Save preferences locally with DataStore

1. Before you begin

Introduction

In this unit, you have learned how to use SQL and Room to save data locally on a device. SQL and Room are powerful tools. However, in cases where you don't need to store relational data, DataStore can provide a simple solution. The DataStore Jetpack Component is a great way to store small and simple data sets with low overhead. DataStore has two different implementations, Preferences DataStore and Proto DataStore.

* Preferences DataStore stores key-value pairs. The values can be Kotlin's basic data types, such as String, Boolean, and Integer. It does not store complex datasets. It does not require a predefined schema. The primary use case of the Preferences Datastore is to store user preferences on their device.
* Proto DataStore stores custom data types. It requires a predefined schema that maps proto definitions with object structures.

Only Preferences DataStore is covered in this codelab, but you can read more about Proto DataStore in the DataStore documentation.

Preferences DataStore is a great way to store user-controlled settings, and in this codelab, you learn how to implement DataStore (<https://developer.android.com/topic/libraries/architecture/datastore>) to do exactly that!

Prerequisites:

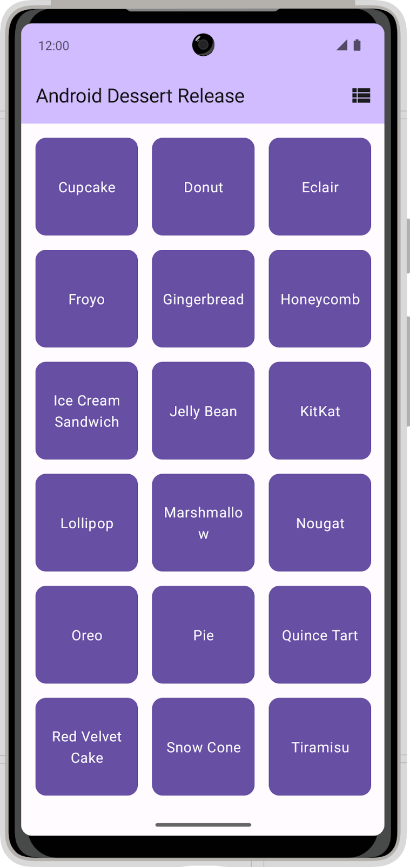
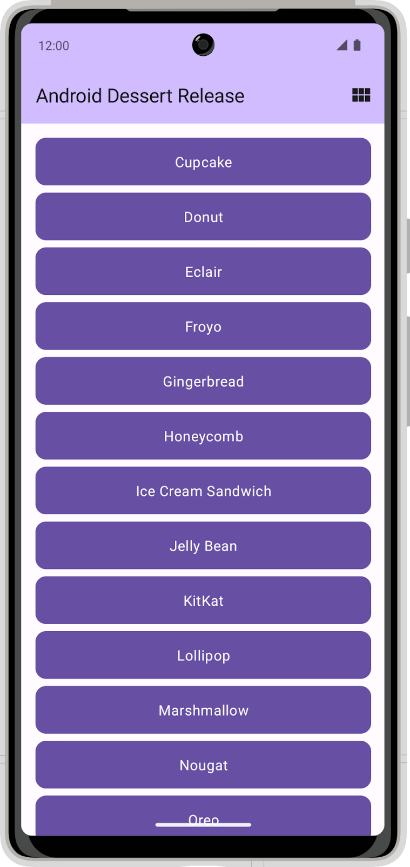
* Complete the Android Basics with Compose coursework through the Read and Update Data with Room codelab.

What you'll need

* A computer with internet access and Android Studio
* A device or emulator
* The starter code for the Dessert Release app

What you'll build

The Dessert Release app shows a list of Android releases. The icon in the app bar toggles the layout between a grid view and a list view.



In its current state, the app does not persist the layout selection. When you close the app, your layout selection does not save and the setting returns to the default selection. In this codelab, you add DataStore to the Dessert Release app and use it to store a layout selection preference.

2. Download the starter code

Click the following link to download all the code for this codelab:

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

Or if you prefer, you can clone the Dessert Release code from GitHub:

**Starter code URL**:

[https://github.com/google-developer-training/basic-android-kotlin-compose-training-dessert-release](https://github.com/google-developer-training/basic-android-kotlin-compose-training-dessert-release/tree/starter)

**Branch name with starter code**: starter

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

1. In Android Studio, open the basic-android-kotlin-compose-training-dessert-release folder.
2. Open the Dessert Release app code in Android Studio.

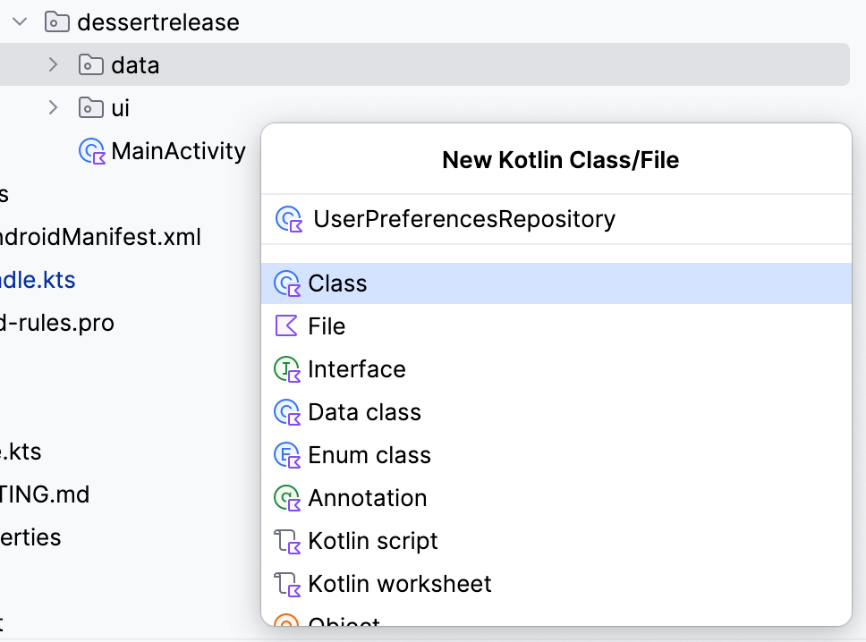
3. Set up dependencies

Add the following to dependencies in the app/build.gradle.kts file:

|  |
| --- |
| implementation("androidx.datastore:datastore-preferences:1.0.0") |

4. Implement the user preferences repository

1. In the data package, create a new class called UserPreferencesRepository.



1. In the UserPreferencesRepository constructor, define a private value property to represent a DataStore object instance with a Preferences type.

|  |
| --- |
| class UserPreferencesRepository(     private val dataStore: DataStore<Preferences> ){ } |

**Note**: Make sure to use the androidx.datastore.preferences.core.Preferences import for the Preferences class.

DataStore stores key-value pairs. To access a value you must define a key.

1. Create a companion object inside the UserPreferencesRepository class.
2. Use the booleanPreferencesKey() function to define a key and pass it the name is\_linear\_layout. Similar to SQL table names, the key needs to use an underscore format. This key is used to access a boolean value indicating whether the linear layout should be shown.

|  |
| --- |
| class UserPreferencesRepository(     private val dataStore: DataStore<Preferences> ){     private companion object {         val IS\_LINEAR\_LAYOUT = booleanPreferencesKey("is\_linear\_layout")     }     ... } |

Write to the DataStore

You create and modify the values within a DataStore by passing a lambda to the edit() method. The lambda is passed an instance of MutablePreferences, which you can use to update values in the DataStore. All the updates inside this lambda are executed as a single transaction. Put another way, the update is *atomic* — it happens all at one time. This type of update prevents a situation in which some values update but others do not.

1. Create a suspend function and call it saveLayoutPreference().
2. In the saveLayoutPreference() function, call the edit() method on the dataStore object.

|  |
| --- |
| suspend fun saveLayoutPreference(isLinearLayout: Boolean) {     dataStore.edit {      } } |

1. To make your code more readable, define a name for the MutablePreferences provided in the lambda body. Use that property to set a value with the key you defined and the boolean passed to the saveLayoutPreference() function.

|  |
| --- |
| suspend fun saveLayoutPreference(isLinearLayout: Boolean) {     dataStore.edit { preferences ->         preferences[IS\_LINEAR\_LAYOUT] = isLinearLayout     } } |

**Note**: The value does not exist in DataStore until this function is called and the value is set. By setting up the key-value pair in the edit() method, the value is defined and initialized until the app's cache or data is cleared.

Read from the DataStore

Now that you have created a way to write isLinearLayout into dataStore, take the following steps to read it:

1. Create a property in UserPreferencesRepository of type Flow<Boolean> called isLinearLayout.

|  |
| --- |
| val isLinearLayout: Flow<Boolean> = |

1. You can use the DataStore.data property to expose DataStore values. Set isLinearLayout to the data property of the DataStore object.

|  |
| --- |
| val isLinearLayout: Flow<Boolean> = dataStore.data |

**Note**: This code does not compile and the dataStore.data instruction is underlined in red. This outcome is expected, as the implementation is not yet complete.

The data property is a Flow of Preferences objects. The Preferences object contains all the key-value pairs in the DataStore. Each time the data in the DataStore is updated, a new Preferences object is emitted into the Flow.

1. Use the map function to convert the Flow<Preferences> into a Flow<Boolean>.

This function accepts a lambda with the current Preferences object as a parameter. You can specify the key you previously defined to obtain the layout preference. Bear in mind that the value might not exist if saveLayoutPreference hasn't been called yet, so you must also supply a default value.

1. Specify true to default to the linear layout view.

**Note**: Remember that until the preference is defined and initialized, it does not exist in the DataStore. That is why you must programmatically confirm that the preference exists and provide a default value if it does not.

|  |
| --- |
| val isLinearLayout: Flow<Boolean> = dataStore.data.map { preferences ->     preferences[IS\_LINEAR\_LAYOUT] ?: true } |

Exception handling

Any time you interact with the file system on a device, it's possible that something can fail. For example, a file might not exist, or the disk could be full or unmounted. As DataStore reads and writes data from files, IOExceptions can occur when accessing the DataStore. You use the catch{} operator to catch exceptions and handle these failures.

1. In the companion object, implement an immutable TAG string property to use for logging.

|  |
| --- |
| private companion object {     val IS\_LINEAR\_LAYOUT = booleanPreferencesKey("is\_linear\_layout")     const val TAG = "UserPreferencesRepo" } |

1. Preferences DataStore throws an IOException when an error is encountered while reading data. In the isLinearLayout initialization block, before map(), use the catch{} operator to catch the IOException.

|  |
| --- |
| val isLinearLayout: Flow<Boolean> = dataStore.data     .catch {}     .map { preferences ->         preferences[IS\_LINEAR\_LAYOUT] ?: true     } |

1. In the catch block, if there is an IOexception, log the error and emit emptyPreferences(). If a different type of exception is thrown, prefer re-throwing that exception. By emitting emptyPreferences() if there is an error, the map function can still map to the default value.

|  |
| --- |
| val isLinearLayout: Flow<Boolean> = dataStore.data     .catch {         if(it is IOException) {             Log.e(TAG, "Error reading preferences.", it)             emit(emptyPreferences())         } else {             throw it         }     }     .map { preferences ->         preferences[IS\_LINEAR\_LAYOUT] ?: true     } |

5. Initialize the DataStore

In this codelab, you must handle the dependency injection manually. Therefore, you must manually provide the UserPreferencesRepository class with a Preferences DataStore. Follow these steps to inject DataStore into the UserPreferencesRepository.

1. Find the dessertrelease package.
2. Within this directory, create a new class called DessertReleaseApplication and implement the Application class. This is the container for your DataStore.

|  |
| --- |
| class DessertReleaseApplication: Application() { } |

1. Inside of the DessertReleaseApplication.kt file, but outside the DessertReleaseApplication class, declare a private const val called LAYOUT\_PREFERENCE\_NAME.
2. Assign the LAYOUT\_PREFERENCE\_NAME variable the string value layout\_preferences, which you can then use as the name of the Preferences Datastore that you instantiate in the next step.

|  |
| --- |
| private const val LAYOUT\_PREFERENCE\_NAME = "layout\_preferences" |

1. Still outside the DessertReleaseApplication class body but in the DessertReleaseApplication.kt file, create a private value property of type DataStore<Preferences> called Context.dataStore using the preferencesDataStore delegate. Pass LAYOUT\_PREFERENCE\_NAME for the name parameter of the preferencesDataStore delegate.

|  |
| --- |
| private const val LAYOUT\_PREFERENCE\_NAME = "layout\_preferences" private val Context.dataStore: DataStore<Preferences> by preferencesDataStore(     name = LAYOUT\_PREFERENCE\_NAME ) |

1. Inside the DessertReleaseApplication class body, create a lateinit var instance of the UserPreferencesRepository.

|  |
| --- |
| private const val LAYOUT\_PREFERENCE\_NAME = "layout\_preferences" private val Context.dataStore: DataStore<Preferences> by preferencesDataStore(     name = LAYOUT\_PREFERENCE\_NAME )  class DessertReleaseApplication: Application() {     lateinit var userPreferencesRepository: UserPreferencesRepository } |

1. Override the onCreate() method.

|  |
| --- |
| private const val LAYOUT\_PREFERENCE\_NAME = "layout\_preferences" private val Context.dataStore: DataStore<Preferences> by preferencesDataStore(     name = LAYOUT\_PREFERENCE\_NAME )  class DessertReleaseApplication: Application() {     lateinit var userPreferencesRepository: UserPreferencesRepository      override fun onCreate() {         super.onCreate()     } } |

1. Inside the onCreate() method, initialize userPreferencesRepository by constructing a UserPreferencesRepository with dataStore as its parameter.

|  |
| --- |
| private const val LAYOUT\_PREFERENCE\_NAME = "layout\_preferences" private val Context.dataStore: DataStore<Preferences> by preferencesDataStore(     name = LAYOUT\_PREFERENCE\_NAME )  class DessertReleaseApplication: Application() {     lateinit var userPreferencesRepository: UserPreferencesRepository      override fun onCreate() {         super.onCreate()         userPreferencesRepository = UserPreferencesRepository(dataStore)     } } |

1. Add the following line inside the <application> tag in the AndroidManifest.xml file.

|  |
| --- |
| <application     android:name=".DessertReleaseApplication"     ... </application> |

This approach defines DessertReleaseApplication class as the entry point of the app. The purpose of this code is to initialize the dependencies defined in the DessertReleaseApplication class before launching the MainActivity.

6. Use the UserPreferencesRepository

Provide the repository to the ViewModel

Now that the UserPreferencesRepository is available through dependency injection, you can use it in DessertReleaseViewModel.

1. In the DessertReleaseViewModel, create a UserPreferencesRepository property as a constructor parameter.

|  |
| --- |
| class DessertReleaseViewModel(     private val userPreferencesRepository: UserPreferencesRepository ) : ViewModel() {     ... } |

1. Within the ViewModel's companion object, in the viewModelFactory initializer block, obtain an instance of the DessertReleaseApplication using the following code.

|  |
| --- |
| companion object {         val Factory: ViewModelProvider.Factory = viewModelFactory {             initializer {                 val application = (this[APPLICATION\_KEY] as DessertReleaseApplication)                 ...             }         }     } } |

1. Create an instance of the DessertReleaseViewModel and pass the userPreferencesRepository.

|  |
| --- |
| companion object {         val Factory: ViewModelProvider.Factory = viewModelFactory {             initializer {                 val application = (this[APPLICATION\_KEY] as DessertReleaseApplication)                 DessertReleaseViewModel(application.userPreferencesRepository)             }         }     } } |

The UserPreferencesRepository is now accessible by the ViewModel. The next steps are to use the read and write capabilities of the UserPreferencesRepository that you implemented previously.

Store the layout preference

1. Edit the selectLayout() function in the DessertReleaseViewModel to access the preferences repository and update the layout preference.
2. Recall that writing to the DataStore is done asynchronously with a suspend function. Start a new Coroutine to call the preference repository's saveLayoutPreference() function.

|  |
| --- |
| fun selectLayout(isLinearLayout: Boolean) {     viewModelScope.launch {         userPreferencesRepository.saveLayoutPreference(isLinearLayout)     } } |

Read the layout preference

In this section, you refactor the existing uiState: StateFlow in the ViewModel to reflect the isLinearLayout: Flow from the repository.

1. Delete the code that initializes the uiState property to MutableStateFlow(DessertReleaseUiState).

|  |
| --- |
| val uiState: StateFlow<DessertReleaseUiState> = |

The linear layout preference from the repository has two possible values, true or false, in the form of a Flow<Boolean>. This value must map to a UI state.

1. Set the StateFlow to the result of the map() collection transformation called on the isLinearLayout Flow.

|  |
| --- |
| val uiState: StateFlow<DessertReleaseUiState> =     userPreferencesRepository.isLinearLayout.map { isLinearLayout -> } |

1. Return an instance of the DessertReleaseUiState data class, passing the isLinearLayout Boolean. The screen uses this UI state to determine the correct strings and icons to display.

|  |
| --- |
| val uiState: StateFlow<DessertReleaseUiState> =     userPreferencesRepository.isLinearLayout.map { isLinearLayout ->         DessertReleaseUiState(isLinearLayout)     } |

UserPreferencesRepository.isLinearLayout is a Flow which is *cold* (<https://developer.android.com/kotlin/flow/stateflow-and-sharedflow#sharein>). However, for providing state to the UI, it's better to use a *hot flow*, like StateFlow, so that the state is always available immediately to the UI.

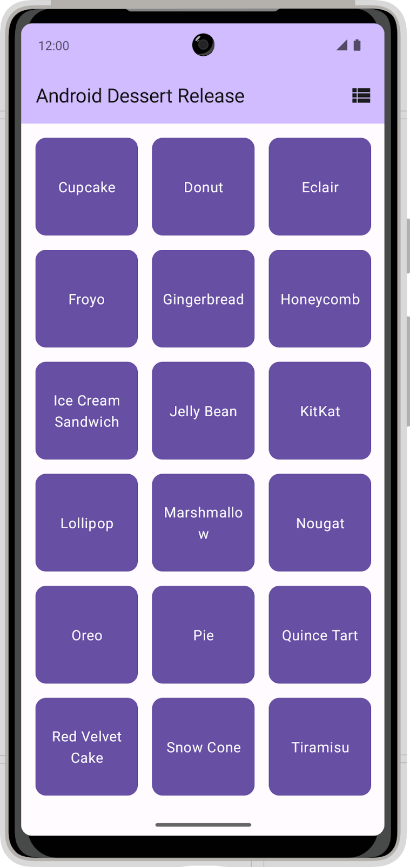
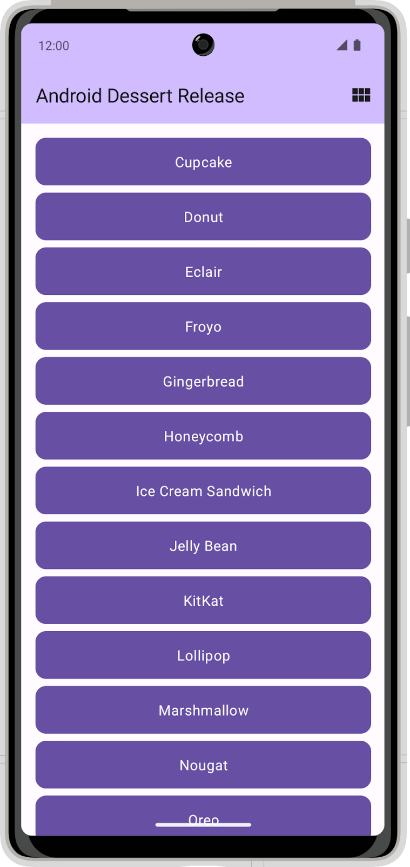
1. Use the stateIn() function to convert a Flow to a StateFlow.
2. The stateIn() function accepts three parameters: scope, started, and initialValue. Pass in viewModelScope, SharingStarted.WhileSubscribed(5\_000) and DessertReleaseUiState() for these parameters, respectively.

|  |
| --- |
| val uiState: StateFlow<DessertReleaseUiState> =     userPreferencesRepository.isLinearLayout.map { isLinearLayout ->         DessertReleaseUiState(isLinearLayout)     } .stateIn(         scope = viewModelScope,         started = SharingStarted.WhileSubscribed(5\_000),         initialValue = DessertReleaseUiState()     ) |

**Note**: Please read Migrating from LiveData to Kotlin's Flow (<https://medium.com/androiddevelopers/migrating-from-livedata-to-kotlins-flow-379292f419fb>) to learn more about the started parameter and why SharingStarted.WhileSubscribed(5\_000) is passed to it.

1. Launch the app. Notice that you can click on the toggle icon to toggle between a grid layout and a linear layout.

**Note**: Try toggling the layout and closing the app. Reopen the app and notice that your layout preference was saved.



Congratulations! You successfully added Preferences DataStore to your app to save the user's layout preference.

7. 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-dessert-release.git  $ cd basic-android-kotlin-compose-training-dessert-release  $ git checkout main |

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-dessert-release/archive/refs/heads/main.zip>

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

If you want to see the solution code, view it on GitHub.

<https://github.com/google-developer-training/basic-android-kotlin-compose-training-dessert-release/tree/main>