Chapter 9: Exploring Interoperability APIs

The aim of this book is to show you how to develop beautiful, fast, and maintainable Jetpack Compose apps. The previous chapters helped you get familiar with core techniques and principles, as well as important interfaces, classes, packages, and—of course—composable functions. The remaining chapters cover topics beyond a successful adoption of Android's new declarative user interface toolkit.

In this chapter, we are going to look at `AndroidView()`, `AndroidViewBinding()`, and `ComposeView` as the interoperability **application programming interfaces (APIs)** of Jetpack Compose. The main sections are listed here:

- Showing Views in a Compose app
- Sharing data between Views and composable functions
- Embedding composables in View hierarchies

We start by looking at how to show a traditional View hierarchy in a Compose app. Imagine you have written a custom component (which under the hood consists of several UI elements), such as an image picker, a color chooser, or a camera preview. Instead of rewriting your component with Jetpack Compose, you can save your investment by simply reusing it. A lot of third-party libraries are still written in Views, so I will show you how to use them in Compose apps.

Once you have embedded a View in a Compose app, you need to share data between the View and your composable functions. The *Sharing data between Views and composable functions* section explains how to do this with ViewModels.

Often, you may not want to rewrite an app from scratch but migrate it to Jetpack Compose gradually, replacing View hierarchies with composable functions step by step. The final main section, *Embedding composables in View hierarchies*, discusses how to include a Compose hierarchy in existing View-based apps.

Technical requirements

This chapter is based on the **`ZxingDemo`** and **`InteropDemo`** samples. Please refer to the *Technical requirements* section of *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_09.

Showing Views in a Compose app

Imagine you have written a View-based custom component for one of your previous apps—for example, an image picker, a color chooser, or a camera preview—or you would like to include a third-party library such as *Zebra Crossing* (*ZXing*) to scan **Quick Response** (**QR**) codes and barcodes. To incorporate them into a Compose app, you need to add the View (or the root of a View hierarchy) to your composable functions.

Let's see how this works.

Adding custom components to a Compose app

The **`ZxingDemo`** sample, shown in the following screenshot, uses the *ZXing Android Embedded* barcode scanner library for Android, which is based on the ZXing decoder. It is released under the terms of the Apache

License 2.0 and is hosted on GitHub (https://github.com/journeyapps/zxing-android-embedded):

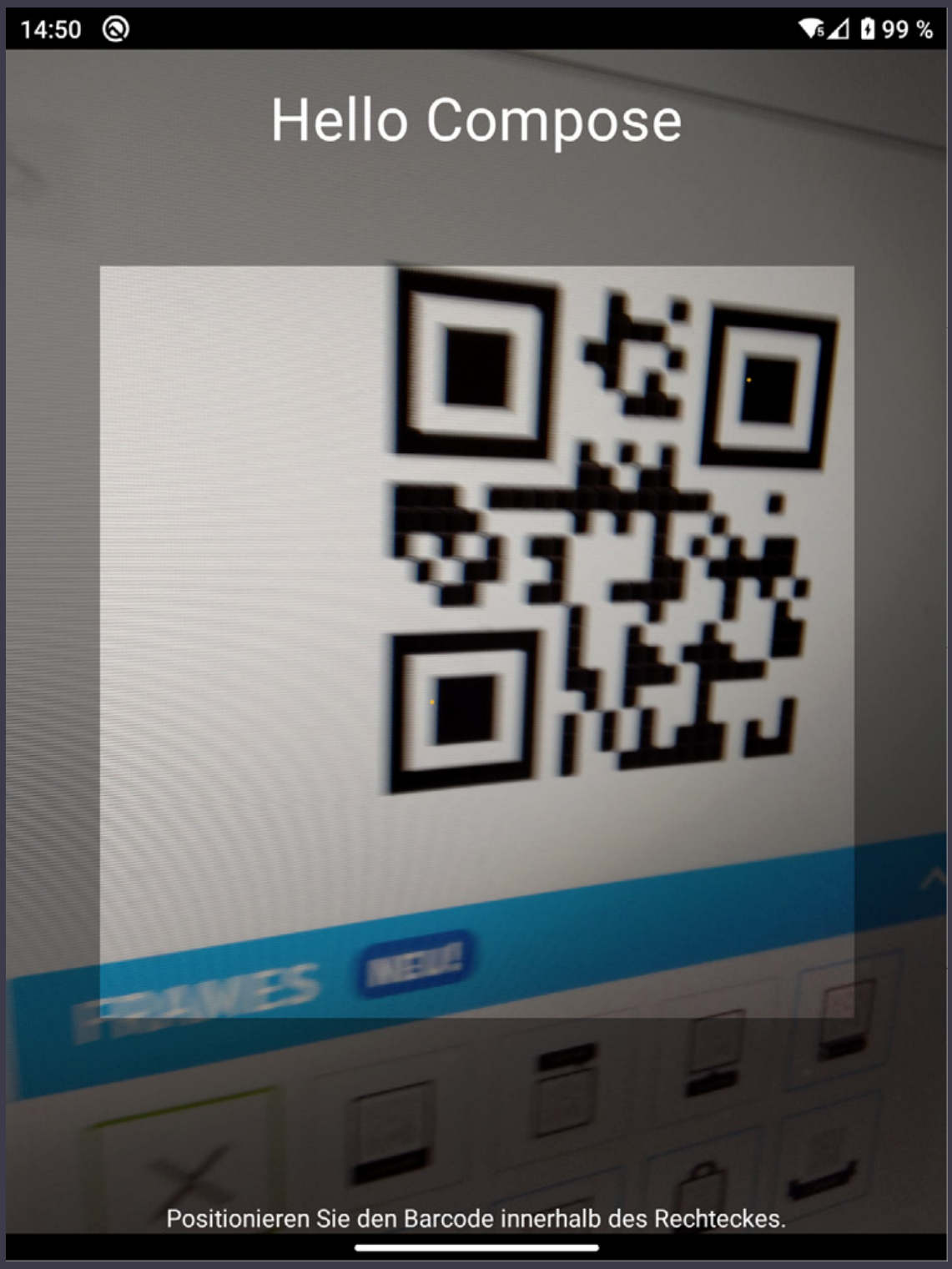

Figure 9.1 – The ZxingDemo sample

My example continuously scans for barcodes and QR codes. The decorated barcode view is provided by the library. If the scanner engine provides a result, the corresponding text is shown as an overlay using `Text()`.

To use *ZXing Android Embedded*, you need to add an implementation dependency to your module-level **build.gradle** file, as follows:

```
implementation 'com.journeyapps:zxing-android-
embedded:4.3.0'
```

The scanner accesses the camera and (optionally) the device vibrator. The app must request at least **android.permission.WAKE_LOCK** and **android.permission.CAMERA** permissions in the manifest, and the **android.permission.CAMERA** permission during runtime. My implementation is based on **ActivityResultContracts.RequestPermission**, which replaces the traditional approach overriding **onRequestPermissionsResult()**. Also, depending on the lifecycle of the activity, the scanner must be paused and resumed. For the sake of simplicity, I use a **lateinit** variable named **barcodeView** and invoke **barcodeView.pause()** and **barcodeView.resume()** when needed. Please refer to the source code of the project for details. Next, I will show you how to initialize the scanner library. This involves inflating a layout file (named **layout.xml**), as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<com.journeyapps.barcodescanner.DecoratedBarcodeView
  xmlns:android="http://schemas.android.com/apk/res/a
ndroid"
  android:id="@+id/barcode_scanner"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:layout_alignParentTop="true" />
```

The layout consists of only one element, **DecoratedBarcodeView**. It is configured to fill all available space. The following code snippet is part of **onCreate()**. Please remember that **barcodeView** is accessed in some lifecycle functions such as **onPause()**, and therefore is a **lateinit** property:

```
val root = layoutInflater.inflate(R.layout.layout,
null)
barcodeView = root.findViewById(R.id.barcode_scanner)
val formats = listOf(BarcodeFormat.QR_CODE,
```

```
    BarcodeFormat.CODE_39)
barcodeView.barcodeView.decoderFactory =
    DefaultDecoderFactory(formats)
barcodeView.initializeFromIntent(intent)
val callback = object : BarcodeCallback {
  override fun barcodeResult(result: BarcodeResult) {
    if (result.text == null || result.text ==
text.value) {
      return
    }
    text.value = result.text
  }
}
barcodeView.decodeContinuous(callback)
```

First, **`layout.xml`** is inflated and assigned to **`root`**. Then, **`barcodeView`** is initialized (**`initializeFromIntent()`**) and configured (by setting a decoder factory). Finally, the continuous scanning process is started using **`decodeContinuous()`**. The **`callback`** lambda expression is invoked every time a new scan result is available. The **`text`** variable is defined like this:

```
private val text = MutableLiveData("")
```

I am using **`MutableLiveData`**, because it can easily be observed as state. Before I show you how to access it inside a composable function, let's briefly recap, as follows:

- We have set up and activated the scanner library.
- When it detects a barcode or a QR code, it updates the value of a **`MutableLiveData`** instance.
- We defined and initialized two **`View`** instances—**`root`** and **`barcodeView`**.

Next, I show you how to access the state obtained from the ViewModel inside a composable, as follows:

```
setContent {
  val state = text.observeAsState()
```

```
    state.value?.let {
      ZxingDemo(root, it)
    }
  }
```

The value of the state and **`root`** are passed to the **`ZxingDemo()`** composable. We display **`value`** using **`Text()`**. The **`root`** parameter is used to include the View hierarchy in the Compose UI. The code is illustrated in the following snippet:

```
@Composable
fun ZxingDemo(root: View, value: String) {
  Box(
    modifier = Modifier.fillMaxSize(),
    contentAlignment = Alignment.TopCenter
  ) {
    AndroidView(modifier = Modifier.fillMaxSize(),
      factory = {
        root
      })
    if (value.isNotBlank()) {
      Text(
        modifier = Modifier.padding(16.dp),
        text = value,
        color = Color.White,
        style = MaterialTheme.typography.h4
      )
    }
  }
}
```

The UI consists of a **`Box()`** composable with two children, **`AndroidView()`** and **`Text()`**. **`AndroidView()`** receives a **`factory`** block, which just returns **`root`** (the View hierarchy containing the scanner viewfinder). The **`Text()`** composable shows the last scan result.

The **`factory`** block is called exactly once, to obtain the View to be composed. It will always be invoked on the UI thread, so you can set View properties as needed. In my example, this is not needed, as all initialization has already been done in **`onCreate()`**. Configuring the barcode scanner should not be done in a composable, because preparing the camera and preview is potentially time-consuming. Also, parts of the component tree are accessed on the activity level, therefore references to children (**`barcodeView`**) are needed anyway.

In this section, I have shown you how to include a View hierarchy in your Compose app using **`AndroidView()`**. This composable function is one of the important pieces of the Jetpack Compose interoperability APIs. We used **`layoutInflater.inflate()`** to inflate the component tree and **`findViewById()`** to access one of its children. Modern View-based apps try to avoid **`findViewById()`** and use *View Binding* instead. In the next section, you will learn how to combine View Binding and composable functions.

Inflating View hierarchies with AndroidViewBinding()

Traditionally, activities held references to Views in **`lateinit`** properties, if the corresponding components needed to be modified in different functions. The *Inflating layout files* section of *Chapter 2*, *Understanding the Declarative Paradigm*, discussed some of the issues with this approach and introduced View Binding as a solution. It was adopted by many apps. Therefore, if you want to migrate an existing app to Jetpack Compose, you likely need to combine View Binding and composable functions. This section explains how to achieve that.

The following screenshot shows the **`InteropDemo`** sample:

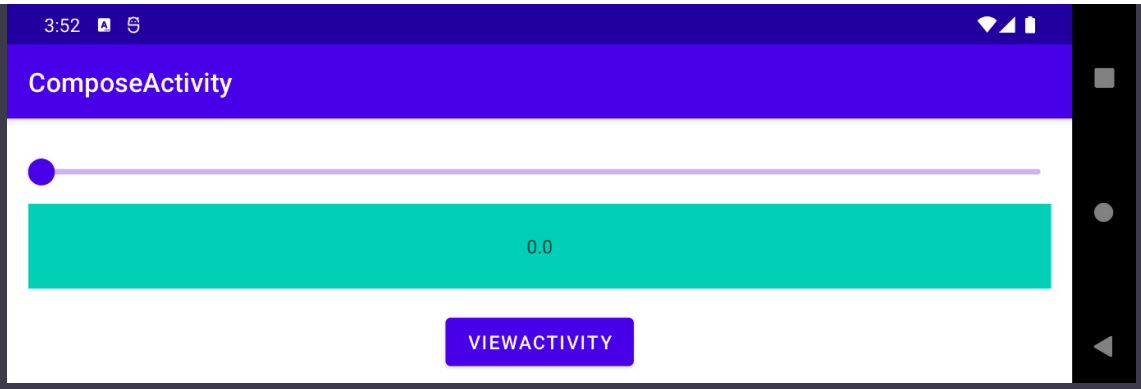

Figure 9.2 – The InteropDemo sample

The **`InteropDemo`** sample consists of two activities. One (**`ViewActivity`**) integrates a composable function in a **`View`** hierarchy. We will turn to this in the *Embedding composables in View hierarchies* section. The second one, **`ComposeActivity`**, does the opposite: it inflates a **`View`** hierarchy using View Binding and shows the component tree inside a **`Column()`** composable. Let's take a look here:

```
class ComposeActivity : ComponentActivity() {
  override fun onCreate(savedInstanceState: Bundle?)
{
    super.onCreate(savedInstanceState)
    val viewModel: MyViewModel by viewModels()
    …
    setContent {
      ViewIntegrationDemo(viewModel) {
        val i = Intent(
          this,
          ViewActivity::class.java
        )
        i.putExtra(KEY, viewModel.sliderValue.value)
        startActivity(i)
      }
    }
  }
}
```

The root composable is called **`ViewIntegrationDemo()`**. It receives a ViewModel and a lambda expression. The ViewModel is used to share data between the Compose and the **`View`** hierarchies, which I will discuss in the *Sharing data between Views and composable functions* section. The lambda expression starts **`ViewActivity`** and passes a value from the ViewModel (**`sliderValue`**). The code is illustrated in the following snippet:

```
@Composable
fun ViewIntegrationDemo(viewModel: MyViewModel,
                        onClick: () -> Unit) {
  val sliderValueState =
    viewModel.sliderValue.observeAsState()
  Scaffold( ... ) {
    Column( ... ) {
      Slider( … )
      AndroidViewBinding(
        modifier = Modifier.fillMaxWidth(),
        factory = CustomBinding::inflate
      ) {
        // Here Views will be updated
      }
    }
  }
}
```

`Scaffold()` is an important integrational composable function. It structures a Compose screen. Besides top and bottom bars, it contains some content—in this case, a **`Column()`** composable with two children, **`Slider()`** and **`AndroidViewBinding()`**. The slider gets its current value from a ViewModel and propagates changes back to it. You will learn more about that in the *Revisiting ViewModels* section.

`AndroidViewBinding()` is similar to **`AndroidView()`**. A **`factory`** block creates a View hierarchy to be composed. **`CustomBinding::inflate`** inflates the layout from the **`custom.xml`** file represented by **`CustomBinding`** and returns an instance of this type. The class is created and updated during builds. It

provides constants that reflect the contents of a layout file named **`custom.xml`**. Here is an abridged version:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
  xmlns:android="http://schemas.android.com/apk/res/a
ndroid"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <com.google.android.material.textview.MaterialTextV
iew
    android:id="@+id/textView"
    ... />
  <com.google.android.material.button.MaterialButton
    android:id="@+id/button"
    …
    android:text="@string/view_activity"
    …
    app:layout_constraintTop_toBottomOf="@id/textView
" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

This **`ConstraintLayout`** has two children, a **`MaterialTextView`** and a **`MaterialButton`**. A button click invokes the lambda expression passed to **`ViewIntegrationDemo()`**. The text fields receive the current slider value. This is done in the **`update`** block. The following code belongs below **`// Here Views will be updated`** inside **`ViewIntegrationDemo()`**:

```
textView.text = sliderValueState.value.toString()
button.setOnClickListener {
    onClick()
}
```

You may be wondering where **`textView`** and **`button`** are defined, and why they can be accessed immediately. The **`update`** block is invoked right after the layout is inflated. It is an extension function of the type whose in-

stance is returned by **`inflate`**—in my example, **`CustomBinding`**. Because the **identifiers (IDs)** of the button and the text field in **`custom.xml`** are **`button`** and **`textView`**, there are corresponding variables in **`CustomBinding`**.

The **`update`** block is also invoked when a value being used by it (**`sliderValueState.value`**) changes. In the next section, we look at when and where such changes are triggered.

Sharing data between Views and composable functions

State is app data that may change over time. Recomposition occurs when state being used by a composable changes. To achieve something similar in the traditional View world, we need to store data in a way that changes to it can be observed. There are many implementations of the *Observable* pattern. The Android Architecture Components (and subsequent Jetpack versions) include **`LiveData`** and **`MutableLiveData`**. Both are frequently used inside ViewModels to store state outside activities.

Revisiting ViewModels

I introduced you to ViewModels in the *Surviving configuration changes* section of *Chapter 5*, *Managing the State of Your Composable Functions*, and the *Persisting and retrieving state* section of *Chapter 7*, *Tips, Tricks, and Best Practices*. Before we look at how to use ViewModels to synchronize data between Views and composable functions, let's briefly recap on key techniques, as follows:

- To create or get an instance of a ViewModel, use the top-level **`viewModels()`** function, which belongs to the **`androidx.activity`** package.
- To observe **`LiveData`** instances as compose state, invoke the **`observeAsState()`** extension function on a ViewModel property inside a composable function.

- To observe **`LiveData`** instances outside of composable functions, invoke **`observe()`**. This function belongs to **`androidx.lifecycle.LiveData`**.
- To change a ViewModel property, invoke the corresponding setter.
  IMPORTANT NOTE
  Please make sure to add implementation dependencies of **`androidx.compose.runtime:runtime-livedata`**, **`androidx.lifecycle:lifecycle-runtime-ktx`**, *and* **`androidx.lifecycle:lifecycle-viewmodel-compose`** *in the module-level* **`build.gradle`** *file as needed.*

Now that we have refamiliarized ourselves with key techniques related to ViewModels, let's look at how the synchronization between Views and composable functions works. *Synchronization* means that a composable function and code related to a View observe the same ViewModel property and may trigger changes on that property. Triggering changes is usually done by invoking a setter. For a **`Slider()`** composable, it may look like this:

```
Slider(
  modifier = Modifier.fillMaxWidth(),
  onValueChange = {
    viewModel.setSliderValue(it)
  },
  value = sliderValueState.value ?: 0F
)
```

This example also shows the readout inside a composable (**`sliderValueState.value`**). Here is how **`sliderValueState`** has been defined:

```
val sliderValueState =
viewModel.sliderValue.observeAsState()
```

Next, let's look at traditional (non-Compose) code using View Binding. The following examples are part of **`ViewActivity`**, which also belongs to the **`InteropDemo`** sample.

Combining View Binding and ViewModels

Activities taking advantage of View Binding usually have a **lateinit** property named **binding**, as illustrated in the following code snippet:

```
  binding = LayoutBinding.inflate(layoutInflater)
```

LayoutBinding.inflate() returns an instance of **LayoutBinding**. **Binding.root** represents the root of the component tree. It is passed to **setContentView()**. Here is an abridged version of the corresponding layout file (**layout.xml**):

```
  <?xml version="1.0" encoding="utf-8"?>
  <androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/a
  ndroid"
    …
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <com.google.android.material.slider.Slider
      android:id="@+id/slider"
      ... />
    <androidx.compose.ui.platform.ComposeView
      android:id="@+id/compose_view"
      ...
      app:layout_constraintTop_toBottomOf="@id/slider"
  />
  </androidx.constraintlayout.widget.ConstraintLayout>
```

ConstraintLayout contains a **com.google.android.material.slider.Slider** and a **ComposeView** (which is discussed in detail in the following section). The ID of the slider is **slider**, so **LayoutBinding** contains an equally named variable. We can therefore link the slider to the ViewModel, like this:

```
  viewModel.sliderValue.observe(this) {
    binding.slider.value = it
  }
```

The block passed to **`observe()`** is invoked when the value stored in **`sliderValue`** changes. By updating **`binding.slider.value`**, we change the position of the slider handle, which means we update the slider. The code is illustrated here:

```
binding.slider.addOnChangeListener { _, value, _ ->
  viewModel.setSliderValue(value) }
```

The block passed to **`addOnChangeListener()`** is invoked when the user drags the slider handle. By invoking **`setSliderValue()`** we update the ViewModel, which in turn triggers updates on observers—for example, our composable functions.

In this section, I familiarized you with the steps needed to tie composable functions and traditional Views to a ViewModel property. When the property is changed, all observers are called, which leads to the update of both the composable and View. In the following section, we continue our look at the **`InteropDemo`** sample. This time, I will show you how to embed composables in a View hierarchy. This is important if an existing app is to be migrated to Jetpack Compose step by step.

Embedding composables in View hierarchies

As you have seen, integrating Views in composable functions is simple and straightforward using **`AndroidView()`** and **`AndroidViewBinding()`**. But what about the other way round? Often, you may not want to rewrite an existing (View-based) app from scratch but migrate it to Jetpack Compose gradually, replacing View hierarchies with composable functions step by step. Depending on the complexity of the activity, it may make sense to start with small composables that reflect portions of the UI and incorporate them into the remaining layout.

`Androidx.compose.ui.platform.ComposeView` makes composables available inside classic layouts. The class extends **`AbstractComposeView`**, which has

ViewGroup as its parent. Once the layout that includes the ComposeView has been inflated, here is how you configure it:

```
binding.composeView.run {
  setViewCompositionStrategy(
      ViewCompositionStrategy.DisposeOnDetachedFromWi
ndow)
  setContent {
    val sliderValue =
            viewModel.sliderValue.observeAsState()
    sliderValue.value?.let {
      ComposeDemo(it) {
        val I = Intent(
          context,
          ComposeActivity::class.java
        )
        i.putExtra(KEY, it)
        startActivity(i)
      }
    }
  }
}
```

setContent() sets the content for this view. An initial composition will occur when the view is attached to a window, or when **createComposition()** is called. While **setContent()** is defined in **ComposeView**, **createComposition()** belongs to **AbstractComposeView**. It performs the initial composition for this view. Typically, you do not need to invoke this function directly.

setViewCompositionStrategy() configures how to manage the disposal of the View's internal composition.
ViewCompositionStrategy.DisposeOnDetached FromWindow (the default) means that the composition is disposed whenever the view becomes detached from a window. This is preferred for simple scenarios, as in my example. If your view is shown inside a fragment or a component with a known **LifecycleOwner**, you should use

DisposeOnViewTreeLifecycleDestroyed or **DisposeOnLifecycleDestroyed** instead. These, however, are topics beyond the scope of this book. The following line creates state based on the **sliderValue** property of the ViewModel and passes the value to **ComposeDemo()**:

```
val sliderValue =
viewModel.sliderValue.observeAsState()
```

This composable also receives a block that launches **ComposeActivity** and passes the current slider value to it, as illustrated in the following code snippet:

```
@Composable
fun ComposeDemo(value: Float, onClick: () -> Unit) {
  Column(
    modifier = Modifier
      .fillMaxSize(),
    horizontalAlignment =
Alignment.CenterHorizontally
  ) {
    Box(
      modifier = Modifier
        .fillMaxWidth()
        .background(MaterialTheme.colors.secondary)
        .height(64.dp),
      contentAlignment = Alignment.Center
    ) {
      Text(
        text = value.toString()
      )
    }
    Button(
      onClick = onClick,
      modifier = Modifier.padding(top = 16.dp)
    ) {
      Text(text = stringResource(id =
```

```
                R.string.compose_activity))
        }
    }
}
```

ComposeDemo(), as illustrated in the following screenshot, puts a **Box()** (which contains a **Text()**) and a **Button()**inside a **Column()**, in order to resemble **ViewActivity**. Wrapping **Text()** inside the **Box()**is necessary to vertically center the text inside an area with a particular height. A click on the button invokes the **onClick** lambda expression. **Text()** just shows the **value** parameter:

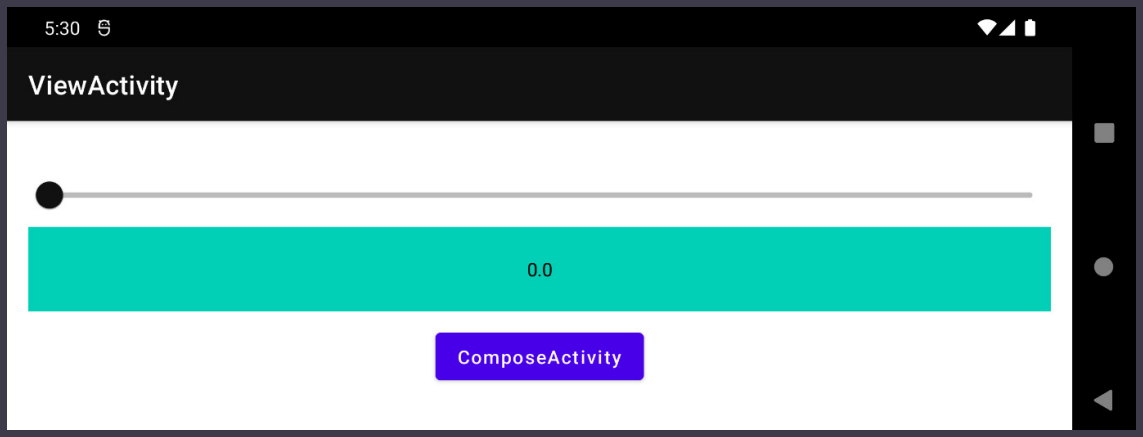

Figure 9.3 – The InteropDemo sample showing ViewActivity

Before closing out this chapter, let me recap on important steps you need to take to include a Compose hierarchy in a layout, as follows:

- Add **androidx.compose.ui.platform.ComposeView** to the layout.
- Decide on a **ViewCompositionStrategy**, depending on where the layout is shown (activity, fragment, ...).
- Set the content using **setContent {}**.
- Obtain a reference to the **ViewModel** by invoking **viewModels()**.
- Register listeners to relevant Views and update the **ViewModel** upon changes.
- Inside composable functions, create state by invoking **observeAsState()** on ViewModel properties as needed.
- Inside composables, update the ViewModel by invoking corresponding setters.

Jetpack Compose interoperability APIs allow for seamless two-way integration of composable functions and `View` hierarchies. They help you use libraries that rely on Views and ease the transition to Compose by making a gradual, fine-grained migration possible.

Summary

In this chapter, we looked at the interoperability APIs of Jetpack Compose, which allow you to mix composable functions and traditional Views. We started by incorporating a traditional View hierarchy from a third-party library in a Compose app, using `AndroidView()`. As recent apps favor View Binding over the direct use of `findViewById()`, I also showed you how to embed layouts in a composable with View Binding and `AndroidViewBinding()`. Once you have embedded a `View` in a Compose UI, you need to share data between the two worlds. The *Sharing data between Views and composable functions* section explained how to achieve this with ViewModels. The final main section, *Embedding composables in View hierarchies*, discussed how to include a Compose UI in existing apps using `ComposeView`.

Chapter 10, Testing and Debugging Compose Apps, focuses on testing your Compose apps. You will learn how to use `ComposeTestRule` and `AndroidComposeTestRule`. Also, I will introduce you to the *semantics tree*.