ViewModel and State in Compose

1. Before you begin

In the previous codelabs, you learned about the lifecycle of activities and the related lifecycle issues with configuration changes. When a configuration change occurs, you can save an app's data through different ways, such as using rememberSaveable or saving the instance state. However, these options can create problems. Most of the time, you can use rememberSaveable but that might mean keeping the logic in or near composables. When apps grow, you should move data and logic away from composables. In this codelab, you will learn about a robust way to design your app and preserve app data during configuration changes by taking advantage of the Android Jetpack library, ViewModel and Android app architecture guidelines.

Android Jetpack libraries are a collection of libraries to make it easier for you to develop great Android apps. These libraries help you follow best practices, free you from writing boilerplate code, and simplify complex tasks so that you can focus on the code you care about, like the app logic.

App architecture is a set of design rules for an app. Much like the blueprint of a house, your architecture provides the structure for your app. A good app architecture can make your code robust, flexible, scalable, testable, and maintainable for years to come. The Guide to app architecture (<https://developer.android.com/topic/libraries/architecture>) provides recommendation on app architecture and recommended best practices.

In this codelab, you learn how to use ViewModel, one of the architecture components from Android Jetpack libraries that can store your app data. The stored data is not lost if the framework destroys and recreates the activities during a configuration change or other events. However, the data is lost if the activity is destroyed because of process death. The ViewModel only caches data through quick activity recreations.

Prerequisites

* Knowledge of Kotlin, including functions, lambdas, and stateless composables
* Basic knowledge of how to build layouts in Jetpack Compose
* Basic knowledge of Material Design

What you'll learn

* Introduction to the Android app architecture
* How to use the ViewModel class in your app
* How to retain UI data through device configuration changes using a ViewModel

What you'll build

* An Unscramble game app where the user can guess the scrambled words

What you'll need

* The latest version of Android Studio
* An internet connection to download starter code

2. App overview

Game overview

The Unscramble app is a single player word scrambler game. The app displays a scrambled word, and the player has to guess the word using all the letters shown. The player scores points if the word is correct. Otherwise, the player can try to guess the word any number of times. The app also has an option to skip the current word. In the top right corner, the app displays the word count, which is the number of scrambled words played in the current game. There are 10 scrambled words per game.

|  |  |  |
| --- | --- | --- |
|  |  |  |

Get the starter code

To get started, download the starter code:

<https://github.com/google-developer-training/basic-android-kotlin-compose-training-unscramble/archive/refs/heads/starter.zip>

Alternatively, you can clone the GitHub repository for the code:

|  |
| --- |
| $ git clone  https://github.com/google-developer-training/basic-android-kotlin-compose-training-unscramble.git  $ cd basic-android-kotlin-compose-training-unscramble  $ git checkout starter |

**Note**: The starter code is in the starter branch of the downloaded repository.

You can browse the starter code in the Unscramble GitHub repository.

<https://github.com/google-developer-training/basic-android-kotlin-compose-training-unscramble/tree/starter>

3. Starter app overview

To familiarize yourself with the starter code, complete the following steps:

1. Open the project with the starter code in Android Studio.
2. Run the app on an Android device or an emulator.
3. Tap the **Submit** and **Skip** buttons to test the app.

You will notice bugs in the app. The scrambled word does not display, but it is hardcoded to "scrambleun" and nothing happens when you tap the buttons.

In this codelab, you will implement the game functionality using the Android app architecture.

Starter code walk through

The starter code has the pre-designed game screen layout for you. In this pathway, you will implement the game logic. You will use architecture components to implement the recommended app architecture and resolve the above-mentioned issues. Here is a brief walkthrough of some files to get you started.

**WordsData.kt**

This file contains a list of the words used in the game, constants for the maximum number of words per game, and the number of points the player scores for every correct word.

|  |
| --- |
| package com.example.android.unscramble.data  const val MAX\_NO\_OF\_WORDS = 10 const val SCORE\_INCREASE = 20  // Set with all the words for the Game val allWords: Set<String> =    setOf(        "animal",        "auto",        "anecdote",        "alphabet",        "all",        "awesome",        "arise",        "balloon",        "basket",        "bench",       // ...        "zoology",        "zone",        "zeal" ) |

**WARNING**: It is not a recommended practice to hardcode strings in the code. Add strings to strings.xml for easier localization. Strings are hardcoded in this example app for simplicity and to enable you to focus on the app architecture.

**MainActivity.kt**

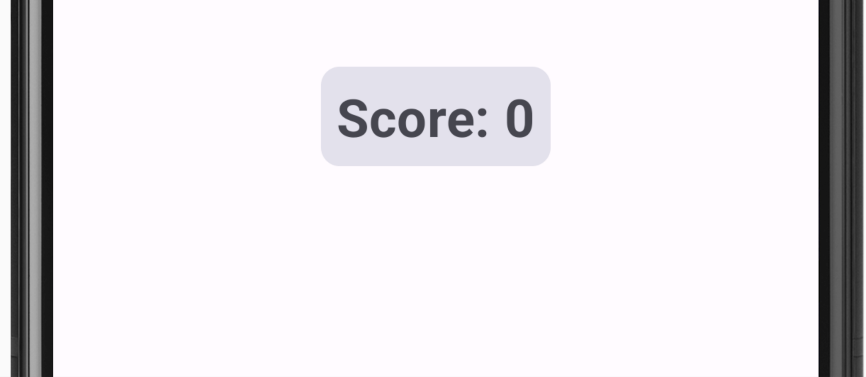
This file contains mostly template generated code. You display the GameScreen composable in the setContent{} block.

**GameScreen.kt**

All the UI composables are defined in the GameScreen.kt file. The following sections provide a walkthrough of some composable functions.

**GameStatus**

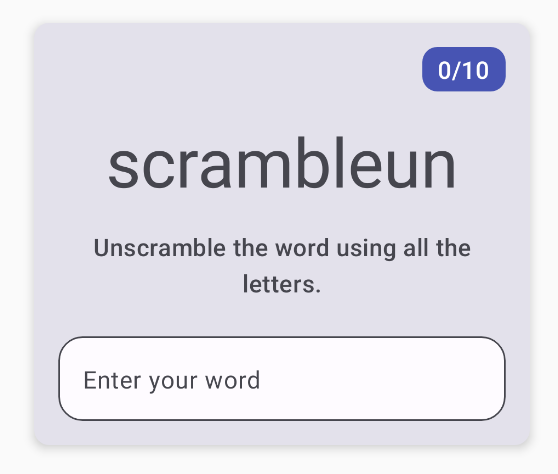
GameStatus is a composable function that displays the game score at the bottom of the screen. The composable function contains a text composable in a Card. For now, the score is hardcoded to 0.



|  |
| --- |
| // No need to copy, this is included in the starter code.  @Composable fun GameStatus(score: Int, modifier: Modifier = Modifier) {     Card(         modifier = modifier     ) {         Text(             text = stringResource(R.string.score, score),             style = typography.headlineMedium,             modifier = Modifier.padding(8.dp)         )     } } |

**GameLayout**

GameLayout is a composable function that displays the main game functionality, which includes the scrambled word, the game instructions, and a text field that accepts the user's guesses.



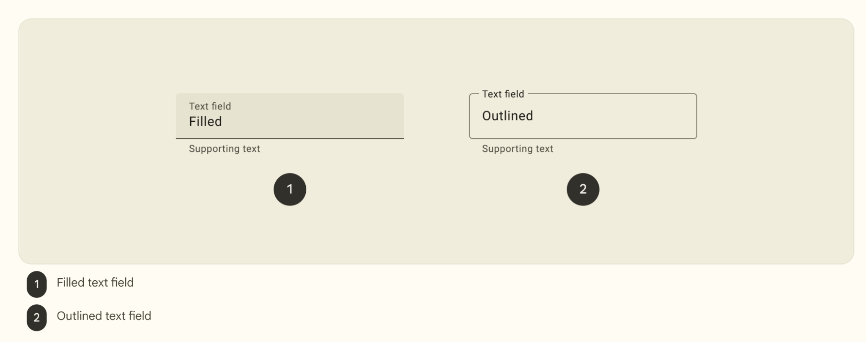
Notice the GameLayout code below contains a column inside a Card with three child elements: the scrambled word text, the instructions text, and the text field for the user's word OutlinedTextField. For now, the scrambled word is hardcoded to be scrambleun. Later in the codelab, you will implement functionality to display a word from the WordsData.kt file.

|  |
| --- |
| // No need to copy, this is included in the starter code.  @Composable fun GameLayout(modifier: Modifier = Modifier) {    val mediumPadding = dimensionResource(R.dimen.padding\_medium)    Card(        modifier = modifier,        elevation = CardDefaults.cardElevation(defaultElevation = 5.dp)    ) {        Column(            verticalArrangement = Arrangement.spacedBy(mediumPadding),            horizontalAlignment = Alignment.CenterHorizontally,            modifier = Modifier.padding(mediumPadding)        ) {            Text(                modifier = Modifier                    .clip(shapes.medium)                    .background(colorScheme.surfaceTint)                    .padding(horizontal = 10.dp, vertical = 4.dp)                    .align(alignment = Alignment.End),                text = stringResource(R.string.word\_count, 0),                style = typography.titleMedium,                color = colorScheme.onPrimary            )            Text(                text = "scrambleun",                style = typography.displayMedium            )            Text(                text = stringResource(R.string.instructions),                textAlign = TextAlign.Center,                style = typography.titleMedium            )            OutlinedTextField(                value = "",                singleLine = true,                shape = shapes.large,                modifier = Modifier.fillMaxWidth(),                colors = TextFieldDefaults.textFieldColors(containerColor = colorScheme.surface),                onValueChange = { },                label = { Text(stringResource(R.string.enter\_your\_word)) },                isError = false,                keyboardOptions = KeyboardOptions.Default.copy(                    imeAction = ImeAction.Done                ),                keyboardActions = KeyboardActions(                    onDone = { }                )            )        }    } } |

The OutlinedTextField composable is similar to the TextField composable from apps in previous codelabs.

Text fields come in two types:

* Filled text fields
* Outlined text fields

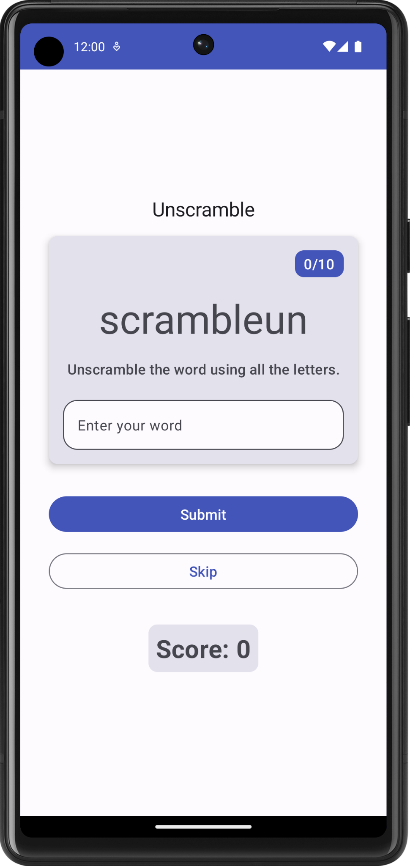


Outlined text fields have less visual emphasis than filled text fields. When they appear in places like forms, where many text fields are placed together, their reduced emphasis helps simplify the layout.

In the starter code, the OutlinedTextField does not update when the user enters a guess. You will update this feature in the codelab.

**GameScreen**

The GameScreen composable contains the GameStatus and GameLayout composable functions, game title, word count and the composables for the **Submit** and **Skip** buttons.

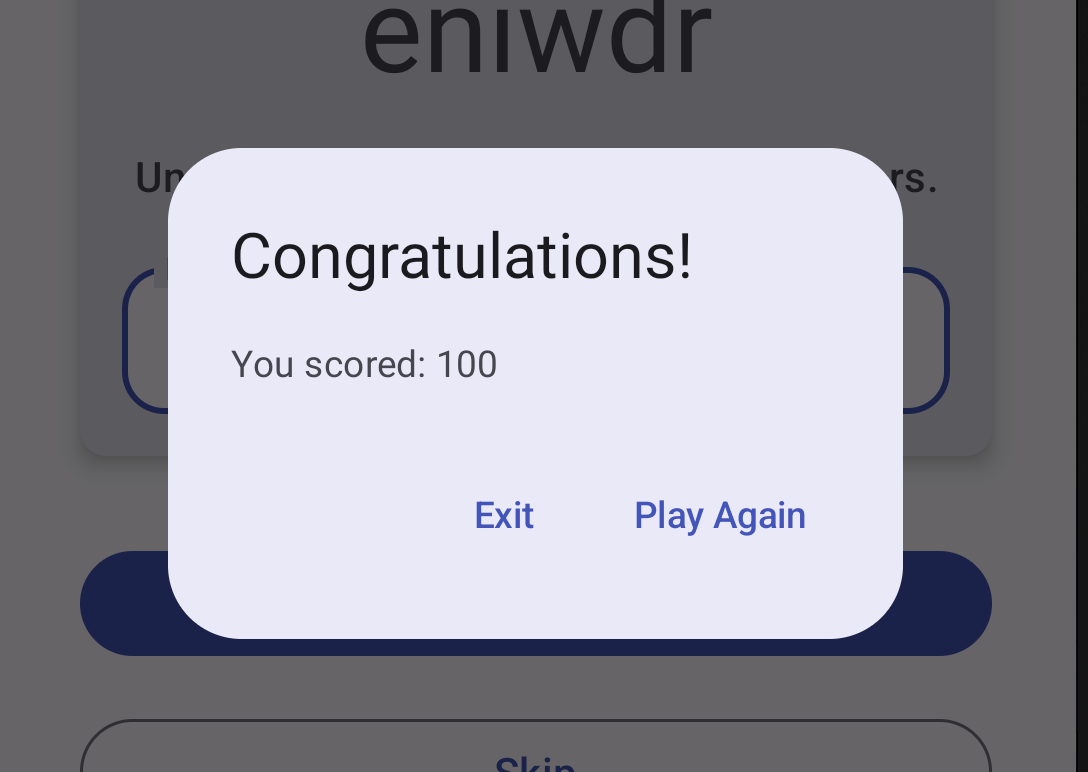


|  |
| --- |
| @Composable fun GameScreen() {     val mediumPadding = dimensionResource(R.dimen.padding\_medium)      Column(         modifier = Modifier             .verticalScroll(rememberScrollState())             .padding(mediumPadding),         verticalArrangement = Arrangement.Center,         horizontalAlignment = Alignment.CenterHorizontally     ) {          Text(             text = stringResource(R.string.app\_name),             style = typography.titleLarge,         )          GameLayout(             modifier = Modifier                 .fillMaxWidth()                 .wrapContentHeight()                 .padding(mediumPadding)         )         Column(             modifier = Modifier                 .fillMaxWidth()                 .padding(mediumPadding),             verticalArrangement = Arrangement.spacedBy(mediumPadding),             horizontalAlignment = Alignment.CenterHorizontally         ) {              Button(                 modifier = Modifier.fillMaxWidth(),                 onClick = { }             ) {                 Text(                     text = stringResource(R.string.submit),                     fontSize = 16.sp                 )             }              OutlinedButton(                 onClick = { },                 modifier = Modifier.fillMaxWidth()             ) {                 Text(                     text = stringResource(R.string.skip),                     fontSize = 16.sp                 )             }         }          GameStatus(score = 0, modifier = Modifier.padding(20.dp))     } } |

Button click events are not implemented in the starter code. You will implement these events as part of the codelab.

**FinalScoreDialog**

The FinalScoreDialog composable displays a dialog–that is, a small window that prompts the user–with options to **Play Again** or **Exit** the game. Later in this codelab, you will implement logic to display this dialog at the end of the game.



|  |
| --- |
| // No need to copy, this is included in the starter code.  @Composable private fun FinalScoreDialog(     score: Int,     onPlayAgain: () -> Unit,     modifier: Modifier = Modifier ) {     val activity = (LocalContext.current as Activity)      AlertDialog(         onDismissRequest = {            // Dismiss the dialog when the user clicks outside the dialog or on the back            // button. If you want to disable that functionality, simply use an empty            // onDismissRequest.         },         title = { Text(text = stringResource(R.string.congratulations)) },         text = { Text(text = stringResource(R.string.you\_scored, score)) },         modifier = modifier,         dismissButton = {             TextButton(                 onClick = {                     activity.finish()                 }             ) {                 Text(text = stringResource(R.string.exit))             }         },         confirmButton = {             TextButton(onClick = onPlayAgain) {                 Text(text = stringResource(R.string.play\_again))             }         }     ) } |

4. Learn about app architecture

An app's architecture provides guidelines to help you allocate the app responsibilities between the classes. A well-designed app architecture helps you scale your app and extend it with additional features. Architecture can also simplify team collaboration.

The most common architectural principles ([https://developer.android.com/jetpack/guide#common-principles](https://developer.android.com/jetpack/guide%23common-principles)) are **separation of concerns** and **driving UI from a model**.

**Separation of concerns**

The separation of concerns design principle states that the app is divided into classes of functions, each with separate responsibilities.

**Drive UI from a model**

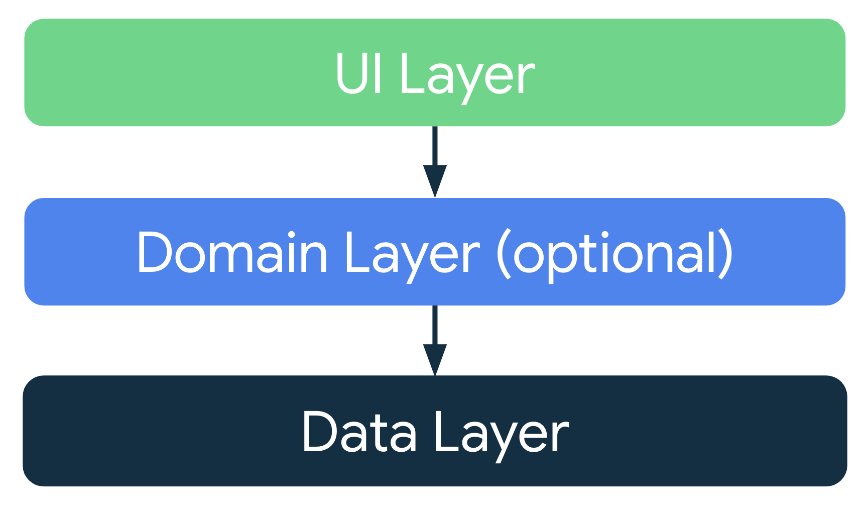
The drive UI from a model principle states that you should drive your UI from a model, preferably a persistent model. Models are components responsible for handling the data for an app. They're independent from the UI elements and app components in your app, so they're unaffected by the app's lifecycle and associated concerns.

Recommended app architecture

Considering the common architectural principles mentioned in the previous section, each app should have at least two layers:

* **UI layer**: a layer that displays the app data on the screen but is independent of the data.
* **Data layer**: a layer that stores, retrieves, and exposes the app data.

You can add another layer, called the domain layer, to simplify and reuse the interactions between the UI and data layers. This layer is optional and beyond the scope of this course.



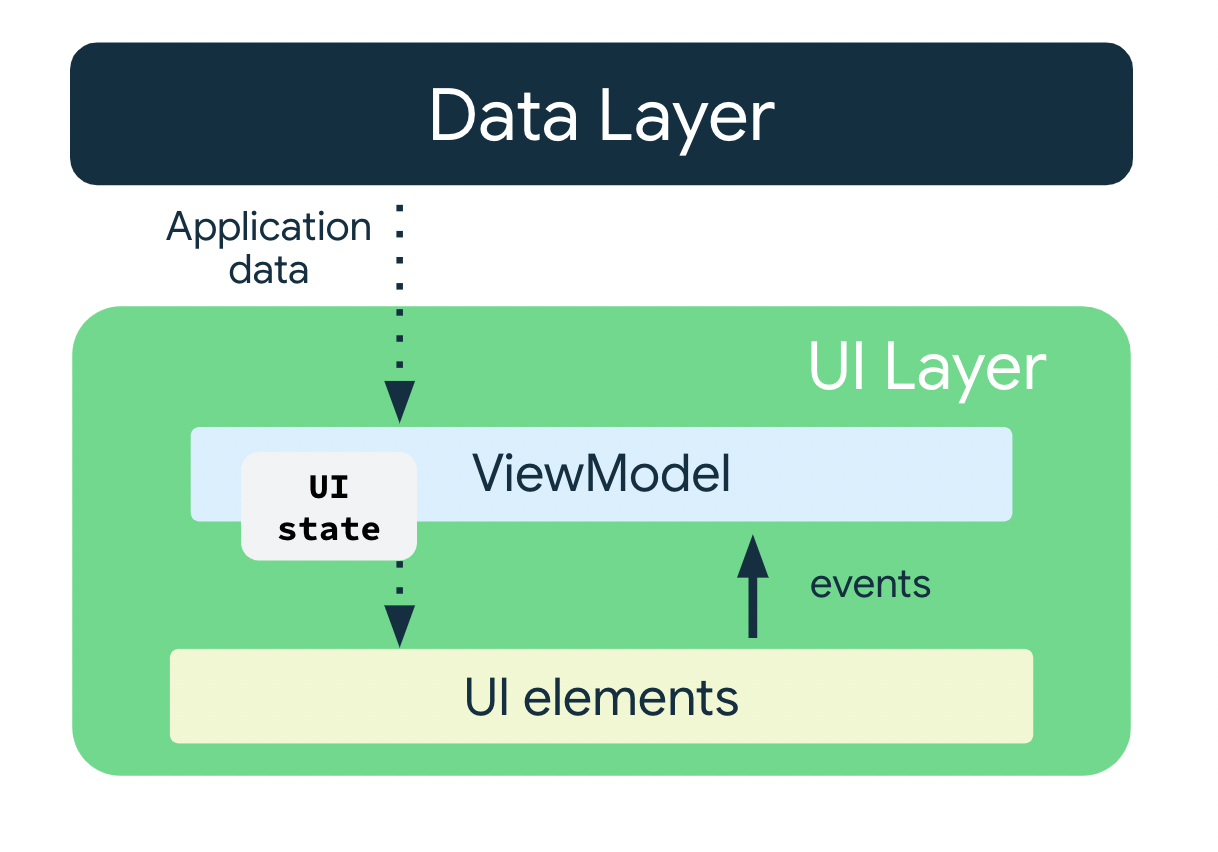
**Note**: The arrows in the diagrams in this guide represent dependencies between classes. For example, the domain layer depends on data layer classes.

UI layer

The role of the UI layer, or presentation layer, is to display the application data on the screen. Whenever the data changes due to a user interaction, such as pressing a button, the UI should update to reflect the changes.

The UI layer is made up of the following components:

* **UI elements**: components that render the data on the screen. You build these elements using Jetpack Compose.
* **State holders**: components that hold the data, expose it to the UI, and handle the app logic. An example state holder is ViewModel.



ViewModel

The ViewModel component holds and exposes the state the UI consumes. The UI state is application data transformed by ViewModel. ViewModel lets your app follow the architecture principle of driving the UI from the model.

ViewModel stores the app-related data that isn't destroyed when the activity is destroyed and recreated by the Android framework. Unlike the activity instance, ViewModel objects are not destroyed. The app automatically retains ViewModel objects during configuration changes so that the data they hold is immediately available after the recomposition.

To implement ViewModel in your app, extend the ViewModel class, which comes from the architecture components library and stores app data within that class.

UI State

The UI is what the user sees, and the UI state is what the app says they should see. The UI is the visual representation of the UI state. Any changes to the UI state immediately are reflected in the UI.



*UI is a result of binding UI elements on the screen with the UI state*.

|  |
| --- |
| // Example of UI state definition, do not copy over  data class NewsItemUiState(     val title: String,     val body: String,     val bookmarked: Boolean = false,     ... ) |

Immutability

The UI state definition in the example above is immutable. Immutable objects provide guarantees that multiple sources do not alter the state of the app at an instant in time. This protection frees the UI to focus on a single role: reading state and updating UI elements accordingly. Therefore, you should never modify the UI state in the UI directly, unless the UI itself is the sole source of its data. Violating this principle results in multiple sources of truth for the same piece of information, leading to data inconsistencies and subtle bugs.

5. Add a ViewModel

In this task, you add a ViewModel to your app to store your game UI state (scrambled word, word count, and score). To resolve the issue in the starter code that you noticed in the previous section, you need to save the game data in the ViewModel.

1. Open build.gradle.kts (Module :app), scroll to the dependencies block and add the following dependency for ViewModel. This dependency is used for adding the lifecycle aware viewmodel to your compose app.

|  |
| --- |
| dependencies { // other dependencies      implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.6.1") //... } |

1. In the ui package, create a Kotlin class/file called GameViewModel. Extend it from the ViewModel class.

|  |
| --- |
| import androidx.lifecycle.ViewModel  class GameViewModel : ViewModel() { } |

1. In the ui package, add a model class for state UI called GameUiState. Make it a data class and add a variable for the current scrambled word.

|  |
| --- |
| data class GameUiState(    val currentScrambledWord: String = "" ) |

StateFlow

StateFlow is a data holder observable flow that emits the current and new state updates. Its value property reflects the current state value. To update state and send it to the flow, assign a new value to the value property of the MutableStateFlow class.

In Android, StateFlow works well with classes that must maintain an observable immutable state.

A StateFlow can be exposed from the GameUiState so that the composables can listen for UI state updates and make the screen state survive configuration changes.

In the GameViewModel class, add the following \_uiState property.

|  |
| --- |
| import kotlinx.coroutines.flow.MutableStateFlow  // Game UI state private val \_uiState = MutableStateFlow(GameUiState()) |

Backing property

A backing property lets you return something from a getter other than the exact object.

For var property, the Kotlin framework generates getters and setters.

For getter and setter methods, you can override one or both of these methods and provide your own custom behavior. To implement a backing property, you override the getter method to return a read-only version of your data. The following example shows a backing property:

|  |
| --- |
| //Example code, no need to copy over  // Declare private mutable variable that can only be modified // within the class it is declared. private var \_count = 0   // Declare another public immutable field and override its getter method.  // Return the private property's value in the getter method. // When count is accessed, the get() function is called and // the value of \_count is returned.  val count: Int     get() = \_count |

As another example, say that you want the app data to be private to the ViewModel:

Inside the ViewModel class:

* The property \_count is private and mutable. Hence, it is only accessible and editable within the ViewModel class.

Outside the ViewModel class:

* The default visibility modifier in Kotlin is public, so count is public and accessible from other classes like UI controllers. A val type cannot have a setter. It is immutable and read-only so you can only override the get() method. When an outside class accesses this property, it returns the value of \_count and its value can't be modified. This backing property protects the app data inside the ViewModel from unwanted and unsafe changes by external classes, but it lets external callers safely access its value.

1. In the GameViewModel.kt file, add a backing property to uiState named \_uiState. Name the property uiState and is of the type StateFlow<GameUiState>.

Now \_uiState is accessible and editable only within the GameViewModel. The UI can read its value using the read-only property, uiState. You can fix the initialization error in the next step.

|  |
| --- |
| import kotlinx.coroutines.flow.StateFlow  // Game UI state  // Backing property to avoid state updates from other classes private val \_uiState = MutableStateFlow(GameUiState()) val uiState: StateFlow<GameUiState> |

1. Set uiState to \_uiState.asStateFlow().

The asStateFlow() makes this mutable state flow a *read-only* state flow.

|  |
| --- |
| import kotlinx.coroutines.flow.StateFlow import kotlinx.coroutines.flow.asStateFlow  // Game UI state private val \_uiState = MutableStateFlow(GameUiState()) val uiState: StateFlow<GameUiState> = \_uiState.asStateFlow() |

Display random scrambled word

In this task, you add helper methods to pick a random word from the WordsData.kt and scramble the word.

1. In the GameViewModel, add a property called currentWord of the type String to save the current scrambled word.

|  |
| --- |
| private lateinit var currentWord: String |

1. Add a helper method to pick a random word from the list and shuffle it. Name it pickRandomWordAndShuffle() with no input parameters and make it return a String.

|  |
| --- |
| import com.example.unscramble.data.allWords  private fun pickRandomWordAndShuffle(): String {    // Continue picking up a new random word until you get one that hasn't been used before    currentWord = allWords.random()    if (usedWords.contains(currentWord)) {        return pickRandomWordAndShuffle()    } else {        usedWords.add(currentWord)        return shuffleCurrentWord(currentWord)    } } |

Android studio flags an error for the undefined variable and function.

1. In the GameViewModel, add the following property after the currentWord property to serve as a mutable set to store used words in the game.

|  |
| --- |
| // Set of words used in the game private var usedWords: MutableSet<String> = mutableSetOf() |

1. Add another helper method to shuffle the current word called shuffleCurrentWord() that takes a String and returns the shuffled String.

|  |
| --- |
| private fun shuffleCurrentWord(word: String): String {    val tempWord = word.toCharArray()    // Scramble the word    tempWord.shuffle()    while (String(tempWord).equals(word)) {        tempWord.shuffle()    }    return String(tempWord) } |

1. Add a helper function to initialize the game called resetGame(). You use this function later to start and restart the game. In this function, clear all the words in the usedWords set, initialize the \_uiState. Pick a new word for currentScrambledWord using pickRandomWordAndShuffle().

|  |
| --- |
| fun resetGame() {    usedWords.clear()    \_uiState.value = GameUiState(currentScrambledWord = pickRandomWordAndShuffle()) } |

1. Add an init block to the GameViewModel and call the resetGame() from it.

|  |
| --- |
| init {    resetGame() } |

When you build your app now, you still see no changes in the UI. You are not passing the data from the ViewModel to the composables in the GameScreen.

6. Architecting your Compose UI

In Compose, the only way to update the UI is by changing the state of the app. What you can control is your UI state. Every time the state of the UI changes, Compose recreates the parts of the UI tree that changed. Composables can accept state and expose events. For example, a TextField/OutlinedTextField accepts a value and exposes a callback onValueChange that requests the callback handler to change the value.

|  |
| --- |
| //Example code no need to copy over  var name by remember { mutableStateOf("") } OutlinedTextField(     value = name,     onValueChange = { name = it },     label = { Text("Name") } ) |

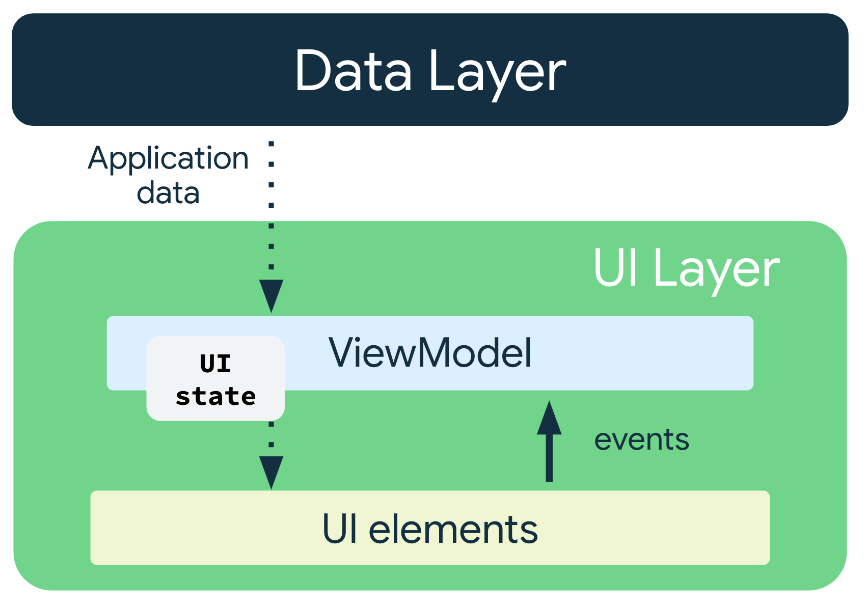
Because composables accept state and expose events, the unidirectional data flow pattern fits well with Jetpack Compose. This section focuses on how to implement the unidirectional data flow pattern in Compose, how to implement events and state holders, and how to work with ViewModels in Compose.

Unidirectional data flow

A *unidirectional data flow* (UDF) is a design pattern in which state flows down and events flow up. By following unidirectional data flow, you can decouple composables that display state in the UI from the parts of your app that store and change state.

The UI update loop for an app using unidirectional data flow looks like the following:

* **Event**: Part of the UI generates an event and passes it upward—such as a button click passed to the ViewModel to handle—or an event that is passed from other layers of your app, such as an indication that the user session has expired.
* **Update state**: An event handler might change the state.
* **Display state**: The state holder passes down the state, and the UI displays it.



The use of the UDF pattern for app architecture has the following implications:

* The ViewModel holds and exposes the state the UI consumes.
* The UI state is application data transformed by the ViewModel.
* The UI notifies the ViewModel of user events.
* The ViewModel handles the user actions and updates the state.
* The updated state is fed back to the UI to render.
* This process repeats for any event that causes a mutation of state.

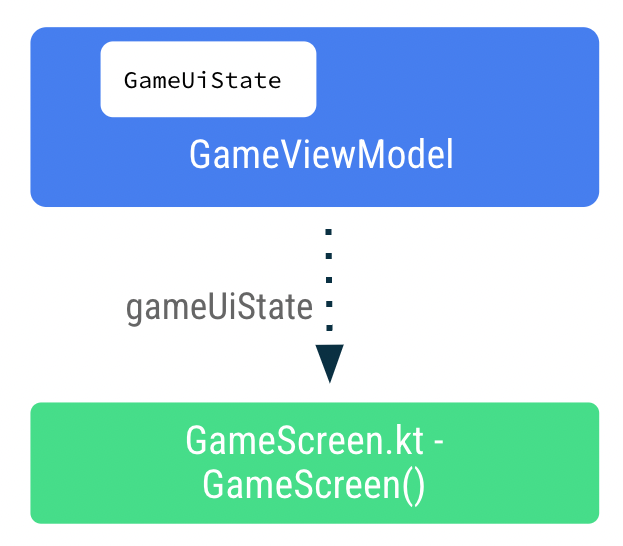
Pass the data

Pass the ViewModel instance to the UI–that is, from the GameViewModel to the GameScreen() in the GameScreen.kt file. In the GameScreen(), use the ViewModel instance to access the uiState using collectAsState().

The collectAsState() function collects values from this StateFlow and represents its latest value via State. The StateFlow.value is used as an initial value. Every time there would be a new value posted into the StateFlow, the returned State updates, causing recomposition of every State.value usage.

1. In the GameScreen function, pass a second argument of the type GameViewModel with a default value of viewModel().

|  |
| --- |
| import androidx.lifecycle.viewmodel.compose.viewModel  @Composable fun GameScreen(    gameViewModel: GameViewModel = viewModel() ) {    // ... } |



1. In the GameScreen() function, add a new variable called gameUiState. Use the by delegate and call collectAsState() on uiState.

This approach ensures that whenever there is a change in the uiState value, recomposition occurs for the composables using the gameUiState value.

|  |
| --- |
| import androidx.compose.runtime.collectAsState import androidx.compose.runtime.getValue  @Composable fun GameScreen(    // ... ) {    val gameUiState by gameViewModel.uiState.collectAsState()    // ... } |

1. Pass the gameUiState.currentScrambledWord to the GameLayout() composable. You add the argument in a later step, so ignore the error for now.

|  |
| --- |
| GameLayout(    currentScrambledWord = gameUiState.currentScrambledWord,    modifier = Modifier        .fillMaxWidth()        .wrapContentHeight()        .padding(mediumPadding) ) |

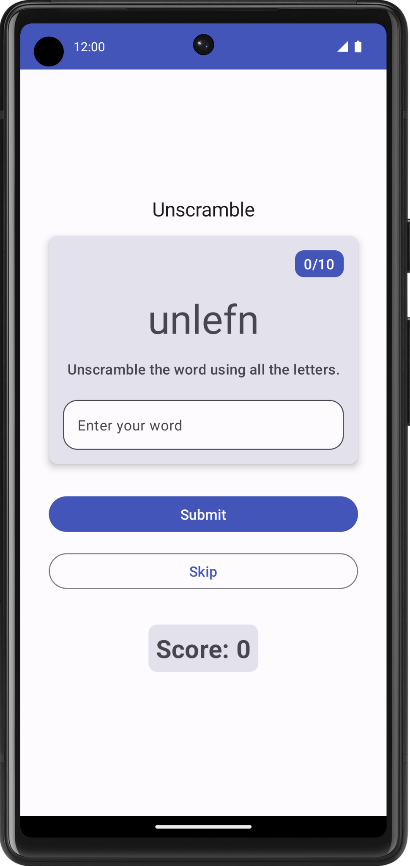
1. Add currentScrambledWord as another parameter to the GameLayout()composable function.

|  |
| --- |
| @Composable fun GameLayout(    currentScrambledWord: String,    modifier: Modifier = Modifier ) { } |

1. Update the GameLayout() composable function to display currentScrambledWord. Set the text parameter of the first text field in the column to currentScrambledWord.

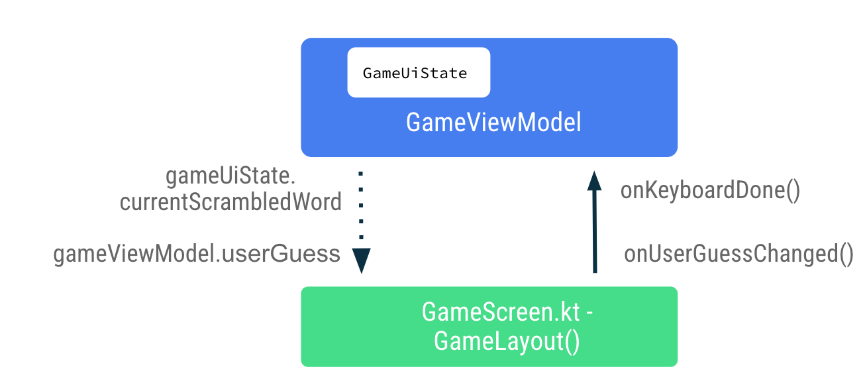
|  |
| --- |
| @Composable fun GameLayout(    // ... ) {    Column(        verticalArrangement = Arrangement.spacedBy(24.dp)    ) {        Text(            text = currentScrambledWord,            fontSize = 45.sp,            modifier = modifier.align(Alignment.CenterHorizontally)        )     //...      } } |

1. Run and build the app. You should see the scrambled word.



Display the guess word

In the GameLayout() composable, updating the user's guess word is one of event callbacks that flows up from GameScreen to the ViewModel. The data gameViewModel.userGuess will flow down from the ViewModel to the GameScreen.



1. In the GameScreen.kt file, in the GameLayout() composable, set onValueChange to onUserGuessChanged and onKeyboardDone() to onDone keyboard action. You fix the errors in the next step.

|  |
| --- |
| OutlinedTextField(    value = "",    singleLine = true,    modifier = Modifier.fillMaxWidth(),    onValueChange = onUserGuessChanged,    label = { Text(stringResource(R.string.enter\_your\_word)) },    isError = false,    keyboardOptions = KeyboardOptions.Default.copy(        imeAction = ImeAction.Done    ),    keyboardActions = KeyboardActions(        onDone = { onKeyboardDone() }    ), |

1. In the GameLayout() composable function, add two more arguments: the onUserGuessChanged lambda takes a String argument and returns nothing, and the onKeyboardDone takes nothing and returns nothing.

|  |
| --- |
| @Composable fun GameLayout(    onUserGuessChanged: (String) -> Unit,    onKeyboardDone: () -> Unit,    currentScrambledWord: String,    modifier: Modifier = Modifier,    ) { } |

1. In the GameLayout() function call, add lambda arguments for onUserGuessChanged and onKeyboardDone.

|  |
| --- |
| GameLayout(    onUserGuessChanged = { gameViewModel.updateUserGuess(it) },    onKeyboardDone = { },    currentScrambledWord = gameUiState.currentScrambledWord, ) |

You define updateUserGuess method in GameViewModel shortly.

1. In the GameViewModel.kt file, add a method called updateUserGuess() that takes a String argument, the user's guess word. Inside the function, update the userGuess with the passed in guessedWord.

|  |
| --- |
| fun updateUserGuess(guessedWord: String){      userGuess = guessedWord   } |

You add userGuess in the ViewModel next.

1. In the GameViewModel.kt file, add a var property called userGuess. Use mutableStateOf() so that Compose observes this value and sets the initial value to "".

|  |
| --- |
| import androidx.compose.runtime.mutableStateOf import androidx.compose.runtime.getValue import androidx.compose.runtime.setValue   var userGuess by mutableStateOf("")    private set |

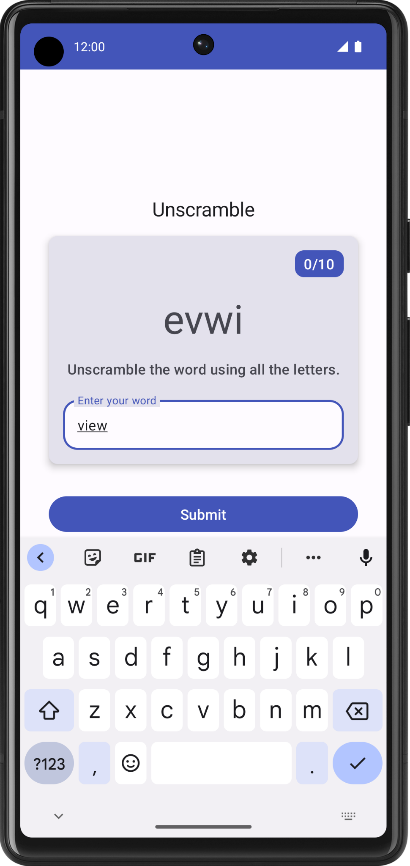
1. In the GameScreen.kt file, inside GameLayout(), add another String parameter for userGuess. Set the value parameter of the OutlinedTextField to userGuess.

|  |
| --- |
| fun GameLayout(    currentScrambledWord: String,    userGuess: String,    onUserGuessChanged: (String) -> Unit,    onKeyboardDone: () -> Unit,    modifier: Modifier = Modifier ) {    Column(        verticalArrangement = Arrangement.spacedBy(24.dp)    ) {        //...        OutlinedTextField(            value = userGuess,            //..        )    } } |

1. In the GameScreen function, update the GameLayout() function call to include userGuess parameter.

|  |
| --- |
| GameLayout(    currentScrambledWord = gameUiState.currentScrambledWord,    userGuess = gameViewModel.userGuess,    onUserGuessChanged = { gameViewModel.updateUserGuess(it) },    onKeyboardDone = { },    //... ) |

1. Build and run your app.
2. Try to guess and enter a word. The text field can display the user's guess.



7. Verify guess word and update score

In this task, you implement a method to verify the word a user guesses and then either update the game score or display an error. You will update the game state UI with the new score and the new word later.

1. In GameViewModel, add another method called checkUserGuess().
2. In the checkUserGuess() function, add an if else block to verify if the user's guess is the same as the currentWord. Reset userGuess to empty string.

|  |
| --- |
| fun checkUserGuess() {        if (userGuess.equals(currentWord, ignoreCase = true)) {    } else {    }    // Reset user guess    updateUserGuess("") } |

1. If the user's guess is wrong, set isGuessedWordWrong to true. MutableStateFlow<T>.update() updates the MutableStateFlow.value using the specified value.

|  |
| --- |
| import kotlinx.coroutines.flow.update     if (userGuess.equals(currentWord, ignoreCase = true)) {    } else {        // User's guess is wrong, show an error        \_uiState.update { currentState ->            currentState.copy(isGuessedWordWrong = true)        }    } |

**Note on copy() method**: Use the copy() function to copy an object, allowing you to alter some of its properties while keeping the rest unchanged.

Example:

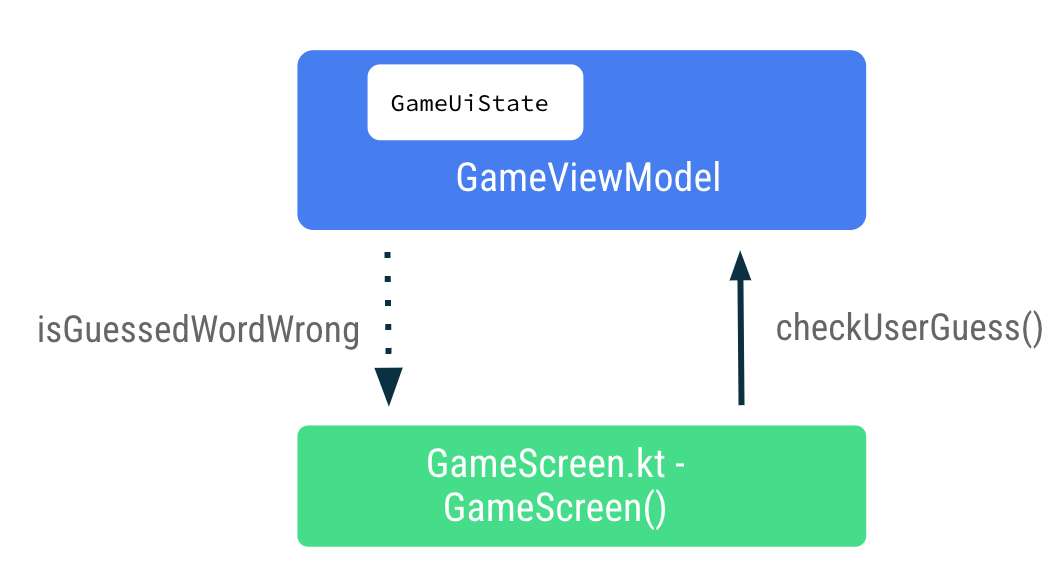
val jack = User(name = "Jack", age = 1)

val olderJack = jack.copy(age = 2)

1. In the GameUiState class, add a Boolean called isGuessedWordWrong and initialize it to false.

|  |
| --- |
| data class GameUiState(    val currentScrambledWord: String = "",    val isGuessedWordWrong: Boolean = false, ) |

Next, you pass the event callback checkUserGuess() up from GameScreen to ViewModel when the user clicks the **Submit** button or the done key in the keyboard. Pass the data, gameUiState.isGuessedWordWrong down from the ViewModel to the GameScreen to set the error in the text field.



1. In the GameScreen.kt file, at the end of the GameScreen() composable function, call gameViewModel.checkUserGuess() inside the onClick lambda expression of the **Submit** button.

|  |
| --- |
| Button(    modifier = modifier        .fillMaxWidth()        .weight(1f)        .padding(start = 8.dp),    onClick = { gameViewModel.checkUserGuess() } ) {    Text(stringResource(R.string.submit)) } |

1. In the GameScreen() composable function, update the GameLayout() function call to pass gameViewModel.checkUserGuess() in the onKeyboardDone lambda expression.

|  |
| --- |
| GameLayout(    currentScrambledWord = gameUiState.currentScrambledWord,    userGuess = gameViewModel.userGuess,    onUserGuessChanged = { gameViewModel.updateUserGuess(it) },    onKeyboardDone = { gameViewModel.checkUserGuess() } ) |

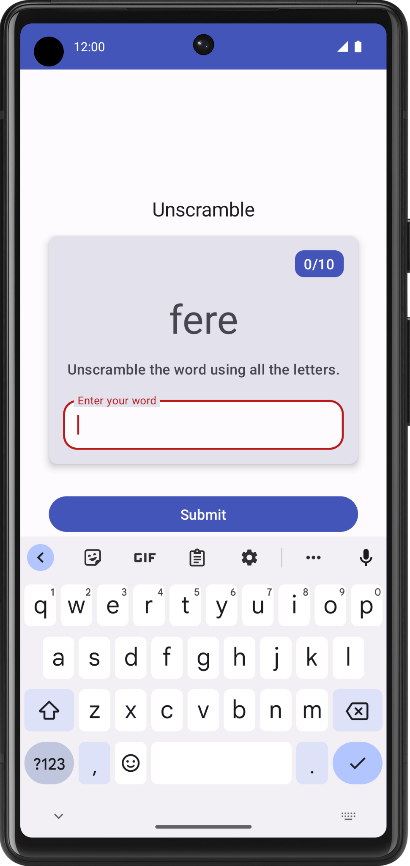
1. In the GameLayout() composable function, add a function parameter for the Boolean, isGuessWrong. Set the isError parameter of the OutlinedTextField to isGuessWrong to display the error in the text field if the user's guess is wrong.

|  |
| --- |
| fun GameLayout(    currentScrambledWord: String,    isGuessWrong: Boolean,    userGuess: String,    onUserGuessChanged: (String) -> Unit,    onKeyboardDone: () -> Unit,    modifier: Modifier = Modifier ) {    Column(        // ,...        OutlinedTextField(            // ...            isError = isGuessWrong,            keyboardOptions = KeyboardOptions.Default.copy(                imeAction = ImeAction.Done            ),            keyboardActions = KeyboardActions(                onDone = { onKeyboardDone() }            ),        ) } } |

1. In the GameScreen() composable function, update the GameLayout() function call to pass isGuessWrong.

|  |
| --- |
| GameLayout(    currentScrambledWord = gameUiState.currentScrambledWord,    userGuess = gameViewModel.userGuess,    onUserGuessChanged = { gameViewModel.updateUserGuess(it) },    onKeyboardDone = { gameViewModel.checkUserGuess() },    isGuessWrong = gameUiState.isGuessedWordWrong,    // ... ) |

1. Build and run your app.
2. Enter a wrong guess and click **Submit**. Observe that the text field turns red, indicating the error.



Notice the text field label still reads "Enter your word". To make it user friendly, you need to add some error text to indicate the word is wrong.

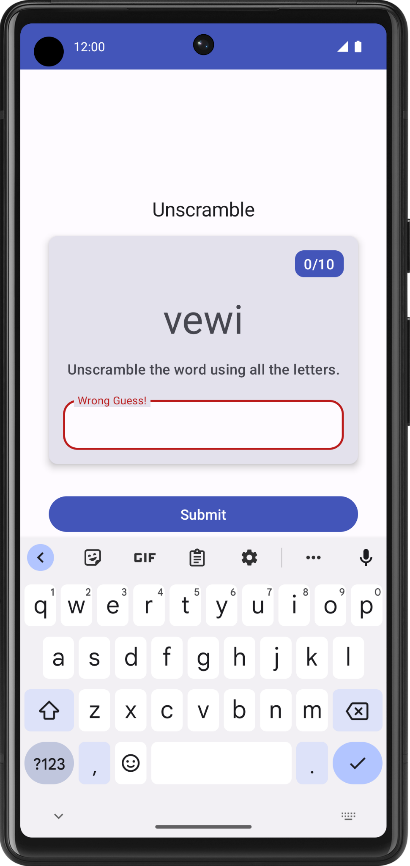
1. In the GameScreen.kt file, in the GameLayout() composable, update the label parameter of the text field depending on the isGuessWrong as follows:

|  |
| --- |
| OutlinedTextField(    // ...    label = {        if (isGuessWrong) {            Text(stringResource(R.string.wrong\_guess))        } else {            Text(stringResource(R.string.enter\_your\_word))        }    },    // ... ) |

1. In the strings.xml file, add a string to the error label.

|  |
| --- |
| <string name="wrong\_guess">Wrong Guess!</string> |

1. Build and run your app again.
2. Enter a wrong guess and click **Submit**. Notice the error label.



8. Update score and word count

In this task, you update the score and the word count as the user plays the game. The score must be part of \_uiState.

1. In GameUiState, add a score variable and initialize it to zero.

|  |
| --- |
| data class GameUiState(    val currentScrambledWord: String = "",    val isGuessedWordWrong: Boolean = false,    val score: Int = 0 ) |

1. To update the score value, in GameViewModel, in checkUserGuess() function, inside the if condition for when the user's guess is correct, increase the score value.

|  |
| --- |
| import com.example.unscramble.data.SCORE\_INCREASE  fun checkUserGuess() {    if (userGuess.equals(currentWord, ignoreCase = true)) {        // User's guess is correct, increase the score        val updatedScore = \_uiState.value.score.plus(SCORE\_INCREASE)    } else {        //...    } } |

1. In GameViewModel, add another method called updateGameState to update the score, increment the current word count and pick a new word from the WordsData.kt file. Add an Int named updatedScore as the parameter. Update the game state UI variables as follows:

|  |
| --- |
| private fun updateGameState(updatedScore: Int) {    \_uiState.update { currentState ->        currentState.copy(            isGuessedWordWrong = false,            currentScrambledWord = pickRandomWordAndShuffle(),            score = updatedScore        )    } } |

1. In the checkUserGuess() function, if the user's guess is correct, make a call to updateGameState with the updated score to prepare the game for the next round.

|  |
| --- |
| fun checkUserGuess() {    if (userGuess.equals(currentWord, ignoreCase = true)) {        // User's guess is correct, increase the score        // and call updateGameState() to prepare the game for next round        val updatedScore = \_uiState.value.score.plus(SCORE\_INCREASE)        updateGameState(updatedScore)    } else {        //...    } } |

The completed checkUserGuess() should look like the following:

|  |
| --- |
| fun checkUserGuess() {    if (userGuess.equals(currentWord, ignoreCase = true)) {        // User's guess is correct, increase the score        // and call updateGameState() to prepare the game for next round        val updatedScore = \_uiState.value.score.plus(SCORE\_INCREASE)        updateGameState(updatedScore)    } else {        // User's guess is wrong, show an error        \_uiState.update { currentState ->            currentState.copy(isGuessedWordWrong = true)        }    }    // Reset user guess    updateUserGuess("") } |

Next, similar to the updates for the score, you need to update the word count.

1. Add another variable for the count in the GameUiState. Call it currentWordCount and initialize it to 1.

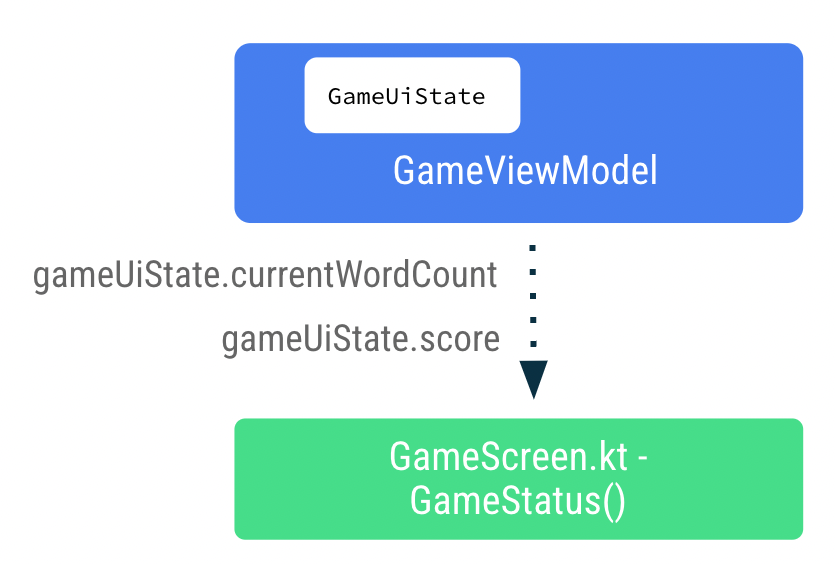
|  |
| --- |
| data class GameUiState(    val currentScrambledWord: String = "",    val currentWordCount: Int = 1,    val score: Int = 0,    val isGuessedWordWrong: Boolean = false, ) |

1. In the GameViewModel.kt file, in the updateGameState() function, increase the word count as shown below. The updateGameState() function is called to prepare the game for the next round.

|  |
| --- |
| private fun updateGameState(updatedScore: Int) {    \_uiState.update { currentState ->        currentState.copy(            //...            currentWordCount = currentState.currentWordCount.inc(),            )    } } |

Pass score and word count

Complete the following steps to pass the score and word count data down from ViewModel to the GameScreen.



1. In the GameScreen.kt file, in the GameLayout() composable function, add word count as argument and pass the wordCount format arguments to the text element.

|  |
| --- |
| fun GameLayout(    onUserGuessChanged: (String) -> Unit,    onKeyboardDone: () -> Unit,    wordCount: Int,    //... ) {    //...     Card(        //...    ) {        Column(            // ...        ) {            Text(                //..                text = stringResource(R.string.word\_count, wordCount),                style = typography.titleMedium,                color = colorScheme.onPrimary            )   // ...  } |

1. Update the GameLayout() function call to include the word count.

|  |
| --- |
| GameLayout(    userGuess = gameViewModel.userGuess,    wordCount = gameUiState.currentWordCount,    //... ) |

1. In the GameScreen() composable function, update the GameStatus() function call to include score parameters. Pass the score from the gameUiState.

|  |
| --- |
| GameStatus(score = gameUiState.score, modifier = Modifier.padding(20.dp)) |

1. Build and run the app.
2. Enter the guess word and click **Submit**. Notice the score and word count update.
3. Click **Skip**, and notice nothing happens.

To implement the skip functionality you need to pass the skip event callback to the GameViewModel.

1. In the GameScreen.kt file, in the GameScreen() composable function, make a call to gameViewModel.skipWord() in the onClick lambda expression.

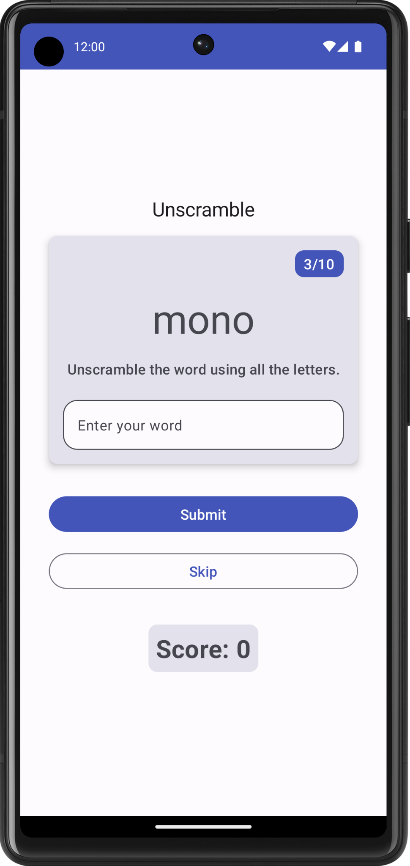
Android Studio displays an error because you have not implemented the function yet.You fix this error in the next step by adding the skipWord() method. When the user skips a word, you need to update the game variables and prepare the game for the next round.

|  |
| --- |
| OutlinedButton(    onClick = { gameViewModel.skipWord() },    modifier = Modifier.fillMaxWidth() ) {    //... } |

1. In GameViewModel, add the method skipWord().
2. Inside the skipWord() function, make a call to updateGameState(), passing the score and reset the user guess.

|  |
| --- |
| fun skipWord() {    updateGameState(\_uiState.value.score)    // Reset user guess    updateUserGuess("") } |

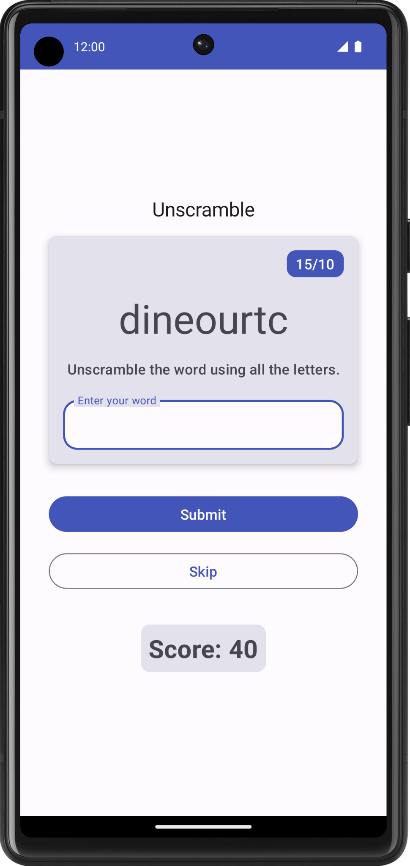
1. Run your app and play the game. You should now be able to skip words.



You can still play the game beyond 10 words. In your next task, you'll handle the last round of the game.

9. Handle last round of game

In the current implementation, users can skip or play beyond 10 words. In this task, you add logic to end the game.



To implement game-end logic, you first need to check if the user reaches the maximum number of words.

1. In GameViewModel add an if-else block and move the existing function body inside the else block.
2. Add an if condition to check the usedWords size is equal to MAX\_NO\_OF\_WORDS.

|  |
| --- |
| import com.example.android.unscramble.data.MAX\_NO\_OF\_WORDS   private fun updateGameState(updatedScore: Int) {    if (usedWords.size == MAX\_NO\_OF\_WORDS){        //Last round in the game    } else{        // Normal round in the game        \_uiState.update { currentState ->            currentState.copy(                isGuessedWordWrong = false,                currentScrambledWord = pickRandomWordAndShuffle(),                currentWordCount = currentState.currentWordCount.inc(),                score = updatedScore            )        }    } } |

1. Inside the if block, add the Boolean flag isGameOver and set the flag to true to indicate the end of the game.
2. Update the score and reset isGuessedWordWrong inside the if block. The following code shows how your function should look:

|  |
| --- |
| private fun updateGameState(updatedScore: Int) {    if (usedWords.size == MAX\_NO\_OF\_WORDS){        //Last round in the game, update isGameOver to true, don't pick a new word        \_uiState.update { currentState ->            currentState.copy(                isGuessedWordWrong = false,                score = updatedScore,                isGameOver = true            )        }    } else{        // Normal round in the game        \_uiState.update { currentState ->            currentState.copy(                isGuessedWordWrong = false,                currentScrambledWord = pickRandomWordAndShuffle(),                currentWordCount = currentState.currentWordCount.inc(),                score = updatedScore            )        }    } } |

1. In GameUiState, add the Boolean variable isGameOver and set it to false.

|  |
| --- |
| data class GameUiState(    val currentScrambledWord: String = "",    val currentWordCount: Int = 1,    val score: Int = 0,    val isGuessedWordWrong: Boolean = false,    val isGameOver: Boolean = false ) |

1. Run your app and play the game. You cannot play beyond 10 words.



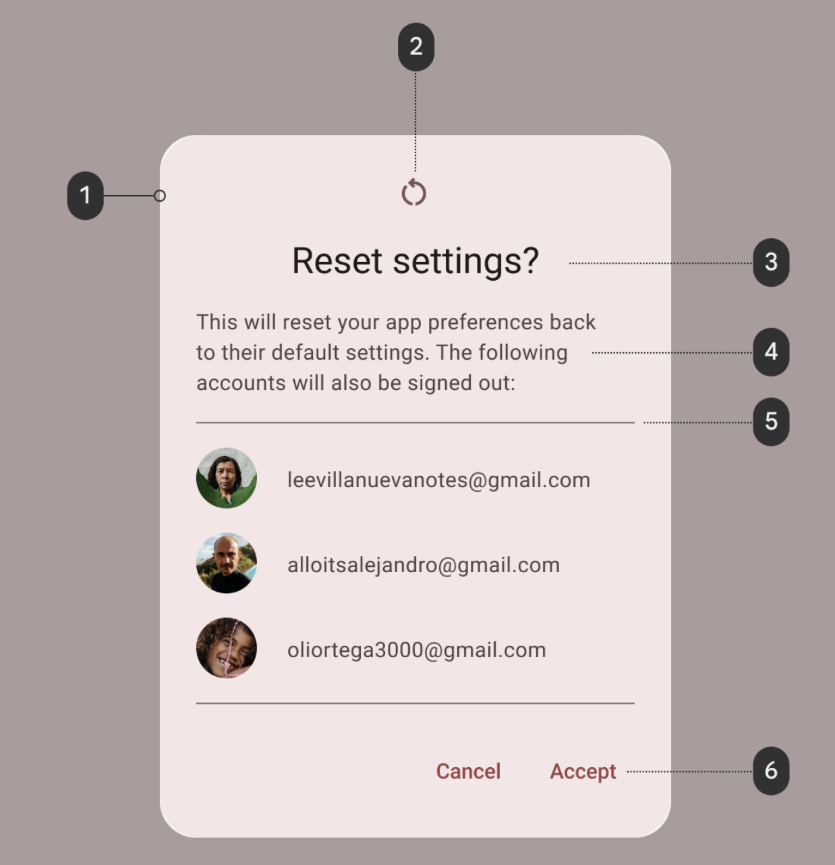
When the game is over, it would be nice to let the user know and ask if they would like to play again. You will implement this feature in your next task.

Display game end dialog

In this task, you pass isGameOver data down to GameScreen from the ViewModel and use it to display an alert dialog with options to end or restart the game.

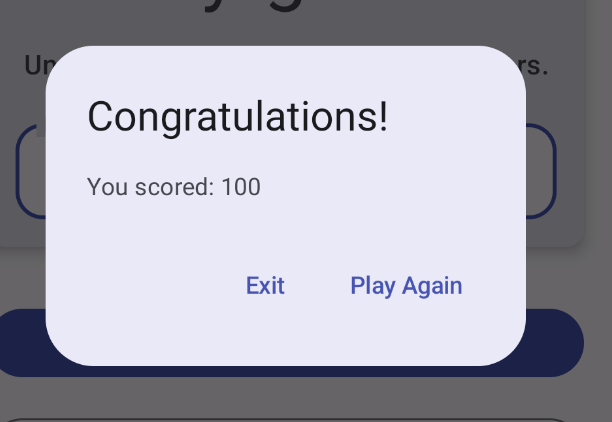
A dialog is a small window that prompts the user to make a decision or enter additional information. Normally, a dialog does not fill the entire screen, and it requires users to take an action before they can proceed. Android provides different types of dialogs. In this codelab, you learn about Alert Dialogs.

Anatomy of alert dialog



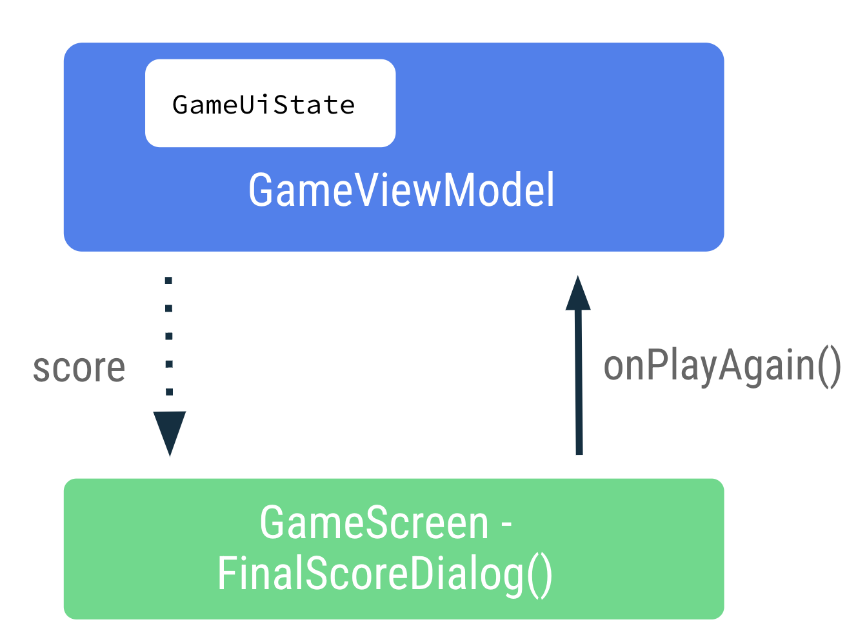
1. Container
2. Icon (optional)
3. Headline (optional)
4. Supporting text
5. Divider (optional)
6. Actions

The GameScreen.kt file in the starter code already provides a function that displays an alert dialog with options to exit or restart the game.



|  |
| --- |
| @Composable private fun FinalScoreDialog(    onPlayAgain: () -> Unit,    modifier: Modifier = Modifier ) {    val activity = (LocalContext.current as Activity)     AlertDialog(        onDismissRequest = {            // Dismiss the dialog when the user clicks outside the dialog or on the back            // button. If you want to disable that functionality, simply use an empty            // onDismissRequest.        },        title = { Text(stringResource(R.string.congratulations)) },        text = { Text(stringResource(R.string.you\_scored, 0)) },        modifier = modifier,        dismissButton = {            TextButton(                onClick = {                    activity.finish()                }            ) {                Text(text = stringResource(R.string.exit))            }        },        confirmButton = {            TextButton(                onClick = {                    onPlayAgain()                }            ) {                Text(text = stringResource(R.string.play\_again))            }        }    ) } |

In this function, title and text parameters display the headline and supporting text in the alert dialog. The dismissButton and confirmButton are the text buttons. In dismissButton parameter, you display the text **Exit** and terminate the app by finishing the activity. In confirmButton parameter, you restart the game and display the text **Play Again**.



1. In the GameScreen.kt file, in the FinalScoreDialog() function, notice the parameter for the score to display the game score in the alert dialog.

|  |
| --- |
| @Composable private fun FinalScoreDialog(    score: Int,    onPlayAgain: () -> Unit,    modifier: Modifier = Modifier ) { |

1. In the FinalScoreDialog() function, notice the usage of the text parameter lambda expression to use score as format argument to the dialog text.

|  |
| --- |
| text = { Text(stringResource(R.string.you\_scored, score)) } |

1. In the GameScreen.kt file, at the end of the GameScreen() composable function, after the Column block, add an if condition to check gameUiState.isGameOver.
2. In the if block, display the alert dialog. Make a call to FinalScoreDialog() passing in the score and gameViewModel.resetGame() for the onPlayAgain event callback.

|  |
| --- |
| if (gameUiState.isGameOver) {    FinalScoreDialog(        score = gameUiState.score,        onPlayAgain = { gameViewModel.resetGame() }    ) } |

The resetGame() is an event callback that is passed up from the GameScreen to the ViewModel.

1. In the GameViewModel.kt file, recall the resetGame() function, initializes the \_uiState, and picks a new word.

|  |
| --- |
| fun resetGame() {    usedWords.clear()    \_uiState.value = GameUiState(currentScrambledWord = pickRandomWordAndShuffle()) } |

1. Build and run your app.
2. Play the game until the end, and observe the alert dialog with options to **Exit** the game or **Play Again**. Try the options displayed on the alert dialog.



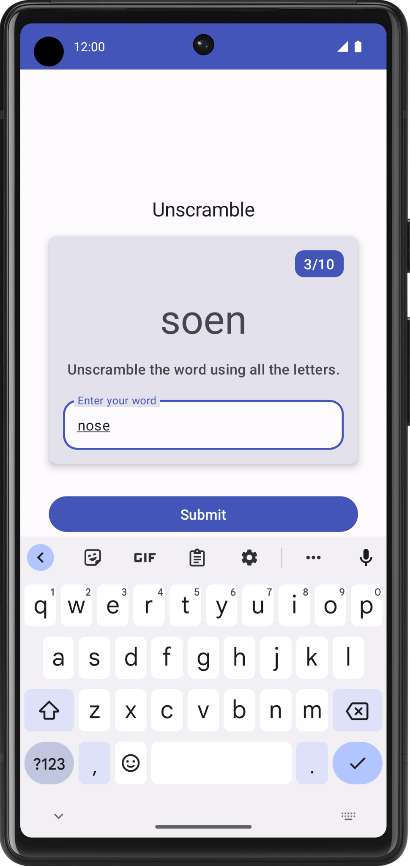
10. State in device rotation

In previous codelabs, you learned about configuration changes in Android. When a configuration change occurs, Android restarts the activity from scratch, running all the lifecycle startup callbacks.

The ViewModel stores the app-related data that isn't destroyed when the Android framework destroys and recreates activity. ViewModel objects are automatically retained and they are not destroyed like the activity instance during configuration change. The data they hold is immediately available after the recomposition.

In this task, you check if the app retains the state UI during a configuration change.

1. Run the app and play some words. Change the configuration of the device from portrait to landscape, or vice versa.
2. Observe that the data saved in the ViewModel's state UI is retained during the configuration change.





11. Get the solution code

To download the code for the finished codelab, you can use these git commands:

|  |
| --- |
| $ git clone https://github.com/google-developer-training/basic-android-kotlin-compose-training-unscramble.git  $ cd basic-android-kotlin-compose-training-unscramble  $ git checkout viewmodel |

Alternatively, you can download the repository as a zip file, unzip it, and open it in Android Studio.

<https://github.com/google-developer-training/basic-android-kotlin-compose-training-unscramble/archive/refs/heads/viewmodel.zip>

**Note**: The solution code is in the viewmodel branch of the downloaded repository.

If you want to see the solution code for this codelab, view it on <https://github.com/google-developer-training/basic-android-kotlin-compose-training-unscramble/tree/viewmodel>.

12. Conclusion

Congratulations! You have completed the codelab. Now you understand how the Android app architecture guidelines recommend separating classes that have different responsibilities and driving the UI from a model.

Don't forget to share your work on social media with #AndroidBasics!

Learn more

|  |  |
| --- | --- |
| Guide to app architecture | <https://developer.android.com/topic/architecture> |
| UI layer | <https://developer.android.com/topic/architecture/ui-layer> |
| Manage state with Unidirectional Data Flow | <https://developer.android.com/topic/architecture/ui-layer#udf> |
| Learning Pathway: Modern Android App Architecture | <https://developer.android.com/courses/pathways/android-architecture> |