material3Typography` object, which provides several predefined styles for your text.

By using `Material 3` in Jetpack Compose, you can create modern and visually appealing interfaces that follow the latest design guidelines. Also note that `Material 3` components, theming, icons, and typography can all be used together to create a cohesive and consistent design system for your app.

See also...

There is much to cover in Material Design, and trying to cover all components in a single recipe will not do it justice. To learn more about the components and how to ensure your application follows the Material Design guideline, read more here: **https://material.io/components**.

Testing your applications on a foldable device

Testing your apps on foldable devices is essential to ensure they work correctly and provide an excellent user experience. In this recipe, we will look at some tips to test your apps on foldable devices in Jetpack Compose.

Getting ready

You will need to have done the previous recipes. You can access the entire code in the *Technical requirements* section.

How to do it...

Here are some tips to test your applications on foldable devices:

- **Use an emulator**: You can use the Android emulator to test your app on foldable devices without buying a physical device. The emulator provides a range of foldable device configurations that you can use to test your app.
- **Use real devices**: Testing your app on an actual foldable device can provide a more accurate representation of how your app will work on these devices. If you have access to a foldable device, it's highly recommended to test your app on it.
- **Test different screen modes**: Foldable devices come in different screen modes, such as single-screen, dual-screen, and extended screens. It's essential to test your app on different screen modes to ensure it works correctly in all modes.
- **Test with different screen sizes**: Foldable devices come in different sizes, so it's crucial to test your app on different screen sizes to ensure it works well on all devices.
- **Test app transition**: Testing your app's transition between different screen modes can help you identify any issues with an app's layout or behavior. Make sure to test all the transition modes, such as fold, unfold, and hinge.
- **Use automated testing**: Automated testing can help you test your app on differ-

ent screen sizes, modes, and orientations more efficiently. You can use tools such as Espresso or UI Automator to write automated tests for your app.

How it works...

Overall, testing your app on foldable devices requires careful consideration of a device's unique features and abilities. By following these tips, you can ensure that your app is optimized for foldable devices and provides an excellent user experience.