Chapter 2: Handling UI State with Jetpack ViewModel

In this chapter, we will cover one of the most important libraries in Jetpack: **ViewModel**. In the first section, *Understanding the Jetpack ViewModel*, we will explore the concept and usages behind the `ViewModel` architecture component. We will see what it is, why we need it in our apps, and how we can implement one in our Restaurants app, which we started in the previous chapter.

In the next section, *Defining and handling state with Compose*, we will study how state is managed in Compose and exemplify usages of state inside our project. Afterward, in the *Hoisting state in Compose* section, we will understand what state hoisting is, why we need to achieve it, and then we will apply it to our app.

Finally, in the *Recovering from system-initiated process death* section, we will cover what a system-initiated process death is, how it occurs, and how essential it is for our applications to be able to recover from it by restoring the previous state details.

To summarize, in this chapter, we're going to cover the following main topics:

- Understanding the Jetpack ViewModel
- Defining and handling state with Compose
- Hoisting state in Compose
- Recovering from system-initiated process death

Before jumping in, let's set up the technical requirements for this chapter.

Technical requirements

When building Compose-based Android projects with Jetpack ViewModel, you usually require your day-to-day tools. However, to follow along smoothly, make sure you have the following:

- The Arctic Fox 2020.3.1 version of Android Studio. You can also use a newer Android Studio version or even Canary builds but note that IDE interface and other generated code files might differ from the ones used throughout this book.
- Kotlin 1.6.10 or newer plugin installed in Android Studio.
- The Restaurants app code from the previous chapter.

The starting point for this chapter is represented by the Restaurants app that we developed in the previous chapter. If you haven't followed the implementation side by side, access the starting point for this chapter by navigating to the **`Chapter_01`** directory of this book's GitHub repository and importing the Android project entitled **`chapter_1_restaurants_app`**.

To access the solution code for this chapter, navigate to the **`Chapter_02`** folder:

https://github.com/PacktPublishing/Kickstart-Modern-Android-Development-with-Jetpack-and-Kotlin/tree/main/Chapter_02/chapter_2_restaurants_app.

The project coding solution for the Restaurants app that we will develop throughout this chapter can be found in the **`chapter_2_restaurants_app`** Android project folder, which you can import.

Understanding the Jetpack ViewModel

While developing Android applications, you must have heard of the term **ViewModel**. If you haven't heard of it, then don't worry – this section

aims to clearly illustrate what this component represents and why we need it in the first place.

To summarize, this section will cover the following topics:

- What is a ViewModel?
- Why do you need ViewModels?
- Introducing Android Jetpack ViewModel
- Implementing your first ViewModel

Let's start with the first question: what is this `ViewModel` that we keep hearing about in Android?

What is a ViewModel?

Initially, the `ViewModel` was designed to allow developers to persist UI state across configuration changes. In time, the `ViewModel` became a way to also recover from edge cases such as system-initiated process death.

However, often, Android apps require you to write code that is responsible for getting the data from the server, transforming it, caching it, and then displaying it. To delegate some work, developers made use of this separate component, which should model the UI (also called the **View**) – the *ViewModel*.

So, we can perceive a `ViewModel` class as a component that manages and caches the UI's state:

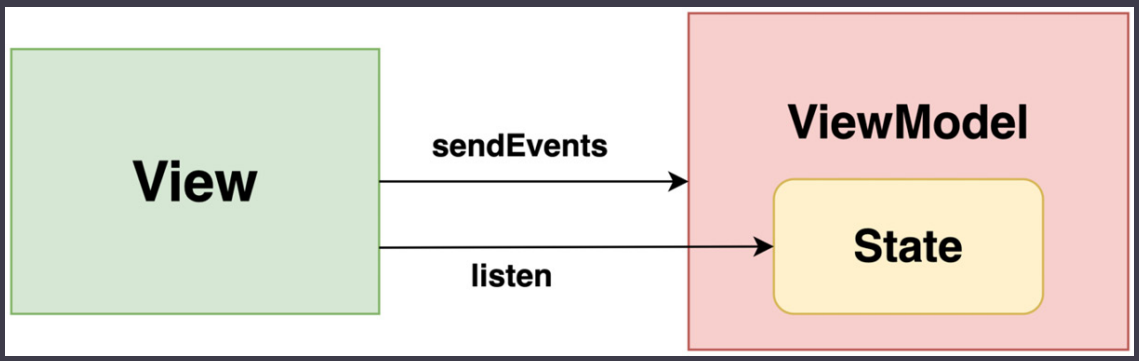

Figure 2.1 – ViewModel stores the state and receives interactions

As we can see, `ViewModel` not only handles the UI state and provides it to the UI but also receives user interaction events from `View` and updates the state accordingly.

In Android, the views are usually represented by **UI controllers** such as `Activity`, `Fragment`, or `Composable` since they are intended to display the UI data. These components are prone to being recreated when configuration changes occur, so `ViewModel` must find a way to cache and then restore the UI state – more on this in the next section, *Why do you need ViewModels?*.

NOTE

`ViewModel` *oversees what data is sent back to the UI controllers and how the UI state reacts to user-generated events. That's why we can call* `ViewModel` *as a master of the UI controller – since it represents the authority that performs decision-making for UI-related events.*

We can try to enumerate some core activities that a `ViewModel` should perform. `ViewModel` should be able to do the following:

- Hold, manage, and preserve the entire UI state.
- Request data or reload content from the server or other sources.
- Prepare data to be displayed by applying various transformations (such as map, sort, filter, and so on).
- Accept user interaction events and change the state based on those events.

Even though you now understand what a `ViewModel` is, you might be wondering, why do we need a separate class that holds the UI state or that prepares data to be displayed? Why can't we do that directly in the UI, in `Activity`, `Fragment`, or even inside the Composable? We'll address this question next.

Why do you need ViewModels?

Imagine that we put all the state-handling logic inside the UI classes. Following this approach, we may soon add other logic for handling network requests, caching, or any other implementation details – everything will be inside the UI layer.

Obviously, this is not a great approach. If we do that, we will end up with an **`Activity`**, **`Fragment`**, or **`composable`** function that has way too many responsibilities. In other words, our UI components will become bloated with so much code and so many responsibilities, thus making the entire project difficult to maintain, fix, or extend.

`ViewModel` is an architecture component that alleviates these potential issues. By adding **`ViewModel`** components to our projects, we are taking the first step toward a solid *architecture* since we can delegate the responsibilities of a UI controller to components such as **`ViewModel`**.

NOTE

`ViewModel` *should not have a reference to a UI controller and should run independently of it. This reduces coupling between the UI layer and* **`ViewModel`** *and allows multiple UI components to reuse the same* **`ViewModel`**.

Preventing multiple responsibilities in UI controllers is the cornerstone of a good system architecture since it promotes a very simple principle called **separation of concerns**. This principle states that every component/module within our app should have and handle one concern.

If, in our case, we add the entire application logic inside **`Activity`**, **`Fragment`**, or **`composable`**, these components will become huge pieces of code that violate the separation of concerns principle, simply because they know how to do everything: from displaying the UI to getting data and serving their UI states. To alleviate this, we can start implementing ViewModels.

Next, we'll see how ViewModels are designed in Android.

Introducing Android Jetpack ViewModel

Creating a **`ViewModel`** class to govern the UI state of a **`View`** is doable and straightforward. We can simply create a separate class and move the corresponding logic there.

However, as we mentioned previously, UI controllers have their own lifecycle: the **`Activity`** or **`Fragment`** objects have their own lifecycles, while composables have a composition cycle. That's why UI controllers are usually fragile and end up being recreated when different events occur, such as a configuration change or a process death. When this happens, any UI state information is lost.

Moreover, UI controllers usually need to make async calls (to obtain data from the server, for example) that have to be managed correctly. This means that when the system destroys UI controllers (such as by calling **`onDestroy()`** on an **`Activity`**), you need to manually interrupt or cancel any pending or ongoing work. Otherwise, your application can leak memory since your UI controller's memory reference cannot be freed up by the system. This is because it's still trying to finish some asynchronous work.

To preserve the UI state and to manage async work easier, our **`ViewModel`** class should be able to get around these *downsides*. But how?

Jetpack ViewModel comes to the rescue! Because the Android **`ViewModel`** is lifecycle aware, this means that it knows how to outlive events such as configuration changes, which are triggered by the user.

It does that by having a *lifecycle scope* tied to the lifecycle of its UI controller. Let's see how the lifecycle of an **`Activity`** and a **`composable`** are defined as opposed to the one of **`ViewModel`**:

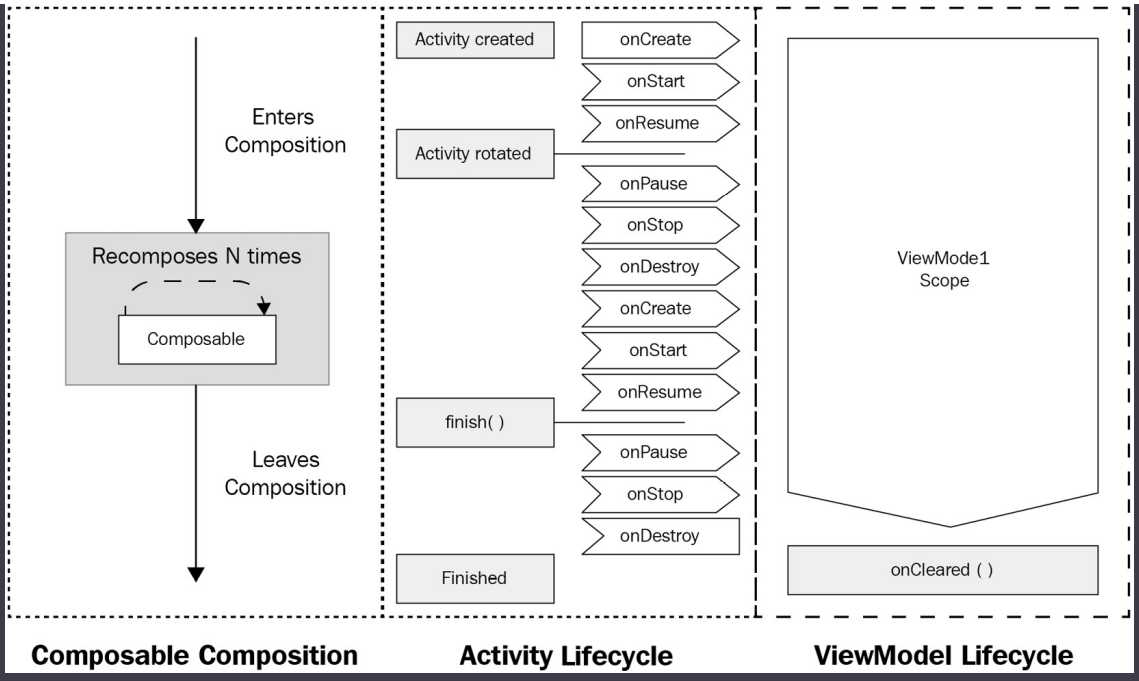

Figure 2.2 – The ViewModel's lifecycle in comparison to UI controller's lifecycle

IMPORTANT NOTE

When the **`ViewModel`** *is used in Compose, it lives by default for as long as the parent* **`Fragment`** *or* **`Activity`** *does. For the* **`ViewModel`** *to live as long as a top-level composable (or screen composable) function does, as shown in the previous diagram, the composable must be used in conjunction with a navigation library. More granular composables can have smaller lifetimes. Don't worry, we will cover the aspect of scoping the lifetime of a ViewModel to the lifetime of a screen composable in Chapter 5, Adding Navigation in Compose with Jetpack Navigation.*

When the UI is recreated or recomposed because of such events, the **`ViewModel`**'s lifecycle awareness allows it to outlive those events and avoid being destroyed, thus allowing the state to be preserved. When the entire lifecycle is finalized, the **`ViewModel`**'s **`onCleared()`** method is called to allow you to easily clean up any pending async work.

Yet one question arises: how can the Jetpack ViewModel do that?

By design, the **ViewModel** classes outlive specific instantiations of **LifecycleOwners**. In our case, UI controllers are **LifecycleOwners** since they have a designated lifecycle, and they can be **Activity** or **Fragment** objects.

To understand how **ViewModel** components are scoped to a specific **Lifecycle**, let's have a look at a traditional way of getting a reference to a **ViewModel** instance:

```
val vm = ViewModelProvider(this)
[MyViewModel::class.java]
```

To obtain an instance of **MyViewModel**, we pass a **ViewModelStoreOwner** to the **ViewModelProvider** constructor. We used to get our **ViewModel** like this in **Activity** or **Fragment** classes, so this is a reference to the current **ViewModelStoreOwner**.

To control the lifetime of the instance of our **MyViewModel**, **ViewModelProvider** needs an instance of **ViewModelStoreOwner** because when it creates an instance of **MyViewModel**, it will link the lifetime of this instance to the lifetime of **ViewModelStoreOwner** – that is, of our **Activity**.

The **Activity** or **Fragment** components are **LifecycleOwners** with a lifecycle, meaning that every time you get a reference to your **ViewModel**, the object you receive is scoped to the **LifecycleOwner**'s lifecycle. This means that your **ViewModel** remains alive in memory until the **LifecycleOwner**'s lifecycle is finished.

NOTE

We will explain the inner workings of **ViewModel** *components and how they are scoped to the lifecycle of a* **LifecycleOwner** *in more detail in Chapter 12, Exploring the Jetpack Lifecycle Components.*

In Compose, the **ViewModel** objects are instantiated differently by using a special inline function called **viewModel()**, which abstracts all the boilerplate code that was needed previously.

NOTE

Optionally, if you need to pass parameters whose values are decided at run-time to your **`ViewModel`**, *you can create and pass a* **`ViewModelFactory`** *instance to the* **`viewModel()`** *constructor.* **`ViewModelFactory`** *is a special class that allows you to control the way your* **`ViewModel`** *is instantiated.*

Now that we have provided an overview of how the Android **`ViewModel`** works, let's create one!

Implementing your first ViewModel

It's time to create a **`ViewModel`** inside the Restaurants application that we created in the previous chapter. To do this, follow these steps:

1. First, create a new file by left-clicking the application package, selecting **New**, and then selecting **Kotlin Class/File**. Enter **`RestaurantsViewModel`** as the name and select **File** as the type. Inside the newly created file, add the following code:

```
import androidx.lifecycle.ViewModel
class RestaurantsViewModel(): ViewModel() {
   fun getRestaurants() = dummyRestaurants
}
```

Our **`RestaurantsViewModel`** inherits from the **`ViewModel`** class (previously referenced as the Jetpack ViewModel) that's defined in **`androidx.lifecycle.ViewModel`**, so it becomes lifecycle aware of the components that instantiate it.

Moreover, we've added the **`getRestaurants()`** method to our **`ViewModel`**, allowing it to be the provider of our **`dummyRestaurants`** list – a first and shy step toward giving it responsibility for governing the UI state.

Next, it's time to prepare to instantiate our **`RestaurantsViewModel`**. In Compose, we can't use the previous syntax for instantiating **`ViewModel`** ob-

jects, so we will use a special and dedicated syntax instead.

2. To gain access to this special syntax, go to the **`build.gradle`** file in the app module and inside the **`dependencies`** block, add the ViewModel-Compose dependency:

```
dependencies {
      […]
    debugImplementation "androidx.compose.ui:ui-
        tooling:$compose_version"
    implementation "androidx.lifecycle:lifecycle-
        viewmodel-compose:2.4.1"
}
```

After updating the **`build.gradle`** file, make sure to sync your project with its Gradle files. You can do that by clicking on the **File** menu option and then by selecting **Sync Project with Gradle Files**.

3. Going back to the **`RestaurantsScreen`** file, we want to instantiate our **`RestaurantsViewModel`** inside our **`RestaurantsScreen`** composable function. We can do this using the **`viewModel()`** inline function syntax and specifying the type of **`ViewModel`** that we expect; that is, **`RestaurantsViewModel`**:

```
@Composable
fun RestaurantsScreen() {
   val viewModel: RestaurantsViewModel =
viewModel()
   LazyColumn( … ) {
       items(viewModel.getRestaurants()) {
restaurant->
            RestaurantItem(restaurant)
       }
   }
}
```

Behind the scenes, the **`viewModel()`** function gets the default **`ViewModelStoreOwner`** for our **`RestaurantsScreen()`** composable. Since we haven't implemented a navigation library, the default **`ViewModelStoreOwner`** will be the calling parent of our composable – the **`MainActivity`** component. This means that for now, even though our **`RestaurantsViewModel`** has been instantiated inside a composable, it will live for as long as our **`MainActivity`** does.

In other words, our **`RestaurantsViewModel`** is scoped to the lifecycle of our **`MainActivity`**, thereby outliving our **`RestaurantsScreens`** composable, or any other composable we would pass to the **`setContent()`** method call from within **`MainActivity`**.

To make sure that our **`ViewModel`** lives for as long as the composable function that needs it does, we will implement a navigation library in *Chapter 5, Adding Navigation in Compose with Jetpack Navigation*.

We also made sure that we now get the restaurants to be displayed from our **`RestaurantsViewModel`** by calling **`getRestaurants()`** on the **`viewModel`** variable.

NOTE

From this point on, on certain older Compose versions, the Compose Preview functionality might not work as expected anymore. As the **`RestaurantsScreen`** *composable now depends on a* **`RestaurantsViewModel`** *object, Compose can fail to infer the data that is passed to the previewed composable, thereby not being able to show us the content. That's why directly referencing a* **`ViewModel`** *inside your screen composable isn't a good practice. We will fix this in Chapter 8, Getting Started with Clean Architecture in Android. Alternatively, to see any changes in your code, you can just run the application on your emulator or physical device.*

Getting back to our Restaurants app, we have successfully added a **`ViewModel`**, yet our **`RestaurantsViewModel`** doesn't handle any state for our

UI. It only sends a hardcoded list of restaurants, which has no state. We envisioned that its purpose is to govern the state of the UI, so let's take a break from `ViewModel` and work on understanding state.

Defining and handling state with Compose

State and events are essential to any application since their existence implies that the UI can change over time as you interact with it.

In this section, we will cover the concept of state and events and then integrate them into our Restaurants app.

To summarize, this section will cover the following topics:

- Understanding state and events
- Adding state to our Restaurants app

Let's start by exploring the basic concepts of state and events in Android applications.

Understanding state and events

State represents a possible form of the UI at a certain point in time. This form can change or mutate. When the user interacts with the UI, an event is created that triggers a change in the state of the UI. So, an **event** is represented by different interactions that are initiated by the user that target the app and that consequently cause its state to update.

In simple terms, state changes over time because of events. The UI, on the other hand, should observe the changes within the state so that it can update accordingly:

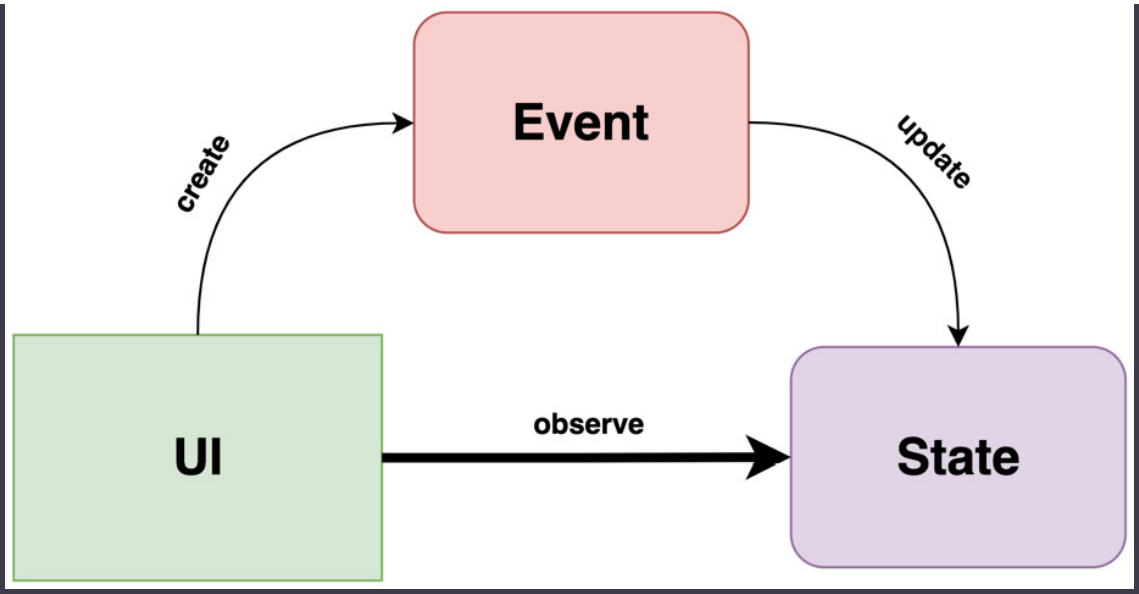

Figure 2.3 – UI update flow

In Compose, composable functions are, by default, stateless. That's why, when we tried to use a **`TextField`** composable in the previous chapter, it didn't present anything to the UI that we typed in with the keyboard. This happened because the composable had no state defined and it didn't get recomposed with the new values that had to be displayed!

This is why, in Compose, it's our job to define state objects for our composables. With the help of *state* objects, we make sure that recomposition is triggered every time a state object's value is changed.

To make such a **`TextField`** display the text that we are typing in, remember that we added a **`textState`** variable. Our **`TextField`** needed such a state object that holds a **`String`** value. This value represents the text that's written by us, which can change as we keep on typing:

```
@Composable
fun NameInput() {
   val textState: MutableState<String> =
       remember { mutableStateOf("") }
   TextField(…)
}
```

Let's have a closer look at how we defined a state object for our **`TextField`**:

- First, we created a variable to hold our state object and made sure that its value can change over time by making it **`MutableState`**. We did that by defining a **`textState`** variable that is of type **`MutableState`**, which, in turn, holds data of type **`String`**.

At its core, **`textState`** is a **`androidx.compose.runtime.State`** object, yet since we want to be able to change its value over time, we directly used a **`MutableState`** that implements **`State`**.

- We instantiated **`textState`** with the **`mutableStateOf("")`** constructor to create a state object and passed an initial value of the data that it holds: an empty string.

We also wrapped the **`mutableStateOf("")`** constructor inside a **`remember { }`** block. The **remember** block allows the state value to be preserved across recompositions. Every time the UI is recomposed because other composables received new data or maybe because of an animation, this state value will be the same because of the **`remember`** block.

Now that we've covered how state objects are defined, some questions remain: how can we alter the state to retrigger recomposition and how can we make sure our **`TextField`** accesses the updated values from our **`textState`**? Let's add these missing pieces:

```
@Composable
fun NameInput() {
   val textState = remember { mutableStateOf("") }
   TextField(
            value = textState.value,
            onValueChange = { newValue ->
                textState.value = newValue
            },
```

```
            label = { Text("Your name") })
}
```

Let's have a closer look at how we wired everything up inside **`TextField`**:

- For **`TextField`** to always have access to the latest value of the **`textState`** state object, we obtained the current state value with the **`.value`** accessor using **`textState.value`**. Then, we passed it to the **`TextField`**'s **`value`** parameter to display it.
- To change the state value, we made use of the **`onValueChange`** callback, which can be portrayed as an *event*. Inside this callback, we updated the **`textState`** state value by using the same **`.value`** accessor and set the new value that was received, called **`newValue`**. Since we updated a **`State`** object, the UI should recompose and our **`TextField`** should render the new input value from the keyboard. This will repeat for as long as we keep on writing.

Now that we have got the hang of defining and altering state in Compose, it's time to add such state functionality to our Restaurants app.

Adding state to our Restaurants app

Let's imagine that the user can scroll through the list of restaurants and then tap a particular one, thereby marking it as a favorite. For this to be more suggestive, we will add a heart icon for each restaurant. To do this, follow these steps:

1. Inside the **`RestaurantsScreen.kt`** file, add another composable inside **`RestaurantItem`** called **`FavoriteIcon`**. Then, pass a weight of **`0.15f`** to make it occupy 15% of the parent **`Row`**:

```
@Composable
fun RestaurantItem(item: Restaurant) {
   Card(...) {
       Row(...) {
           RestaurantIcon(...,
Modifier.weight(0.15f))
```

```
            RestaurantDetails(...,
Modifier.weight(0.7f))
            FavoriteIcon(Modifier.weight(0.15f))
        }
    }
}
```

We have also made sure to decrease the weight of **`RestaurantDetails`** from 85% to 70%.

2. Still inside the **`RestaurantsScreen.kt`** file, define the missing **`FavoriteIcon`** composable, which receives an **`imageVector`** as a predefined icon with **`Icons.Filled.FavoriteBorder`**. Also, make it receive a **`Modifier`** object with **`8.dp`** of padding:

```
@Composable
private fun FavoriteIcon(modifier: Modifier) {
    Image(
        imageVector = Icons.Filled.FavoriteBorder,
        contentDescription = "Favorite restaurant
icon",
        modifier = modifier.padding(8.dp))
}
```

3. If we try to refresh the preview or run the app, we can see several **`RestaurantItem`** composables similar to the following:

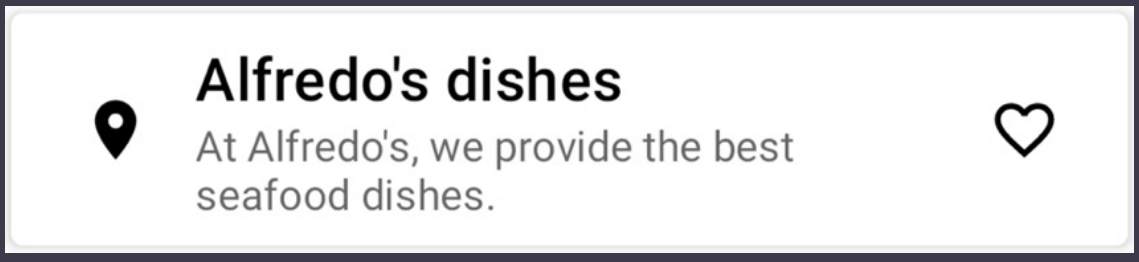

Figure 2.4 – The RestaurantItem composable with a favorite icon

Our **`RestaurantItem`** composable now has a favorite icon. However, when we click it, nothing happens. Clicking it should change the heart icon into

a filled one, marking the restaurant as a favorite. To fix this, we must add a state that allows us to hold the favorite status of a restaurant.

4. Add state to the **`FavoriteIcon`** composable by adding the following code:

```
@Composable
private fun FavoriteIcon(modifier: Modifier) {
   val favoriteState = remember {
       mutableStateOf(false) }
   val icon = if (favoriteState.value)
       Icons.Filled.Favorite
   else
       Icons.Filled.FavoriteBorder
    Image(
        imageVector = icon,
        contentDescription = "Favorite restaurant
icon",
        modifier = modifier
            .padding(8.dp)
            .clickable { favoriteState.value =
                   !favoriteState.value
            }
    )
}
```

To hold the state of being a favorite or not and to trigger a change in this state value, we've done the following:

1. We added a **`favoriteState`** variable that holds a **`MutableState`** of type **`Boolean`** with an initial value of **`false`**. As usual, we wrap the **`mutableStateOf`** constructor inside a **`remember`** block to preserve the state's value across recompositions.
2. We defined an **`icon`** variable that can hold a value of **`Icons.Filled.Favorite`**, which means that the restaurant is your fa-

vorite, or a value of **`Icons.Filled.FavoriteBorder`**, which means that the restaurant is not your favorite.

3. We passed the value of **`icon`** value to the **`imageVector`** parameter of our **`Image`** composable.
4. We added a **`clickable`** modifier that's chained after the **`padding`** one. In this callback, we made sure to update **`favoriteState`** with the **`.value`** accessor by obtaining it and writing the previously negated value.
   NOTE
   When defining state objects in Compose, you can replace the assignment (=) operator with property delegation, which can be achieved with the **`by`** *operator:* **`val favoriteState by remember { … }`**. *By doing this, you will not need to use the* **`.value`** *accessor anymore as it is delegated.*

When we're running or live previewing the application, we can see that upon clicking the empty heart icon of each restaurant, it becomes filled, marking the restaurant as a favorite:

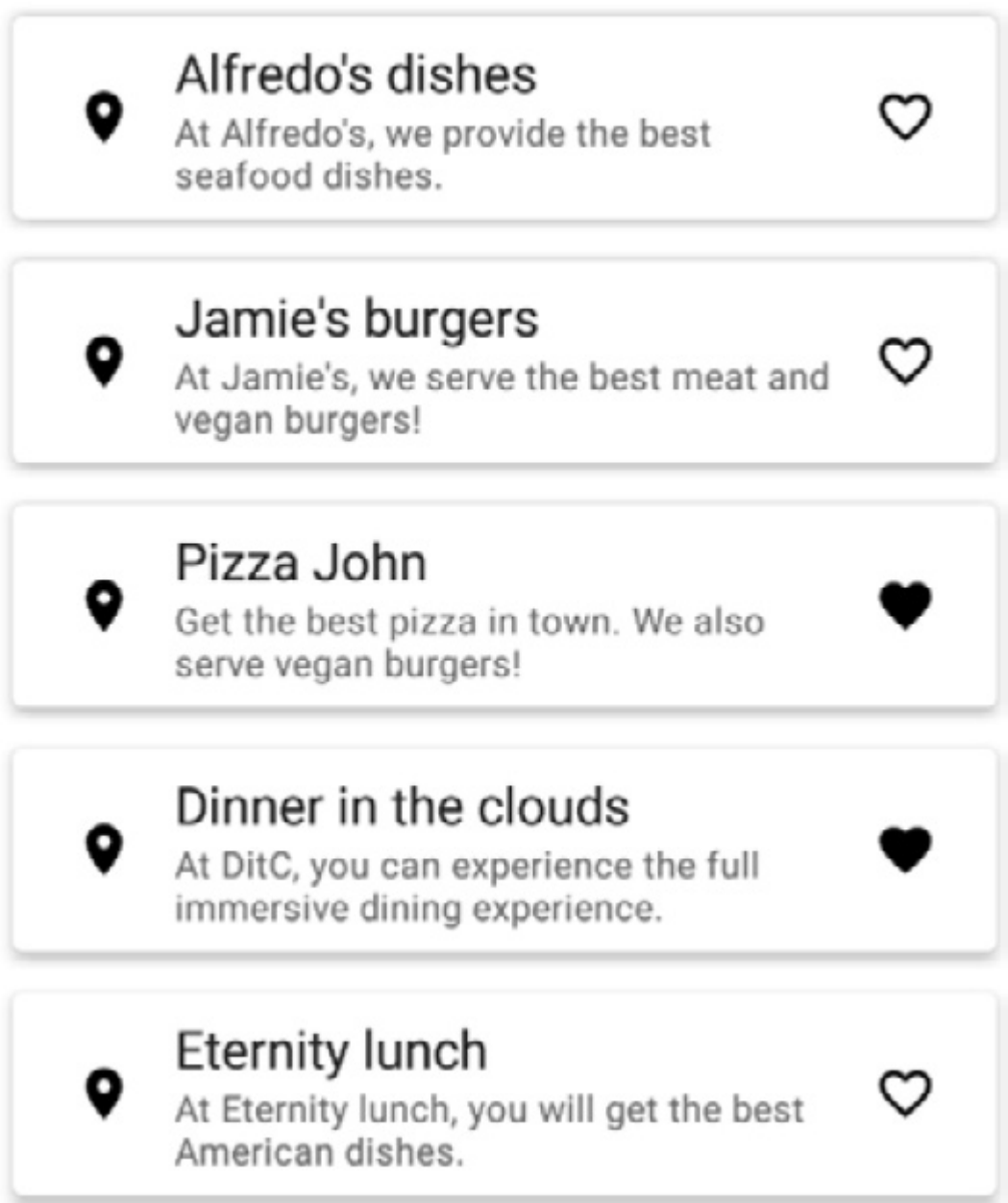

Figure 2.5 – The RestaurantsScreen composable with a favorite state for its items

Most of the time, keeping state and state handling logic inside composable functions is not recommended. Let's explore why this is not the best practice and how we can improve the way we manage state with the help of state hoisting.

Hoisting state in Compose

Composable functions are usually categorized in terms of state handling in two main categories:

- **Stateful**, which allows the composable to hold and manage its state. Stateful composables are those functions where the caller (or parent composable) doesn't need to manage the state. They model basic UI interactions such as animations or expanding content, and they are usually okay to be stateful and hold a **`State`** object.
- **Stateless**, which allows the composable to delegate the state management and to forward event callbacks to its parent composable. Composables that not only impact the UI as their state changes but are also of interest to the presentation or business logic are usually ok to be stateless. This way, the **`ViewModel`** component can be the only source of truth for their state to control and manage UI changes but also to avoid illegal states.

In our case, state changes when a restaurant is marked as a favorite or not. Since we want to control this interaction at the presentation level in the **`ViewModel`** class to keep track of which restaurants have been favorited, we need to move the state up from the **`FavoriteIcon`** composable.

The pattern of moving state up from a composable to its caller composable is called **state hoisting**. To achieve this, we must replace the **`State`** object with two parameters:

- One **`value`** parameter for the data that defines the current state
- A callback function that is triggered as an event when a new value is emitted

By receiving data as input and forwarding events to the parent composable, we make sure that our Compose UI obeys the previously introduced concept of the unidirectional flow of state and events. This concept defines how state values and events should only flow in one direction: the events upwards and the state downwards, and with state hoisting, we enforce just that.

The benefits of state hoisting are as follows:

- **Single source of truth for the state**: The state of our Compose UI can have a single source of truth: the parent composable or, even better, **`ViewModel`**. Composables can be decoupled from their state to avoid illegal states in your UI.
- **Reusability**: Since composables only render the data that's received as input, it's much easier to reuse them within other composables as you can simply pass different values.
- **Encapsulation limitation**: Only stateful composables can change their state internally. This means that you can limit the number of composables that handle their state, which could lead to illegal UI states.

Now that we've briefly covered what state hoisting is and why it is beneficial, it's time to hoist the state within our Restaurants application:

1. First, lift the state from the **`FavoriteIcon`** composable by removing the existing **`favoriteState`** and **`icon`** variables along with their instantiation logic from the top of the body of the function. At the same time, update the **`FavoriteIcon`** composable to accept an **`icon`** parameter for receiving input data and also an **`onClick`** event callback for forwarding upwards events:

```
@Composable
private fun FavoriteIcon(icon: ImageVector,
                         modifier: Modifier,
                         onClick: () -> Unit) {
   Image(
       imageVector = icon,
       contentDescription = "Favorite restaurant
icon",
       modifier = [...]
           .clickable { onClick() })
}
```

Additionally, we passed **`icon`** to the **`imageVector`** parameter of the **`Image`** composable and triggered the **`onClick`** callback function whenever the **`clickable`** event is triggered. By applying these changes, we lifted the state up and transformed **`FavoriteIcon`** from a stateful composable into a stateless one.

2. Now, move the **`favoriteState`** variable in the **`RestaurantItem`** parent composable of **`FavoriteIcon`**. The **`RestaurantItem`** composable provides the state to **`FavoriteIcon`** and is also in charge of updating its state over time:

```
@Composable
fun RestaurantItem(item: Restaurant) {
   val favoriteState = remember {
       mutableStateOf(false) }
   val icon = if (favoriteState.value)
       Icons.Filled.Favorite
   else
       Icons.Filled.FavoriteBorder
   Card(...) {
       Row(...) {
          [...]
          FavoriteIcon(icon,
Modifier.weight(0.15f)) {
               favoriteState.value =
                   !favoriteState.value
          }
       }
   }
}
```

The corresponding **`icon`** for each state is now passed to **`FavoriteIcon`**. Additionally, **`RestaurantItem`** is now listening for **`onClick`** events in the trailing lambda block, where it mutates the **`favoriteState`** object, triggering recomposition upon every click.

Yet, looking at **FavoriteIcon** and **RestaurantIcon**, we can see many similarities. Both are stateless composables that receive an **ImageVector** as a parameter. Since they are stateless and perform similar functions, let's reuse one of them and delete the other.

3. Inside **RestaurantIcon**, add a similar **onClick** function parameter (just like **FavoriteIcon** has) and bind it to the **clickable** modifier's callback:

```
@Composable
private fun RestaurantIcon(icon: ImageVector,
modifier: Modifier, onClick: () -> Unit = { }) {
   Image([...],
       modifier = modifier
           .padding(8.dp)
           .clickable { onClick() }
   )
}
```

Since we don't want to execute anything on click events for the restaurant profile icon, we provided a default empty function (**{ }**) value to the **onClick** parameter.

Once you've done this, you can delete the **FavoriteIcon** composable since we won't need it anymore.

4. Inside the **RestaurantItem** composable, replace **FavoriteIcon** with **RestaurantIcon**:

```
@Composable
fun RestaurantItem(item: Restaurant) {
   val favoriteState = ...
   Card(...) {
       Row(...) {
          RestaurantIcon(…)
          RestaurantDetails(...)
          RestaurantIcon(icon,
Modifier.weight(0.15f)) {
```

```
                favoriteState.value =
!favoriteState.value
            }
        }
    }
}
```

You have now hoisted the state from **`RestaurantIcon`** to the **`RestaurantItem`** composable.

Let's keep on hoisting the state even further uphill, into the **`RestaurantsScreen`** composable. However, we cannot keep individual **`State`** objects for each **`RestaurantItem`** inside this composable, so we will have to change the **`State`** object to hold a list of **`Restaurant`** objects, each having a separate **`isFavorite`** value.

5. Inside the **`Restaurant.kt`** file, add another property for **`Restaurant`** called **`isFavorite`**. It should have a default value of **`false`** since, by default, restaurants are not marked as favorites when the application starts:

```
data class Restaurant(val id: Int,
                      val title: String,
                      val description: String,
                      var isFavorite: Boolean =
false)
val dummyRestaurants = listOf(…)
```

6. Going back inside the **`RestaurantsScreen.kt`** file, hoist the state up again, this time from **`RestaurantItem`**, by adding an **`onClick`** function parameter that's triggered inside the **`RestaurantIcon`**'s callback function parameter. We won't add a new argument for the input data since we already have the **`item`** argument of type **`Restaurant`**, and you can also safely remove the **`favoriteState`** variable since we won't be needing it anymore:

```
@Composable
fun RestaurantItem(item: Restaurant,
```

```
                    onClick: (id: Int) -> Unit) {
    val icon = if (item.isFavorite)
        Icons.Filled.Favorite
    else
        Icons.Filled.FavoriteBorder
    Card(...) {
        Row(...) {
            ...
           RestaurantIcon(…)
           RestaurantDetails(…)
           RestaurantIcon(…) {
               onClick(item.id)
           }
        }
    }
}
```

This time, the **item** parameter will be our **Restaurant** object. **Restaurant** now holds an **isFavorite: Boolean** property that states whether the restaurant is favorited or not. That's why we set the correct value for the **icon** variable based on the item's field by checking the **item.isFavorite** value.

Now, **RestaurantItem** is a stateless composable, so it's time to add a **State** object to its parent.

7. Inside **RestaurantsScreen**, add a **state** variable that will hold our list of restaurants. Its type will be **MutableState<List<Restaurant>>** and we will set the restaurants from **viewModel** as it initial value, finally passing the state's **value** to the **items** constructor of **LazyColumn**:

```
@Composable
fun RestaurantsScreen() {
  val viewModel: RestaurantsViewModel = viewModel()
  val state: MutableState<List<Restaurant>> =
    remember {
```

```
        mutableStateOf(viewModel.getRestaurants())
      }
    LazyColumn(...) {
     items(state.value) { restaurant ->
       RestaurantItem(restaurant) { id ->
         val restaurants =
state.value.toMutableList()
         val itemIndex =
           restaurants.indexOfFirst { it.id == id }
         val item = restaurants[itemIndex]
         restaurants[itemIndex] =
           item.copy(isFavorite = !item.isFavorite)
         state.value = restaurants
        }
      }
    }
}
```

Inside **RestaurantItem**'s **onClick** trailing lambda block, we must toggle the favorite status of the corresponding restaurant and update the state. Because of this, we did the following:

1. We obtained the current list of restaurants by calling **state.value** and converting it into a mutable list so that we could replace the item whose **isFavorite** field's value should be updated.
2. We obtained the index of the item whose **isFavorite** field should be updated via the **indexOfFirst** function, where we matched the **id** property of the **Restaurant** objects.
3. Having found **itemIndex**, we obtained the **item** object of type **Restaurant** and applied the **copy()** constructor, where we negated the **isFavorite** field. The resulting value replaced the existing **item** at **itemIndex**.
4. Finally, we passed the updated **restaurants** list back to the **state** object with the **.value** accessor.

NOTE

For Compose to observe changes within a list of objects of type **`T`** *called* **`List<T>`**, *where* **`T`** *is a data class, you must update the memory reference of the updated item. You can do that by calling the* **`copy()`** *constructor of* **`T`** *so that when the updated list is passed back to your* **`State`** *object, Compose triggers a recomposition. Alternatively, you can use* **`mutableStateListOf<Restaurant>()`** *to have easier recomposition events triggered.*

If we try to run the app, we should notice that the functionality is the same, yet the state was hoisted and that we can now reuse composables such as **`RestaurantItem`** or **`RestaurantIcon`** much easier.

But what happens if we toggle a couple of restaurants that are favorites and then rotate the device, thereby changing the screen orientation?

Even though we used the **`remember`** block to preserve the state across re-compositions, our selections were lost, and all the restaurants are marked as not favorites again. This is because the **`MainActivity`** host of our **`RestaurantsScreen`** composable has been recreated, so any state was also lost when the configuration change occurred.

To fix this, we can do the following:

- Replace the **`remember`** block with **`rememberSaveable`**. This will allow the state to be automatically saved across configuration changes of the host **`Activity`**.
- Hoist the state to **`ViewModel`**. We know that **`RestaurantsViewModel`** is not scoped to the lifecycle of our **`RestaurantsScreen`** yet since no navigation library was used, so this means it's scoped to **`MainActivity`**, which allows it to survive configuration changes.

You can try replacing the **`remember`** block with **`rememberSaveable`** and then rotate the screen to see that the state is now preserved across configuration changes. However, we want to take the high road and make sure **`ViewModel`** is the only source of truth for our state. Let's get started:

1. To lift the state to **`ViewModel`**, we must move the **`State`** object from the **`RestaurantsScreen`** composable to the **`RestaurantsViewModel`** and we must also create a new method called **`toggleFavorite`** that will allow the **`RestaurantsViewModel`** to mutate the value of the **`state`** variable every time we try to toggle the favorite status of a restaurant:

```
class RestaurantsViewModel() : ViewModel() {
   val state = mutableStateOf(dummyRestaurants)
    fun toggleFavorite(id: Int) {
        val restaurants =
state.value.toMutableList()
        val itemIndex =
            restaurants.indexOfFirst { it.id == id
}
        val item = restaurants[itemIndex]
        restaurants[itemIndex] =
            item.copy(isFavorite =
!item.isFavorite)
        state.value = restaurants
    }
}
```

The new method called **`toggleFavorite`** accepts the **`id`** property of the targeted restaurant. Inside this method, we moved the code from the **`RestaurantItem`**'s **`onClick`** trailing lambda block, where we toggle the favorite status of the corresponding item and update its state.

By this time, you can safely remove the **`getRestaurants()`** method from the **`RestaurantsViewModel`** class since we won't be needing it anymore.

NOTE

The **`State`** *object that's contained within the* **`ViewModel`** *should not be publicly available for other classes to modify it, since we want it to be encapsulated and allow only the* **`ViewModel`** *to update it. We will fix this in Chapter 7, Introducing Presentation Patterns in Android.*

2. Inside the **`RestaurantsScreen`** composable, remove the **`state`** variable and pass the restaurants from **`RestaurantsViewModel`** by accessing the value of its state through the **`.value`** accessor with **`viewModel.state.value`**:

```
fun RestaurantsScreen() {
   val viewModel: RestaurantsViewModel =
viewModel()
   LazyColumn(...) {
       items(viewModel.state.value) { restaurant ->
           RestaurantItem(restaurant) { id ->
               viewModel.toggleFavorite(id)
           }
       }
   }
}
```

We also removed the old code from the **`RestaurantItem`**'s **`onClick`** trailing lambda block and replaced it with a call to our **`ViewModel`**'s **`toggleFavorite`** method.

If you run the application, the UI should perform as expected, so you should be able to toggle any restaurants as favorite and your selections should be saved upon events like orientation change.

The only difference is that now, **`RestaurantsViewModel`** is the only source of truth for the state of **`RestaurantsScreen`** and we no longer need to hold or save the UI state inside the composables themselves.

We now know how to hoist the state up into the **`ViewModel`**. Now, let's cover a very important scenario in the world of Android that's related to process death.

Recovering from system-initiated process death

We've already learned how, whenever a configuration change occurs, our `Activity` is recreated, which can cause our UI to lose its state. To bypass this issue and to preserve the UI's state, we ended up implementing a `ViewModel` component and hoisted the UI state there.

But what would happen in the case of a system-initiated process death?

A **system-initiated process death** happens when the user places our application in the background and decides to use other apps for a while – in the meantime, though, the system decides to kill our app's process to free up system resources, which initiates process death.

Let's try to simulate such an event and see what happens:

1. Start the Restaurants app using the IDE's **Run** button and mark some restaurants as favorites:

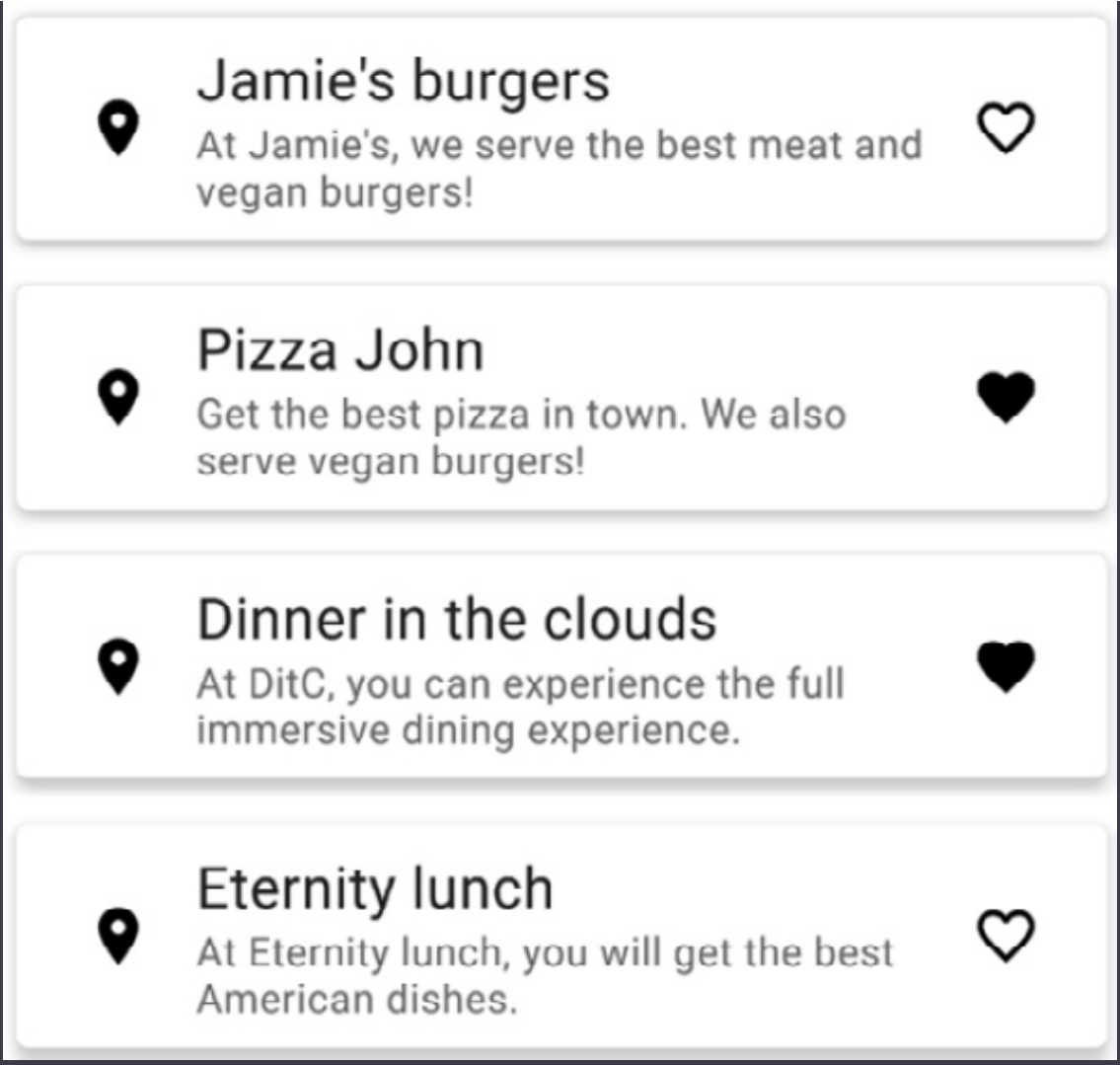

Figure 2.6 – The RestaurantsScreen composable with favorite selections made

2. Place the app in the background by pressing the **Home** button on the device/emulator.
3. In Android Studio, select the **Logcat** window and then press the red square button on the left-hand side to terminate the application:

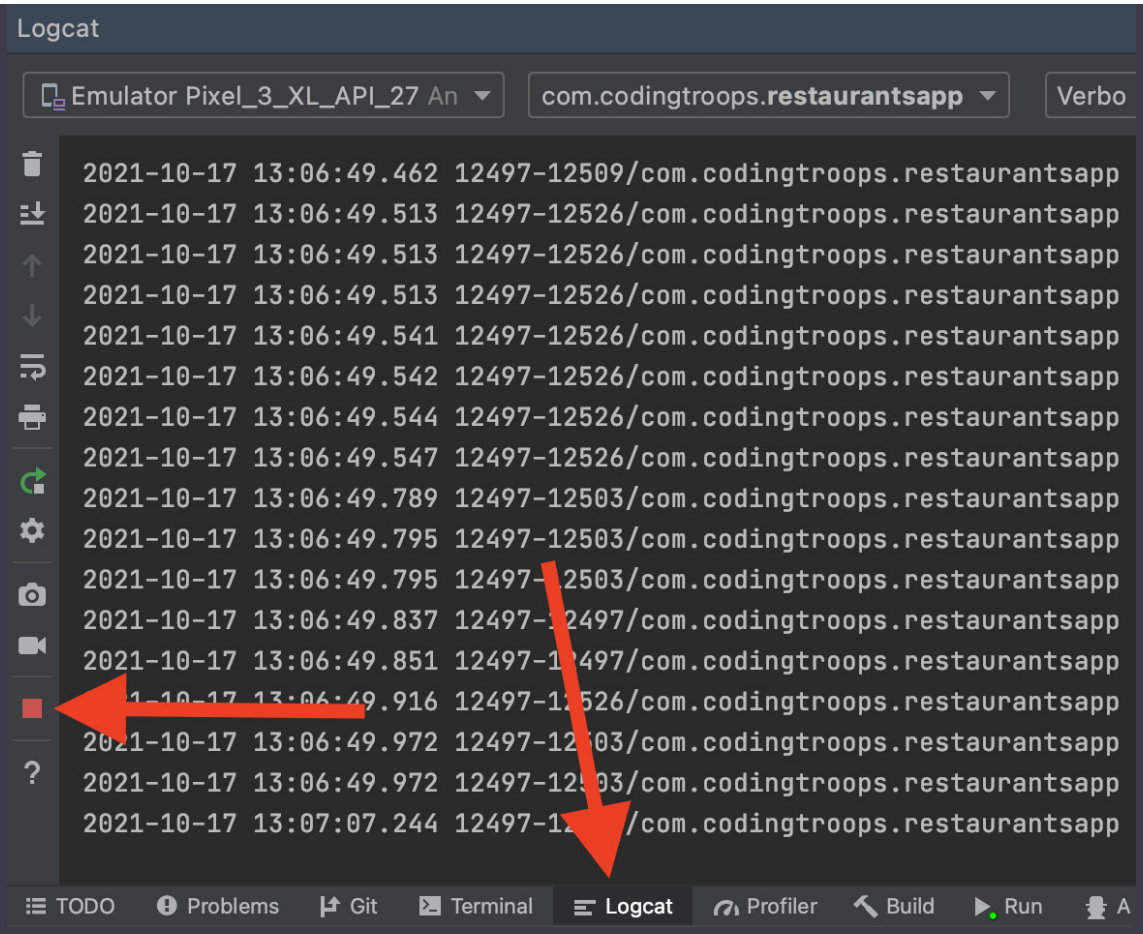

Figure 2.7 – Killing the process in Logcat to simulate system-initiated process death

4. Relaunch the application from the application drawer:

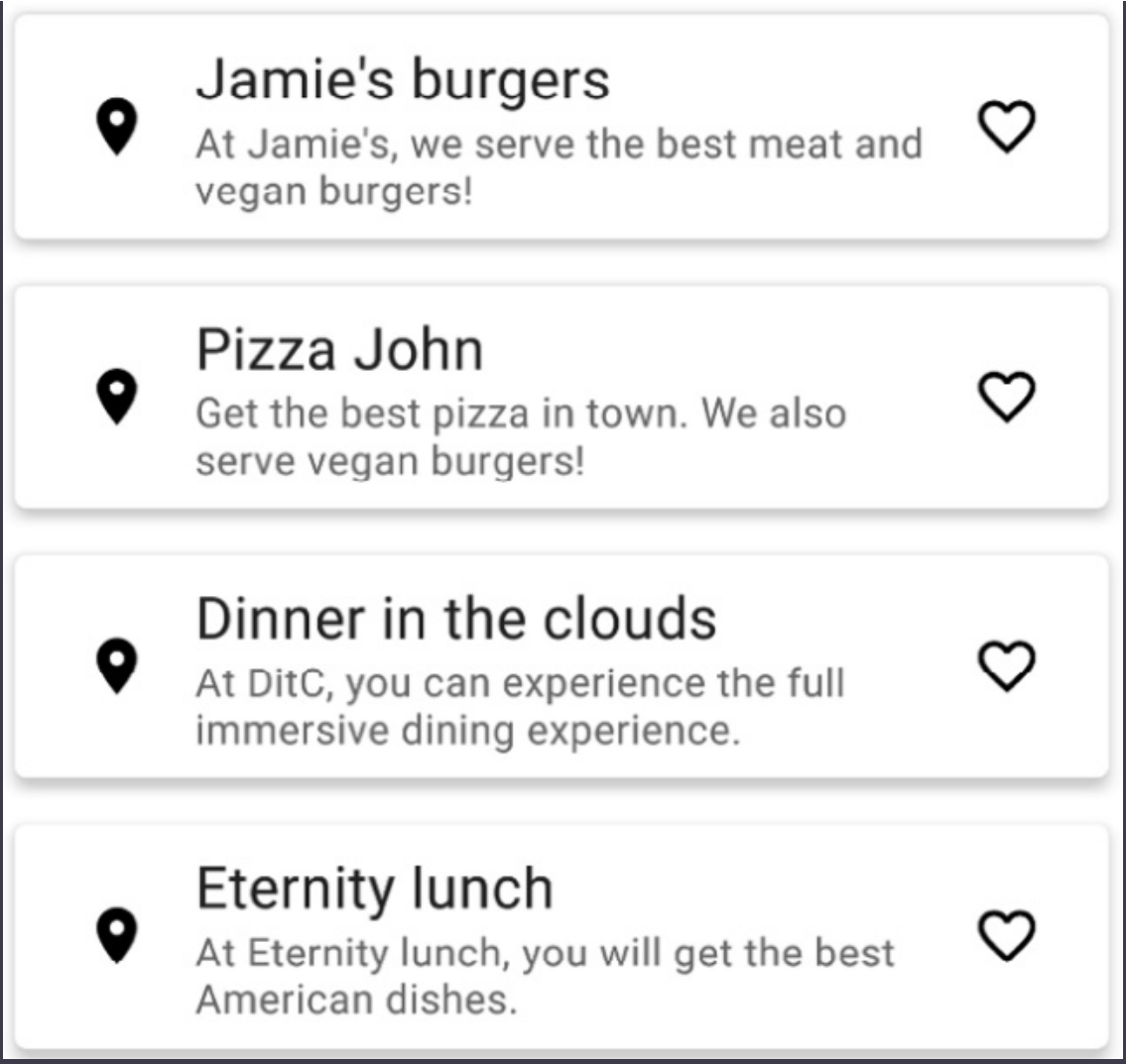

Figure 2.8 – The RestaurantsScreen composable with favorite selections lost

We have now simulated a situation where the system would kill our process. When we return to the app, we can see that our selections are now gone and that the restaurants that were favorited are now in their default states.

To restore state upon system-initiated process death, we used to use the **Saved State module**, which allowed us to save state-related details in the `onSaveInstanceState()` callback of our activity.

Similarly, every `ViewModel` that uses the default `ViewModelFactory` (like we did with the `viewModel()` inline syntax previously) can access a `SavedStateHandle` object through its constructor. If you use a custom

`ViewModelFactory`, make sure that it extends **`AbstractSavedStateViewModelFactory`**.

The **`SavedStateHandle`** object is a key-value map that allows you to save and then restore objects that are crucial to your state. This map survives the event of process death when this event is initiated by the system, which allows you to retrieve and restore your saved objects.

NOTE

When we're saving state-related data, it's crucial to save lightweight objects that define the state and not the entire data that is described on the screen. For large data, we should use local persistence.

Let's try to do this in our application by saving a list of **`id`** values of the restaurants that were toggled as favorites in **`SavedStateHandle`**. Saving the **`id`** values is better than saving the entire list of restaurants since a list of **`Int`** values is lightweight. And since we can always get the restaurant list back at runtime, the only thing that's missing is to remember which of them were favorited.

NOTE

Usually, **`SavedStateHandle`** *is used for saving transient data like sorting or filtering selections performed by the user, or other selections that you need to restore upon system-initiated process death. In our case though, favorited restaurants should be restored not only upon system-initiated process death but also upon a simple application restart. That's why we will save these selections as part of the domain data of the app inside a local database later in Chapter 6, Adding Offline Capabilities with Jetpack Room.*

Let's use a **`SavedStateHandle`** object to recover from system-initiated process death:

1. Add the **`SavedStateHandle`** parameter to your **`RestaurantsViewModel`**:

```
class RestaurantsViewModel(
```

```
    private val stateHandle: SavedStateHandle) :
        ViewModel() {
    …
}
```

2. Call a **`storeSelection`** method whenever we toggle the favorite status of a restaurant inside the **`toggleFavorite`** method and pass the respective restaurant:

```
class RestaurantsViewModel(…) {
    fun toggleFavorite(id: Int) {
        …
        restaurants[itemIndex] =
item.copy(isFavorite =
            !item.isFavorite)
        storeSelection(restaurants[itemIndex])
        state.value = restaurants
    }
...
}
```

This code won't compile though because we haven't yet defined the **`storeSelection`** method. Let's do that up next.

3. Inside **`RestaurantsViewModel`**, create a new **`storeSelection`** method that receives a **`Restaurant`** object whose **`isFavorite`** property has just been altered, and saves that selection inside the **`SavedStateHandle`** object provided by the **`RestaurantsViewModel`** class:

```
private fun storeSelection(item: Restaurant) {
    val savedToggled = stateHandle
      .get<List<Int>?>(FAVORITES)
      .orEmpty().toMutableList()
    if (item.isFavorite) savedToggled.add(item.id)
    else savedToggled.remove(item.id)
    stateHandle[FAVORITES] = savedToggled
}
companion object {
```

```
    const val FAVORITES = "favorites"
}
```

This new method will try to save the **id** value of a restaurant in our **stateHandle** object every time we toggle its favorite status. It does this as follows:

1. It obtains a list containing the IDs of the previously favorited restaurants from **stateHandle** by accessing the **FAVORITES** key inside the map. It stores the result in a **savedToggle** mutable list. If no restaurants were favorited, the list will be empty.
2. If this restaurant was marked as favorite, it adds the ID of the restaurant to the **savedToggle** list. Otherwise, it removes it.
3. Saves the updated list of favorited restaurants with the **FAVORITES** key inside the **stateHandle** map.

We have also added a **companion object** construct to the **RestaurantsViewModel** class as a static extension object. We used this **companion object** to define a constant value for the key used to save the restaurant's selection inside our **stateHandle** map.

Now, we've made sure to cache the selections of favorite restaurants before process death, so our next step is to find a way to restore these selections after the app recovers from a system-initiated process death event.

4. Call a **restoreSelections()** extension method on the **dummyRestaurants** list that we are passing as an initial value to our **state** object. This call should restore the UI selections:

```
class RestaurantsViewModel(
   private val stateHandle: SavedStateHandle):
      ViewModel() {
   val state = mutableStateOf(
     dummyRestaurants.restoreSelections()
   )
    ...
```

```
}
```

This code won't compile though because we haven't yet defined the **restoreSelections** method. Let's do that up next.

5. Inside **RestaurantsViewModel**, define the **restoreSelections** extension function that will allow us to retrieve the restaurants that were favorited upon process death:

```
private fun List<Restaurant>.restoreSelections():
        List<Restaurant> {
    stateHandle.get<List<Int>?>(FAVORITES)?.let {
            selectedIds ->
        val restaurantsMap = this.associateBy {
it.id }
        selectedIds.forEach { id ->
            restaurantsMap[id]?.isFavorite = true
        }
        return restaurantsMap.values.toList()
    }
    return this
}
```

This extension function will allow us to mark those restaurants that were marked by the user previously as favorites upon system-initiated process death. The **restoreSelections** extension function achieves that in the following way:

1. First, by obtaining the list with the unique identifiers of the previously favorited restaurants from **stateHandle** by accessing the **FAVORITES** key inside the map. If the list is not **null**, this means that a process death occurred, and it references the list as **selectedIds**; otherwise, it will return the list without any modifications.
2. Then, by creating a map from the input list of restaurants with the key being the **id** value of the restaurant and the value the **Restaurant** object itself.

3. By iterating over the unique identifiers of the favorited restaurants and for each of them, by trying to access the respective restaurant from our new list and sets its **`isFavorite`** value to **`true`**.
4. By returning the modified restaurants list from **`restaurantMap`**. This list should now contain the restored **`isFavorite`** values from before the death process occurred.

6. Finally, build the app and then repeat *steps 1*, *2*, *3*, and *4* from when we simulated a system-initiated process death.

The application should now correctly display the UI state with the previously favorited restaurants from before the system-initiated process death.

With that, we've made sure that our application not only stores the UI state at the **`ViewModel`** level but that it also can recover from extraordinary events, such as system-initiated process death.

Summary

In this chapter, we learned what a **`ViewModel`** class is, we explored the concepts that define it, and we learned how to instantiate one. We tackled why a **`ViewModel`** is useful as a single source of truth for the UI's *state*: to avoid illegal and undesired states.

For that to make sense, we explored how a UI is defined by its state and how to define such a state in Compose. We then understood what *state hoisting* is and how to separate widgets between *stateless* and *stateful* composables.

Finally, we put all these new concepts into practice by defining state in our Restaurants app, hoisting it, and then lifting it even higher into the newly created **`ViewModel`**.

Finally, we learned how system-initiated process death occurs and how to allow the app to recover by restoring the previous state with the help of **`SavedStateHandle`**.

In the next chapter, we will add real data to our Restaurants app by connecting it to our database using Retrofit.

Further reading

Working with ViewModels and handling state changes in Compose represent two essential topics for reliable projects. Let's see what other subjects revolve around them.

Exploring ViewModel with runtime-provided arguments

In most cases, you can declare and provide dependencies to your **`ViewModel`** inside the constructor, at compile time. In some cases, though, you might need to initialize a **`ViewModel`** instance with a parameter that's only known at runtime.

For example, when we're adding a composable screen that displays the details of a restaurant, instead of sending the ID of the target restaurant from the composable to **`ViewModel`** through a function call, we can provide it directly to the **`ViewModel`** constructor through **ViewModelFactory**.

To explore the process of building a **`ViewModelFactory`**, check out the following Codelab: https://developer.android.com/codelabs/kotlin-android-training-view-model#7.

Exploring ViewModel for Kotlin Multiplatform projects

While this chapter covered the Jetpack ViewModel for Compose in pure Android apps, if you're aiming to build cross-platform projects using **Kotlin Multiplatform (KMP)** or **Kotlin Multiplatform Mobile (KMM)**, the Jetpack ViewModel might not be your best option.

When we're building cross-platform projects, we should try to avoid platform-specific dependencies. The Jetpack ViewModel is suited for Android and therefore is an Android dependency, so we might need to build or define a ViewModel.

To learn more about KMM and platform-agnostic ViewModels, check out the following GitHub example: https://github.com/dbaroncelli/D-KMP-sample.

Understanding how to minimize the number of recompositions

In this chapter, we learned how to trigger recompositions by using **`State`** objects. While in Compose, recompositions happen often, we haven't had a chance to optimize the performance of our Compose-based screens.

We can reduce the number of recompositions by ensuring that the input of the composables is deeply stable. To learn more about how to achieve this, go to https://developer.android.com/jetpack/compose/lifecycle?hl=bn-IN&skip_cache=true#skipping.