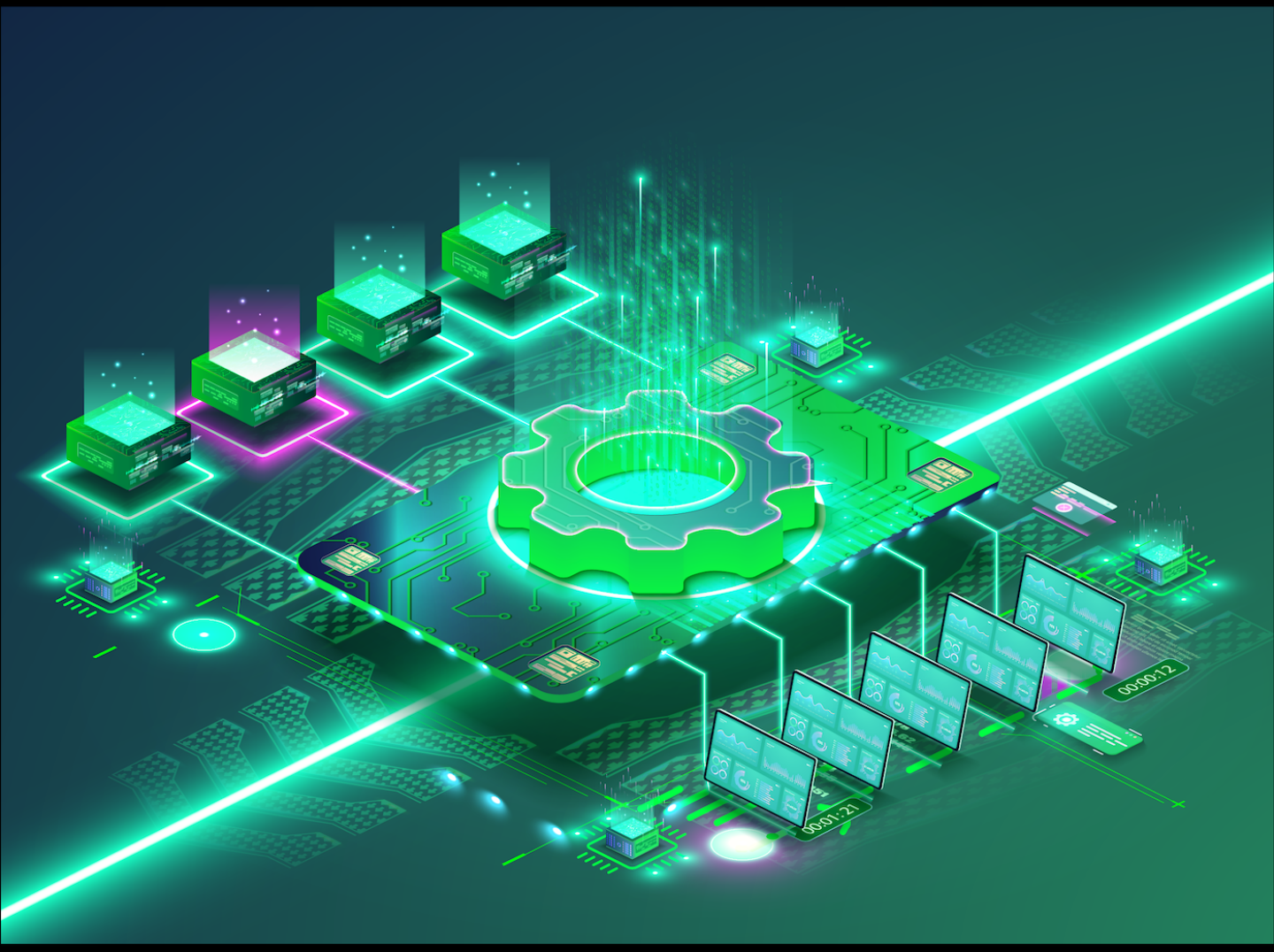
Jetpack Compose 1.6 Essentials



Neil Smyth

Payload publishing

Jetpack Compose 1.6 Essentials

Jetpack Compose 1.6 Essentials

ISBN-13: 978-1-951442-91-0

Rev: 1.0

https://www.payloadbooks.com.

Chapter 21

21. An Overview of Compose State and Recomposition

State is the cornerstone of how the Compose system is implemented. As such, a clear understanding of state is an essential step in becoming a proficient Compose developer. In this chapter, we will explore and demonstrate the basic concepts of state and explain the meaning of related terms such as *recomposition*, *unidirectional data flow,* and *state hoisting*. The chapter will also cover saving and restoring state through *configuration changes*.

21.1 The basics of state

In declarative languages such as Compose, *state* is generally referred to as "a value that can change over time". At first glance, this sounds much like any other data in an app. A standard Kotlin variable, for example, is by definition designed to store a value that can change at any time during execution. State, however, differs from a standard variable in two significant ways.

First, the value assigned to a state variable in a composable function needs to be remembered. In other words, each time a composable function containing state (a *stateful function*) is called, it must remember any state values from the last time it was invoked. This is different from a standard variable which would be re-initialized each time a call is made to the function in which it is declared.

The second key difference is that a change to any state variable has far reaching implications for the entire hierarchy tree of composable functions that make up a user interface. To understand why this is the case, we now need to talk about recomposition.

21.2 Introducing recomposition

When developing with Compose, we build apps by creating hierarchies of composable functions. As previously discussed, a composable function can be thought of as taking data and using that data to generate sections of a user interface layout. These elements are then rendered on the screen by the Compose runtime system. In most cases, the data passed from one composable function to another will have been declared as a state variable in a parent function. This means that any change of state value in a parent composable will need to be reflected in any child composables to which the state has been passed. Compose addresses this by performing an operation referred to as *recomposition*.

Recomposition occurs whenever a state value changes within a hierarchy of composable functions. As soon as Compose detects a state change, it works through all of the composable functions in the activity and recomposes any functions affected by the state value change. Recomposing simply means that the function gets called again and passed the new state value.

Recomposing the entire composable tree for a user interface each time a state value changes would be a highly inefficient approach to rendering and updating a user interface. Compose avoids this overhead using a technique called *intelligent recomposition* that involves only recomposing those functions directly affected by the state change. In other words, only functions that read the state value will be recomposed when the value changes.

21.3 Creating the StateExample project

Launch Android Studio and select the New Project option from the welcome screen. Within the resulting new project dialog, choose the *Empty Activity* template before clicking on the Next button.

Enter *StateExample* into the Name field and specify *com.example.stateexample* as the package name. Before clicking on the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo). On completion of the project creation process, the StateExample project should be listed in the Project tool window located along the left-hand edge of the Android Studio main window.

21.4 Declaring state in a composable

The first step in declaring a state value is to wrap it in a MutableState object. MutableState is a Compose class which is referred to as an *observable type*. Any function that reads a state value is said to have *subscribed* to that observable state. As a result, any changes to the state value will trigger the recomposition of all subscribed functions.

Within Android Studio, open the *MainActivity.kt* file, delete the Greeting composable and modify the class so that it reads as follows:

```
package com.example.stateexample
.
.
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            StateExampleTheme {
                Surface(color = MaterialTheme.colorScheme.background) {
                    DemoScreen()
                }
            }
        }
    }
}

@Composable
fun DemoScreen() {
    MyTextField()
}

@Composable
fun MyTextField() {

}

@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    StateExampleTheme {
```

```
            DemoScreen()
        }
}
```

The objective here is to implement MyTextField as a stateful composable function containing a state variable and an event handler that changes the state based on the user's keyboard input. The result is a text field in which the characters appear as they are typed.

MutableState instances are created by making a call to the *mutableStateOf()* runtime function, passing through the initial state value. The following, for example, creates a MutableState instance initialized with an empty String value:

```
var textState = { mutableStateOf("") }
```

This provides an observable state which will trigger a recomposition of all subscribed functions when the contained value is changed. The above declaration is, however, missing a key element. As previously discussed, state must be remembered through recompositions. As currently implemented, the state will be reinitialized to an empty string each time the function in which it is declared is recomposed. To retain the current state value, we need to use the *remember* keyword:

```
var myState = remember { mutableStateOf("") }
```

Remaining within the *MainActivity.kt* file, add some imports and modify the MyTextField composable as follows:

```
.
.
import androidx.compose.material3.*
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.foundation.layout.Column
.
.
@Composable
fun MyTextField() {

    var textState = remember { mutableStateOf("") }

    val onTextChange = { text : String ->
        textState.value = text
    }

    TextField(
        value = textState.value,
        onValueChange = onTextChange
    )
}
```

If the code editor reports that the Material 3 TextField is experimental, modify the MyTextField composable as follows:

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun MyTextField() {
```

```
    var textState by remember { mutableStateOf("") }
.
.
```

Test the code using the Preview panel in interactive mode and confirm that keyboard input appears in the TextField as it is typed.

When looking at Compose code examples, you may see MutableState objects declared in different ways. When using the above format, it is necessary to read and set the *value* property of the MutableState instance. For example, the event handler to update the state reads as follows:

```
val onTextChange = { text: String ->
     textState.value = text
}
```

Similarly, the current state value is assigned to the TextField as follows:

```
TextField(
    value = textState.value,
    onValueChange = onTextChange
)
```

A more common and concise approach to declaring state is to use Kotlin property delegates via the *by* keyword as follows (note that two additional libraries need to be imported when using property delegates):

```
.
.
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue
.
.
@Composable
fun MyTextField() {

    var textState by remember { mutableStateOf("") }
.
.
```

We can now access the state value without needing to directly reference the MutableState *value* property within the event handler:

```
val onTextChange = { text: String ->
     textState = text
}
```

This also makes reading the current value more concise:

```
TextField(
    value = textState,
    onValueChange = onTextChange
 )
```

A third technique separates the access to a MutableState object into a *value* and a *setter function* as follows:

```
var (textValue, setText) = remember { mutableStateOf("") }
```

When changing the value assigned to the state we now do so by calling the *setText* setter, passing through the new value:

```
val onTextChange = { text: String ->
     setText(text)
}
```

The state value is now accessed by referencing *textValue*:

```
TextField(
    value = textValue,
    onValueChange = onTextChange
)
```

In most cases, the use of the *by* keyword and property delegates is the most commonly used technique because it results in cleaner code. Before continuing with the chapter, revert the example to use the *by* keyword.

21.5 Unidirectional data flow

Unidirectional data flow is an approach to app development whereby state stored in a composable should not be directly changed by any child composable functions. Consider, for example, a composable function named FunctionA containing a state value in the form of a Boolean value. This composable calls another composable function named FunctionB that contains a Switch component. The objective is for the switch to update the state value each time the switch position is changed by the user. In this situation, adherence to unidirectional data flow prohibits FunctionB from directly changing the state value.

Instead, FunctionA would declare an event handler (typically in the form of a lambda) and pass it as a parameter to the child composable along with the state value. The Switch within FunctionB would then be configured to call the event handler each time the switch position changes, passing it the current setting value. The event handler in FunctionA will then update the state with the new value.

Make the following changes to the *MainActivity.kt* file to implement FunctionA and FunctionB together with a corresponding modification to the preview composable:

```
@Composable
fun FunctionA() {

    var switchState by remember { mutableStateOf(true) }

    val onSwitchChange = { value : Boolean ->
        switchState = value
    }

    FunctionB(
        switchState = switchState,
        onSwitchChange = onSwitchChange
    )
}

@Composable
fun FunctionB(switchState: Boolean, onSwitchChange : (Boolean) -> Unit ) {
    Switch(
```

```
            checked = switchState,
            onCheckedChange = onSwitchChange
        )
    }

    @Preview(showBackground = true)
    @Composable
    fun GreetingPreview() {
        StateExampleTheme {
            Column {
                DemoScreen()
                FunctionA()
            }
        }
    }
```

Preview the app using interactive mode and verify that clicking the switch changes the slider position between on and off states.

We can now use this example to break down the state process into the following individual steps which occur when FunctionA is called:

1. The *switchState* state variable is initialized with a true value.

2. The *onSwitchChange* event handler is declared to accept a Boolean parameter which it assigns to *switchState* when called.

3. FunctionB is called and passed both *switchState* and a reference to the *onSwitchChange* event handler.

4. FunctionB calls the built-in Switch component and configures it to display the state assigned to *switchState*. The Switch component is also configured to call the *onSwitchChange* event handler when the user changes the switch setting.

5. Compose renders the Switch component on the screen.

The above sequence explains how the Switch component gets rendered on the screen when the app first launches. We can now explore the sequence of events that occur when the user slides the switch to the "off" position:

1. The switch is moved to the "off" position.

2. The Switch component calls the *onSwitchChange* event handler passing through the current switch position value (in this case *false*).

3. The *onSwitchChange* lambda declared in FunctionA assigns the new value to *switchState*.

4. Compose detects that the *switchState* state value has changed and initiates a recomposition.

5. Compose identifies that FunctionB contains code that reads the value of *switchState* and therefore needs to be recomposed.

6. Compose calls FunctionB with the latest state value and the reference to the event handler.

7. FunctionB calls the Switch composable and configures it with the state and event handler.

8. Compose renders the Switch on the screen, this time with the switch in the "off" position.

The key point to note about this process is that the value assigned to *switchState* is only changed from within FunctionA and never directly updated by FunctionB. The Switch setting is not moved from the "on" position to the "off" position directly by FunctionB. Instead, the state is changed by calling upwards to the event handler located in FunctionA, and allowing recomposition to regenerate the Switch with the new position setting.

As a general rule, data is passed down through a composable hierarchy tree while events are called upwards to handlers in ancestor components as illustrated in Figure 21-1:

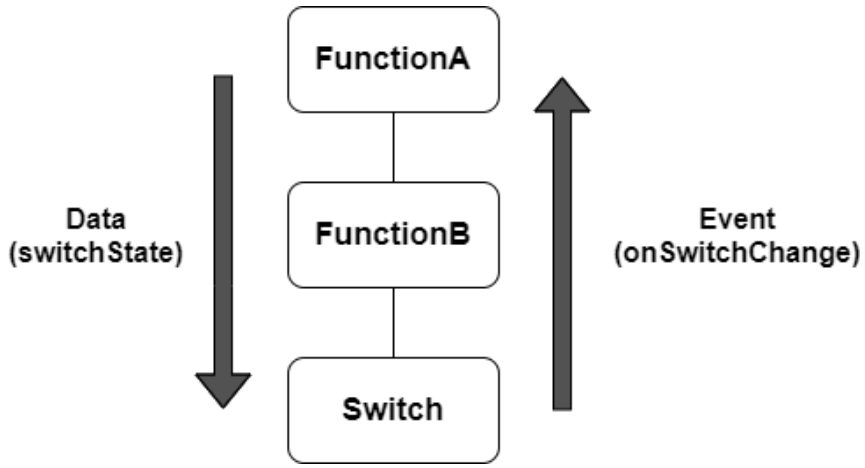


Figure 21-1

21.6 State hoisting

If you look up the word "hoist" in a dictionary it will likely be defined as the act of raising or lifting something. The term *state hoisting* has a similar meaning in that it involves moving state from a child composable up to the calling (parent) composable or a higher ancestor. When the child composable is called by the parent, it is passed the state along with an event handler. When an event occurs in the child composable that requires an update to the state, a call is made to the event handler passing through the new value as outlined earlier in the chapter. This has the advantage of making the child composable stateless and, therefore, easier to reuse. It also allows the state to be passed down to other child composables later in the app development process.

Consider our MyTextField example from earlier in the chapter:

```
@Composable
fun DemoScreen() {
    MyTextField()
}

@Composable
fun MyTextField() {

    var textState by remember { mutableStateOf("") }

    val onTextChange = { text : String ->
        textState = text
    }
```

```
    TextField(
        value = textState,
        onValueChange = onTextChange
    )
}
```

The self-contained nature of the MyTextField composable means that it is not a particularly useful component. One issue is that the text entered by the user is not accessible to the calling function and, therefore, cannot be passed to any sibling functions. It is also not possible to pass a different state and event handler through to the function, thereby limiting its re-usability.

To make the function more useful we need to hoist the state into the parent DemoScreen function as follows:

```
@Composable
fun DemoScreen() {

    var textState by remember { mutableStateOf("") }

    val onTextChange = { text : String ->
        textState = text
    }

    MyTextField(text = textState, onTextChange = onTextChange)
}

@Composable
fun MyTextField(text: String, onTextChange : (String) -> Unit) {

    var textState by remember { mutableStateOf("") }
—
    val onTextChange = { text : String ->
        textState = text
    }

    TextField(
        value = text,
        onValueChange = onTextChange
    )
}

@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    StateExampleTheme {
        DemoScreen()
    }
}
```