5

Using DataStore to Store Data and Testing

Modern Android Development practices help Android developers create better applications. DataStore is a data storage solution provided by the Android Jetpack library. It allows developers to store key-value pairs or complex objects asynchronously and with consistency guarantees. Data is critical in Android development, and how we save and persist data matters. In this chapter, we will explore using DataStore to persist our data and look at best practices using DataStore.

In this chapter, we'll be covering the following recipes:

- Implementing DataStore
- Adding Dependency Injection to DataStore
- Using Android Proto DataStore versus DataStore
- Handling data migration with DataStore
- Writing tests for our DataStore instance

Technical requirements

The complete source code for this chapter can be found at **https://github.com/PacktPublishing/Modern-Android-13-Development-Cookbook/tree/main/chapter_five**.

Implementing DataStore

When building mobile applications, it is critical to ensure that you persist your data in order to allow for smooth loading, reduce network issues, or even handle data entirely offline. In this recipe, we will look at how to store data in our Android applications using the Modern Android Development Jetpack library called DataStore.

DataStore is a data storage solution for Android applications that enables you to store key-value pairs or any typed objects with protocol buffers. Moreover, DataStore uses Kotlin coroutines and flows to store data consistently, transactionally, and asynchronously.

If you have built Android applications before, you might have used `SharedPreferences`. The new Preferences DataStore aims to replace

this old method. It is also fair to say that Preferences DataStore harnesses the power of **SharedPreferences** since they are pretty similar. In addition, Google's documentation recommends that if you're currently using **SharedPreferences** in your project to store data, you consider migrating to the latest DataStore version.

Another way to store data in Android is by using Room. This will be covered in *Chapter 6*, *Using the Room Database and Testing*; for now, we will just look at DataStore. Moreover, it is essential to note that DataStore is ideal for simple or small datasets and does not have support for partial updates or referential integrity.

How to do it...

Let's go ahead and create a new, empty Compose project and call it **DataStoreSample**. In our example project, we will create a task entry app where users can save tasks. We will allow users to enter only three tasks, then use DataStore to store the tasks and later log the data and see whether it was inserted correctly. An additional exercise to try is to display data when users want to see it:

1. In our newly created project, let's go ahead and delete code that we don't need. In this case, we're referring to the **Greeting(name: String)** that comes with all empty Compose projects. Keep the Preview function since we will use it to view the screen we create.
2. Now, let's go on and add the required dependencies for DataStore and sync the project. Also, note that there are versions of the DataStore library that are specific to RxJava 2 and 3:

```
dependencies {
implementation "androidx.DataStore:DataStore-preferences:1.x.x"
}
```

3. Create a new package and call it **data**. Inside **data**, create a new Kotlin data class and call it **Tasks**.
4. Let's now go ahead and construct our data class with the expected input fields:

```
data class Tasks(
    val firstTask: String,
    val secondTask: String,
    val thirdTask: String
)
```

5. Inside the same package, let's add a **TaskDataSource** enum since we will reuse this project to showcase saving data using Proto DataStore in the *Using Android Proto DataStore versus DataStore* recipe:

```
enum class TaskDataSource {
    PREFERENCES_DATA_STORE
}
```

6. Inside our package, let's go ahead and add a **`DataStoreManager`** interface. Inside our class, we will have a **`saveTasks()`** function to save the data and a **`getTasks()`** function to help us retrieve the saved data. A **`suspend`** function in Kotlin is simply a function that can be paused and resumed later.

In addition, the suspend functions can execute long-running operations and await completion without blocking:

```
interface DataStoreManager {
    suspend fun saveTasks(tasks: Tasks)
    fun getTasks(): Flow<Tasks>
}
```

7. Next, we need to implement our interface, so let's go ahead and create a **`DataStoreManagerImpl`** class and implement the **`DataStoreManager`**. To refresh your knowledge of Flows, refer to *Chapter 3*, *Handling the UI State in Jetpack Compose and Using Hilt*:

```
class DataStoreManagerImpl(): DataStoreManager {
    override suspend fun saveTasks(tasks: Tasks) {
        TODO("Not yet implemented")
    }
    override fun getTasks(): Flow<Tasks> {
        TODO("Not yet implemented")
    }
}
```

8. You will notice that once we've implemented the interface, we brought a view to the function, but it says **`TODO`**, and nothing has been implemented. To continue with this step, let's go ahead and add DataStore and pass **`Preference`** in our constructor. We will also need to create the string preference key for each task:

```
class DataStoreManagerImpl(
    private val tasksPreferenceStore:
        DataStore<Preferences>
) : DataStoreManager {
    private val FIRST_TASK =
        stringPreferencesKey("first_task")
    private val SECOND_TASK =
        stringPreferencesKey("second_task")
    private val THIRD_TASK =
        stringPreferencesKey("third_task")
    override suspend fun saveTasks(tasks: Tasks) {
        tasksPreferenceStore.edit {
        taskPreferenceStore ->
            taskPreferenceStore[FIRST_TASK] =
                tasks.firstTask
            taskPreferenceStore[SECOND_TASK] =
                tasks.secondTask
            taskPreferenceStore[THIRD_TASK] =
                tasks.thirdTask
        }
```

```
        }
        override fun getTasks(): Flow<Tasks> {
            TODO("Not yet implemented")
        }
    }
```

9. Finally, let's finish our implementation of the **`DataStore`** section by adding functionality to the **`getTasks`** function:

```
override fun getTasks(): Flow<Tasks> = tasksPreferenceStore.data.map { taskPreference ->
    Tasks(
        firstTask = taskPreference[FIRST_TASK] ?: "",
        secondTask = taskPreference[SECOND_TASK] ?:
        "",
        thirdTask = taskPreference[THIRD_TASK] ?: ""
    )
}
```

10. In our **`MainActivity`** class, let's go on and create a simple UI: three **`TextField`** and a **Save** button. The **Save** button will save our data, and we can try to log data once everything works as expected. Refer to the *Technical requirements* section of this chapter to get the UI code.

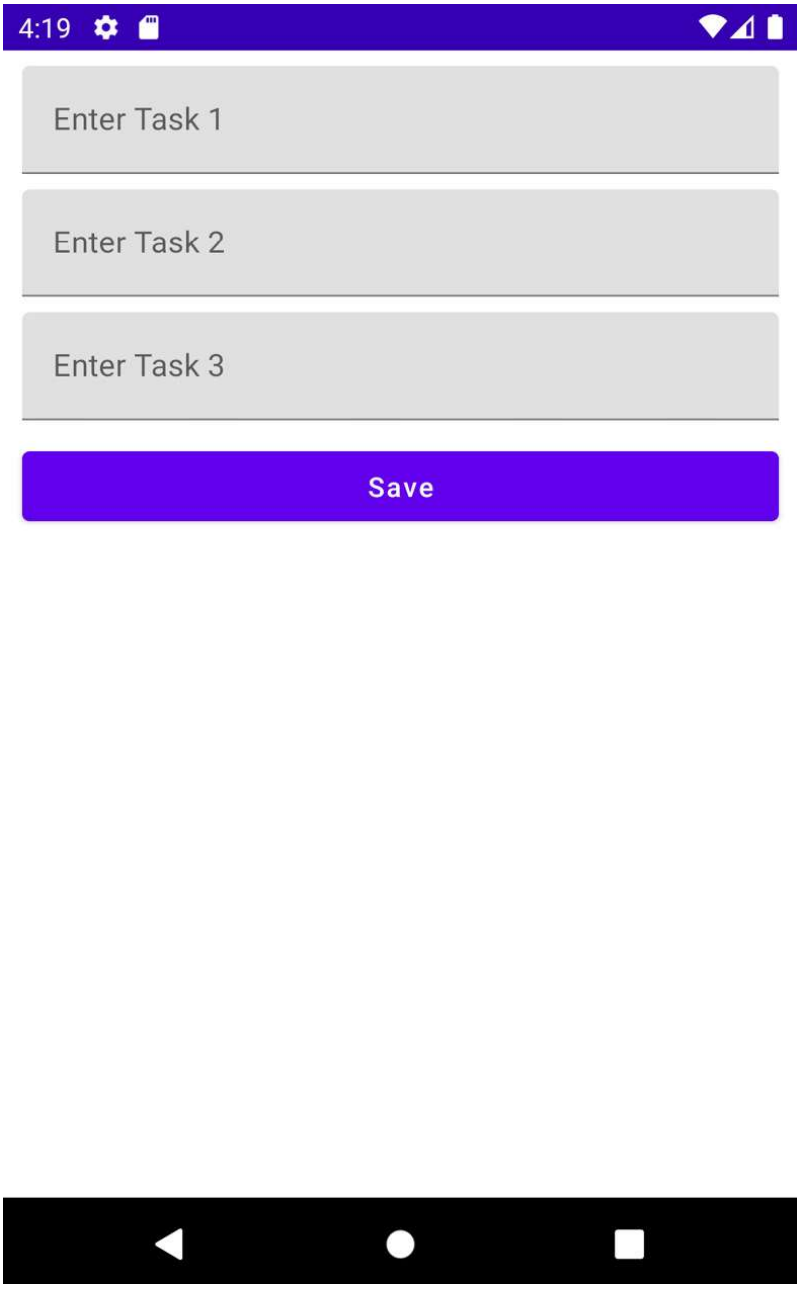

Figure 5.1 – The DataStore UI example

Now that we have our implementation ready, in the following recipe, *Adding Dependency Injection to DataStore*, we will add Dependency Injection and then glue everything together.

How it works...

The new Modern Android Development Jetpack library called Preferences DataStore's main objective is to replace **`SharedPreferences`**. To implement Preferences DataStore, as you have seen in the recipe, we use a DataStore interface that takes in a **`Preference`** abstract class, and we can use this to edit and map the entry data. Furthermore, we create keys for the crucial parts of the key-value pairs:

```
private val FIRST_TASK = stringPreferencesKey("first_task")
private val SECOND_TASK = stringPreferencesKey("second_task")
private val THIRD_TASK = stringPreferencesKey("third_task")
```

To save our data in DataStore, we use **`edit()`**, which is a suspend function that needs to be called from **`CoroutineContext`**. A key difference in using Preferences DataStore compared to **`SharedPreferences`** is that DataStore is safe to call on the UI thread since it uses **`dispatcher.IO`** under the hood.

You also do not need to use **`apply{}`** or **`commit`** functions to save the changes, as is required in **`SharedPreferences`**. Moreover, it handles data updates transactionally. More features are listed in *Figure 5.2*.

Feature	SharedPreferences	PreferencesDataStore	ProtoDataStore
Async API	✅ (only for reading changed values, via listener)	✅ (via `Flow` and RxJava 2 & 3 `Flowable`)	✅ (via `Flow` and RxJava 2 & 3 `Flowable`)
Synchronous API	✅ (but not safe to call on UI thread)	❌	❌
Safe to call on UI thread	❌1	✅ (work is moved to `Dispatchers.IO` under the hood)	✅ (work is moved to `Dispatchers.IO` under the hood)
Can signal errors	❌	✅	✅
Safe from runtime exceptions	❌2	✅	✅
Has a transactional API with strong consistency guarantees	❌	✅	✅
Handles data migration	❌	✅	✅
Type safety	❌	❌	✅ with Protocol Buffers

Figure 5.2 – A list of a sample of Datastore's features taken from developers.android.com

There is more to learn, and it is fair to acknowledge that what we covered in this recipe is just a tiny part of what you can do with DataStore. We will cover more features in the following recipes.

Adding Dependency Injection to DataStore

Dependency Injection is an important design pattern in software engineering, and its use in Android app development can lead to cleaner and more maintainable code. When it comes to DataStore in Android, which is a modern data storage solution introduced in Android Jetpack, adding Dependency Injection can bring several benefits:

- By using Dependency Injection, you can separate the concerns of creating an instance of DataStore from the code that uses it. This means that your business logic code will not have to worry about how to create a DataStore instance and can instead focus on what it needs to do with the data.

- Dependency Injection makes it easier to write unit tests for your app. By injecting a mock DataStore instance into your tests, you can ensure that your tests are not affected by the actual state of the DataStore.
- Dependency Injection can help you break down your code into smaller, more manageable modules. This makes it easier to add new features or modify existing ones without affecting the entire code base.
- By using Dependency Injection, you can easily switch between different implementations of DataStore. This can be useful when testing different types of data storage or when migrating from one storage solution to another.

How to do it...

You need to have completed the previous recipe to continue with this one by executing the following steps:

1. Open your project and add the necessary Hilt dependency. See the *Handling the UI State in Jetpack Compose and Using Hilt* recipe in *Chapter 3* if you need help setting it up.
2. Next, let's go ahead and add our **`@HiltAndroidApp`** class, and in our **`Manifest`** folder, add the **`.name = TaskApp: android:name=".TaskApp"`**:

```
@HiltAndroidApp
class TaskApp : Application()
<application
    android:allowBackup="true"
    android:name=".TaskApp"
    tools:targetApi="31">
...
```

3. Now that we have implemented Dependency Injection, let's go ahead and add **`@AndroidEntryPoint`** to the **`MainActivity`** class, and in **`DataStoreManagerImpl`**, let's go ahead and add the **`@Inject constructor`**. We should have something similar to the following code snippet:

```
class DataStoreManagerImpl @Inject constructor(
    private val tasksPreferenceStore:
    DataStore<Preferences>
) : DataStoreManager {
```

4. Now, we need to create a new folder and call it **`di`**; this is where we will put our **`DataStoreModule`** class. We create a file called **`store_tasks`** to store the Preference values:

```
@Module
@InstallIn(SingletonComponent::class)
class DataStoreModule {
    private val Context.tasksPreferenceStore :
    DataStore<Preferences> by
```

```
        preferencesDataStore(name = "store_tasks")
        @Singleton
        @Provides
        fun provideTasksPreferenceDataStore(
            @ApplicationContext context: Context
        ): DataStore<Preferences> =
           context.tasksPreferenceStore
    }
```

5. We will also need to create an **`abstract`** class for **`DataStoreManagerModule`** inside our **`di`** package. In order to reduce the boilerplate code using manual Dependency Injection, our application also supplies the required dependencies to the classes that need them. You can learn more about this in *Chapter 3*, *Handling the UI State in Jetpack Compose and Using Hilt*:

```
@Module
@InstallIn(SingletonComponent::class)
abstract class DataStoreManagerModule {
    @Singleton
    @Binds
    abstract fun
        bindDataStoreRepository(DataStoreManagerImpl:
            DataStoreManagerImpl): DataStoreManager
}
```

6. Let's now go ahead and create a new package and call it **`service`**:

```
interface TaskService {
    fun getTasksFromPrefDataStore(): Flow<Tasks>
    suspend fun addTasks(tasks: Tasks)
}
class TaskServiceImpl @Inject constructor(
    private val DataStoreManager: DataStoreManager
) : TaskService {
    override fun getTasksFromPrefDataStore() =
             DataStoreManager.getTasks()
    override suspend fun addTasks(tasks: Tasks) {
        DataStoreManager.saveTasks(tasks)
    }
}
```

7. Let's also ensure we have the required dependencies for the newly created service:

```
        @Singleton
        @Binds
        abstract fun bindTaskService(taskServiceImpl:
        TaskServiceImpl): TaskService
    }
```

8. Now that we are done with Dependency Injection and adding all the functionalities required for DataStore, we will go ahead and add a **`ViewModel`** class and implement functionality to save the data once the user clicks the Save button:

```
fun saveTaskData(tasks: Tasks) {
    viewModelScope.launch {
```

```
            Log.d("Task", "asdf Data was inserted
                   correctly")
            taskService.addTasks(tasks)
        }
    }
```

9. Call the **`saveTaskData`** function inside the Compose Save button in the Compose view to save our data:

```
TaskButton(onClick = {
    val tasks = Tasks(
        firstTask = firstText.value,
        secondTask = secondText.value,
        thirdTask = thirdText.value
    )
    taskViewModel.saveTaskData(tasks)},
    text = stringResource(id = R.string.save))
```

10. Lastly, we will need to verify that everything is working, that is, our UI and data storing process. We can verify this, by typing input data in our TextFields and clicking the Save button, and when we log the message it confirms the data is indeed saved.

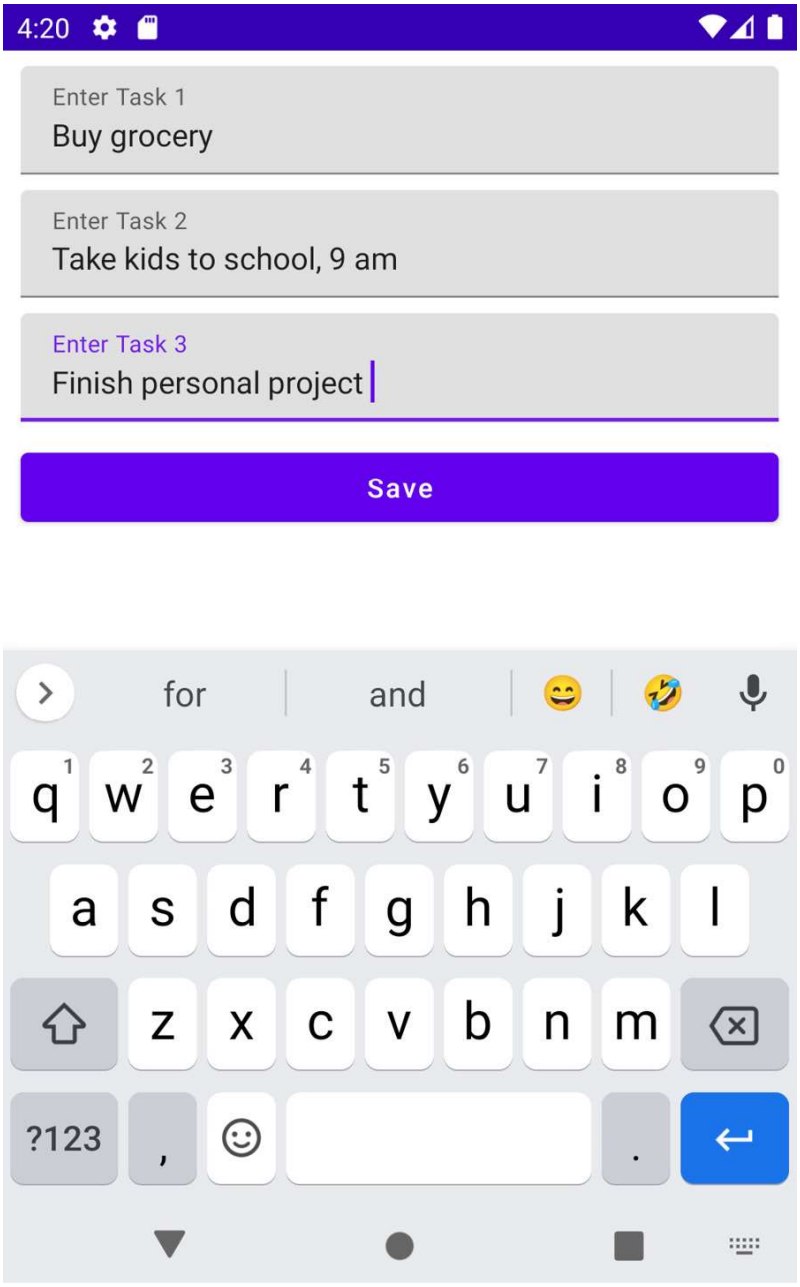

Figure 5.3 – The entry of tasks

11. If you missed it initially, the code for this view can be found in the *Technical requirement* section. Now, you will notice that when we enter the data, as in *Figure 5.4*, we should be able to log the data on our Logcat and verify that our data was inserted correctly.

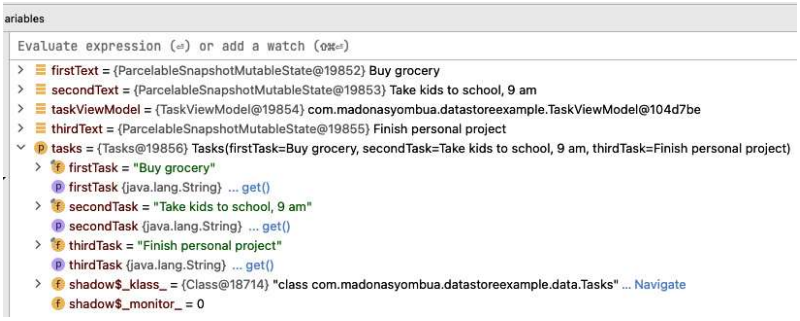

Figure 5.4 – The entry of tasks through debugging

12. A log message should also be displayed in the Logcat tab if all is working as expected.

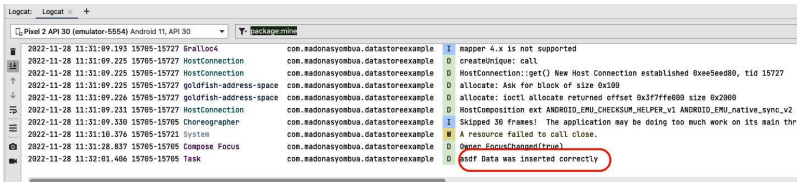

Figure 5.5 – The debug log indicating that data was inserted correctly

How it works...

In this recipe, we opted to use Dependency Injection to supply the required dependencies to specific classes. We've already covered what Dependency Injection is in depth, so we will not explain it again but instead talk about the modules we created.

In our project, we created **`DataStoreManagerModule`** and **`DataStoreModule`**, and all we did was supply the required dependencies. We created a file and called it **`store_tasks`**, which helps us store the Preference values:

```
private val Context.tasksPreferenceStore : DataStore<Preferences> by preferencesDataStore(name = "store_tasks"
```

By default, DataStore uses coroutines and returns a flow value. Some important rules to remember while using DataStore, as per the documentation, are as follows:

- DataStore requires only one instance for a given file in the same process. Hence, we should never create more than one instance of DataStore.
- Always have the generic **`DataStore`** type be immutable to reduce unnecessary and hard-to-trace bugs.
- You should never mix the usage of a single-process DataStore and multi-process DataStore in the same file.

There's more...

As an exercise, you can try to add another button and display the saved data in a lazy column or a text field.

See also

There is more to learn about DataStore, and this recipe has only given you an overview of what you can do with DataStore. You can learn more by following the link at **https://developer.android.com/topic/libraries/architecture/datastore**.

Using Android Proto DataStore versus DataStore

Figure 5.2 shows the differences between **`PreferencesDataStore`**, **`SharedPreferences`**, and **`ProtoDataStore`**. In this recipe, we will explore how we can use Proto DataStore. The Proto DataStore implementation uses DataStore and Protocol Buffers to persist typed objects to the disk.

Proto DataStore is similar to Preferences DataStore, but unlike Preferences DataStore, Proto does not use key-value pairs and just returns the generated object in the flow. The file types and structure of the data depend on the schema of the **`.protoc`** files.

Getting ready

We will use our already created project to show how you can use Proto DataStore in Android. We will also use already created classes and just give the functions different names.

How to do it...

1. We will need to start by setting up the required dependencies, so let's go ahead and add the following to our Gradle app-level file:

```
implementation "androidx.DataStore:DataStore:1.x.x
implementation "com.google.protobuf:protobuf-javalite:3.x.x"
```

2. Next, we will need to add **`protobuf`** to **`plugins`** in our **`build.gradle`** file:

```
plugins {
...
    id "com.google.protobuf" version "0.8.12"
}
```

3. We will need to add the **`protobuf`** configuration to our **`build.gradle`** file to finalize our setup:

```
protobuf {
    protoc {
        artifact = "com.google.protobuf:protoc:3.11.0"
    }
    generateProtoTasks {
        all().each { task ->
            task.builtins {
                 java {
                     option 'lite'
                 }
            }
        }
    }
}
```

4. Now, inside our **package** folder, we will need to add our **proto** file under **app/src/main/**, then create a new directory and call it **proto**. You should now have this in your **app/src/main/proto** file directory:

```
syntax = "proto3";
option java_package =
    "com.madonasyombua.DataStoreexample";
option java_multiple_files = true;
message TaskPreference {
    string first_task = 1;
    string second_task = 2;
    string third_task = 3;
}
```

That was a lot to set up. We can now start adding code to hook everything up.

5. Let's modify classes that might need **ProtoDataStore**. First, let's add **PROTO_DATA_STORE** to the **TaskDataSource** enum class:

```
enum class TaskDataSource {
    PREFERENCES_DATA_STORE,
    PROTO_DATA_STORE
}
```

6. In **DataStoreManager**, let's add **saveTaskToProtoStore()** and **getUserFromProtoStore()**, and our new interface will look like this:

```
interface DataStoreManager {
    suspend fun saveTasks(tasks: Tasks)
    fun getTasks(): Flow<Tasks>
    suspend fun saveTasksToProtoStore(tasks: Tasks)
    fun getTasksFromProtoStore(): Flow<Tasks>
}
```

7. Since we just modified our interface, we will need to go ahead and add new functionalities to the implementation class. You will also notice that the project will complain once you add the functions:

```
override suspend fun saveTasksToProtoStore(tasks: Tasks) {
    TODO("Not yet implemented")
}
override fun getTasksFromProtoStore(): Flow<Tasks> {
    TODO("Not yet implemented")
}
```

8. As recommended, we will need to define a class that implements **Serializer<Type>**, where the type is defined in the Proto file. The purpose of this serializer class is to tell DataStore how to read and write our data type. So, let's create a new object and call it **TaskSerializer()**:

```
object TaskSerializer : Serializer<TaskPreference> {
    override val defaultValue: TaskPreference =
        TaskPreference.getDefaultInstance()
```

```
        override suspend fun readFrom(input: InputStream):
            TaskPreference{
                try {
                    return TaskPreference.parseFrom(input)
                } catch (exception:
                InvalidProtocolBufferException) {
                  throw CorruptionException("Cannot read
                  proto.", exception)
                }
        }
        override suspend fun writeTo(t: TaskPreference,
            output: OutputStream) = t.writeTo(output)
    }
```

9. The **`TaskPreference`** class is auto-generated, and you can access it directly by clicking on it but cannot edit it. Auto-generated files are not editable unless you change the original file.

```
/.../

package com.madonasyombua.datastoreexample;

/**
 * Protobuf type {@code TaskPreference}
 */
public  final class TaskPreference extends
    com.google.protobuf.GeneratedMessageLite<
        TaskPreference, TaskPreference.Builder> implements
    // @@protoc_insertion_point(message_implements:TaskPreference)
    TaskPreferenceOrBuilder {
  private TaskPreference() {
    firstTask_ = "";
    secondTask_ = "";
    thirdTask_ = "";
  }
  public static final int FIRST_TASK_FIELD_NUMBER = 1;
  private java.lang.String firstTask_;
  /**
   * <code>string first_task = 1;</code>
   * @return The firstTask.
   */
  @java.lang.Override
  public java.lang.String getFirstTask() { return firstTask_; }
  /**
   * <code>string first_task = 1;</code>
   * @return The bytes for firstTask.
   */
  @java.lang.Override
  public com.google.protobuf.ByteString
      getFirstTaskBytes() { return com.google.protobuf.ByteString.copyFromUtf8(firstTask_); }
  /**
   * <code>string first_task = 1;</code>
```

Figure 5.6 – A screenshot showing the auto-generated TaskPreference class

10. Now that we have created our data type class, we need to create a **`taskProtoDataStore: DataStore<TaskPreference>`** with the context used with DataStore. Hence, inside **`DataStoreModule`**, let's go ahead and add this code:

```
    private val Context.taskProtoDataStore: DataStore<TaskPreference> by DataStore(
        fileName = "task.pd",
        serializer = TaskSerializer
    )
    @Singleton
    @Provides
```

```
fun provideTasksProtoDataStore(
    @ApplicationContext context: Context
):DataStore<TaskPreference> = context.taskProtoDataStore
```

11. Now, let's go back to **`DataStoreManagerImpl`** and work on the functions that we are yet to implement:

```
override suspend fun saveTasksToProtoStore(tasks: Tasks) {
    taskProtoDataStore.updateData { taskData ->
        taskData.toBuilder()
            .setFirstTask(tasks.firstTask)
            .setSecondTask(tasks.secondTask)
            .setThirdTask(tasks.thirdTask)
            .build()
    }
}
override fun getTasksFromProtoStore(): Flow<Tasks> =
    taskProtoDataStore.data.map { tasks ->
        Tasks(
            tasks.firstTask,
            tasks.secondTask,
            tasks.thirdTask
        )
```

12. In **`TaskService`**, we will also go ahead and add **`getTasksFromProto`**, and **`getTasks()`**:

```
interface TaskService {
    fun getTasksFromPrefDataStore() : Flow<Tasks>
    suspend fun addTasks(tasks: Tasks)
    fun getTasks(): Flow<Tasks>
    fun getTasksFromProtoDataStore(): Flow<Tasks>
}
```

13. When you implement an interface, at first the class that is being implemented will might show compile error, which will prompt you to override the interface functionalities into the class. Hence, inside the **`TaskServiceImpl`** class, add the following code:

```
class TaskServiceImpl @Inject constructor(
    private val DataStoreManager: DataStoreManager
) : TaskService {
    override fun getTasksFromPrefDataStore() =
        DataStoreManager.getTasks()
    override suspend fun addTasks(tasks: Tasks) {
        DataStoreManager.saveTasks(tasks)
        DataStoreManager.saveTasksToProtoStore(tasks)
    }
    override fun getTasks(): Flow<Tasks> =
        getTasksFromProtoDataStore()
    override fun getTasksFromProtoDataStore():
        Flow<Tasks> =
        DataStoreManager.getTasksFromProtoStore()
}
```

Finally, now that we have all our data saved, we can Log to ensure the data is as expected on the UI; check out the link with the code in the

Technical requirements section to see how this is implemented.

IMPORTANT NOTE

Apple M1 has a reported problem with proto. There is an issue open for this; follow this link to resolve the issue: **https://github.com/grpc/grpc-java/issues/7690**. *Hopefully, it will be fixed by the time the book is published. It is important to note that if you use the* **`DataStore-preferences-core`** *artifact with Proguard, you have to manually add Proguard rules to your rule file to prevent your already written fields from being deleted. Also, you can follow the same process to Log and check whether the data is inserted as expected.*

How it works...

You might have noticed that we stored our custom data type as an instance. That is what Proto DataStore does; it stores data as instances of custom data types. The implementations require us to define a schema using protocol buffers, but it provides type safety.

In Android's Proto Datastore library, the **`Serializer<Type>`** interface converts objects of a specific type (**`Type`**) into their corresponding protocol buffer format and vice versa. This interface provides methods for serializing objects to bytes and deserializing bytes back into objects.

Protocol buffers in Android is a language and platform-neutral extensible mechanism for serializing your structured data. Protocol buffers encodes and decodes your data in a binary stream that is really lightweight.

The override val defaultValue is used when defining a property in a data class or a serialized model class. It is part of the Kotlin Serialization library, which is commonly used for serializing and deserializing objects to and from different data formats such as JSON or protocol buffers.

We expose the appropriate property by exposing the flow DataStore data from our stored object and writing a proto DataStore that provides us with an **`updateData()`** function that transactionally updates a stored object.

The **`updateData`** function gives us the current state of the data as an instance of our data type and updates it in an atomic read-write-modify operation.

See also

There is more to learn about how to create defined schemas. You can check out the protobuf language guide here:

https://developers.google.com/protocol-buffers/docs/proto3.

Handling data migration with DataStore

If you have built Android applications before, you might have used **SharedPreferences**; the good news now is that there is support for migration, and you can migrate from **SharedPreferences** to DataStore using **SharedPreferenceMigration**. As with any data, we will always modify our dataset; for instance, we might want to rename our data model values or even change their type.

In such a scenario, we will need a DataStore to DataStore migration; that is what we will be working on in this recipe. The process is pretty similar to the migration from **SharedPreferences**; as a matter of fact, **SharedPreferencesMigration** is an implementation of the **DataMigration** interface class.

Getting ready

Since we just created a new **PreferenceDataStore**, we will not need to migrate it, but we can look at ways to implement a migration in case a need arises.

How to do it...

In this recipe, we will look at how you can utilize the knowledge learned to help you when a need to migrate to DataStore arises:

1. Let's start by looking at the interface that helps with migration. The following code section showcases the **DataMigration** interface, which **SharedPreferencesMigration** implements:

```
/* Copyright 2022 Google LLC.
   SPDX-License-Identifier: Apache-2.0 */
public interface DataMigration<T> {
    public suspend fun shouldMigrate(currentData: T): Boolean
    public suspend fun migrate(currentData: T): T
        public suspend fun cleanUp()
}
```

2. In the **Tasks** data, we might want to change the entries to **Int**; this means changing one of our data types. We will imagine this scenario and try to create a migration based on this. We can start by creating a new **migrateOnePreferencesDataStore**:

```
private val Context.migrateOnePreferencesDataStore : DataStore<Preferences> by preferencesDataStore(
    name = "store_tasks"
)
```

3. Now, let's go on to implement **`DataMigration`** and override its functions. You will need to specify your condition for whether the migration should happen. The migration data shows instructions on how exactly the old data is transformed into new data. Then, once the migration is over, clean up the old storage:

```
private val Context.migrationTwoPreferencesDataStore by preferencesDataStore(
    name = NEW_DataStore,
    produceMigrations = { context ->
        listOf(object : DataMigration<Preferences> {
            override suspend fun
                shouldMigrate(currentData:
                    Preferences) = true
            override suspend fun migrate(currentData:
            Preferences): Preferences {
                val oldData = context
                    .migrateOnePreferencesDataStore
                        .data.first().asMap()
                val currentMutablePrefs =
                    currentData.toMutablePreferences()
                oldToNew(oldData, currentMutablePrefs)
                return
                   currentMutablePrefs.toPreferences()
            }
            override suspend fun cleanUp() {
                context.migrateOnePreferencesDataStore
                    .edit { it.clear() }
            }
        })
    }
)
```

4. Finally, let's create the **`oldToNew()`** function, which is where we can add the data we want to migrate:

```
private fun oldToNew(
    oldData: Map<Preferences.Key<*>, Any>,
    currentMutablePrefs: MutablePreferences
) {
    oldData.forEach { (key, value) ->
        when (value) {
            //migrate data types you wish to migrate
            ...
        }
    }
}
```

How it works...

To better understand how **`DataMigration`** works, we will need to look into the functions that are in the **`DataMigration`** interface. In our interface, we have three functions, as shown in the following code block:

```
public suspend fun shouldMigrate(currentData: T):
    Boolean
public suspend fun migrate(currentData: T): T
public suspend fun cleanUp()
```

The **`shouldMigrate()`** function, as the name suggests, establishes whether the migration needs to be performed or not. If, for instance, no migration is done, which means this will return **`false`**, then no migration or cleanup will occur. Also, it is crucial to note that this function is initialized every time we call our DataStore instance. **`Migrate()`**, on the other hand, performs the migration.

By chance, if the action fails or does not work as expected, DataStore will not commit any data to the disk. Furthermore, the cleanup process will not occur, and an exception will be thrown. Finally, **`cleanUp()`**, as it suggests, just clears any old data from previous data storage.

Writing tests for our DataStore instance

Writing tests is crucial in Android development, and in this recipe, we will write some tests for our DataStore instance. To test our DataStore instance or any DataStore instance, we first need to have instrumentation testing set up since we will be reading and writing in actual files (DataStore), and it is vital to verify that accurate updates are being made.

How to do it...

We will start by creating a simple unit test to test our view model function:

1. In our unit test folder, create a new folder and call it **`test`**, and inside it, go ahead and create a new class called **`TaskViewModelTest`**:

   ```
   class TaskViewModelTest {}
   ```

2. Next, we will need to add some testing dependencies:

   ```
   testImplementation "io.mockk:mockk:1.13.3"
   androidTestImplementation "io.mockk:mockk-android:1.13.3"
   testImplementation "org.jetbrains.kotlinx:kotlinx-coroutines-test:1.5.2"
   ```

3. Now that we have added the required dependencies, let's go ahead and create our mock task service class and mock it, then initialize it in the setup:

   ```
   private lateinit var classToTest: TaskViewModel
   private val mockTaskService = mockk<TaskService>()
   private val dispatcher = TestCoroutineDispatcher()
   @Before
   fun setUp(){
   ```

```
    classToTest = TaskViewModel(mockTaskService)
}
```

4. Since we use a coroutine, we will set up our dispatcher in **@Before annotation** and clear any stored data in the **@After** annotation using the **Dispatchers.resetMain()**. If you run your tests without setting up a coroutine, they will fail with an error. The module with the **Main** dispatcher failed to initialize. For tests, **Dispatchers.setMain** from the **kotlinx-coroutines-test** module can be used:

```
@Before
fun setUp(){
    classToTest = TaskViewModel(mockTaskService)
    Dispatchers.setMain(dispatcher)
}
@After
fun tearDown() {
    Dispatchers.resetMain()
}
```

5. After that is completed, let's go on and create a new test called **Verify add tasks function adds tasks as needed**. In this test, we will create a **fakeTask**, add those tasks to **saveTaskData**, and ensure data is inserted as expected by checking that we did not store **null**:

```
@Test
fun  `Verify add tasks function adds tasks as needed`() = runBlocking {
    val fakeTasks = Tasks(
        firstTask = "finish school work",
        secondTask = "buy gifts for the holiday",
        thirdTask = "finish work"
    )
    val expected = classToTest.saveTaskData(fakeTasks)
    Assert.assertNotNull(expected)
}
```

Finally, when you run the unit test, it should pass, and you will see a green check mark.

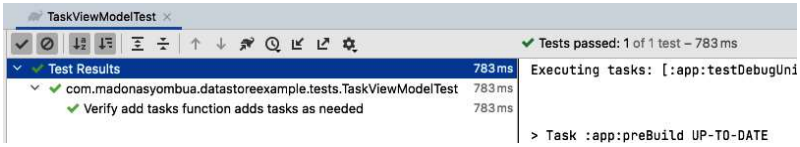

Figure 5.7 – The test passing in the view model

How it works...

There are different mocking libraries used in Android: **Mockito**, **Mockk**, and more. In this recipe, we used **Mockk**, a user-friendly mocking library for Android. **testImplementation "io.mockk:mockk:1.13.3"** is used for unit tests, and **androidTestImplementation "io.mockk:mockk-android:1.13.3"** is used for UI tests.

To test the UI, we will need to follow a pattern, creating a test DataStore instance with default values stored inside it. Then, we create the test subject and verify that the test DataStore values coming from our function match the expected results. We will also need to use **`TestCoroutineDispatcher`**:

```
private val coroutineDispatcher: TestCoroutineDispatcher =
    TestCoroutineDispatcher()
```

The preceding code performs the execution of the coroutines, which is, by default, immediate. This simply means any tasks scheduled to be run without delays are executed immediately. We also use the same coroutines for our view models. That is also because DataStore is based on Kotlin coroutines; hence we need to ensure our tests have the right setup.

See also

There is more to learn about DataStore. We cannot cover it all in just one chapter. For more information on DataStore, that is, Preference and Proto, you can check out this link: **https://developer.android.com/topic/libraries/architecture/datastore**.