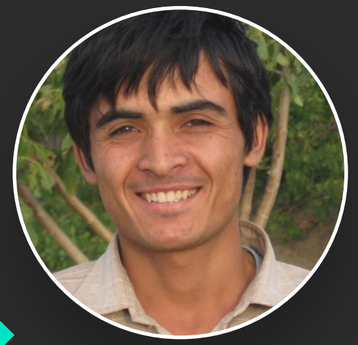
droidcon academy

Exploring Proto DataStore



Mehdi Haghgoo

@IAmMehdiHaghgoo

in/MehdiHaghgoo

Section Overview

- Define and use protobuf objects
- Implement a protobuf serializer
- Create an instance of Proto DataStore
- Read from and write to Proto DataStore
- Handle read and write exceptions

droidcon academy

Define Protobuf objects

- Protobuf:
  - Define a structured data
  - Compiler generates the source code needed to read and write the structured data
- Structured data defined in a proto file like user_pref.pb inside src/main/proto
  - Specify the syntax to be used
  - Structured data is defined using message
  - Members of the structured data are defined inside the message

```
// inside a proto file
syntax = "proto3";

option java_package = "com.droidcon.comicsworld";
option java_multiple_files = true;

message UserComicPreference {
  // enum representing the different sort orders
for our app
   enum SortOrder{
        UNSPECIFIED = 0;
        NONE = 1;
        BY_RATING = 2;
        BY_DATE_ADDED = 3;
    }

}
```

Define a Serializer

- Serializer enables the data-store to know how to:
  - Read and write the data-structure in our proto file
  - Return a default value in case there is no data persisted in our proto file
- To create a serializer we implement the Serializer<T> interface

```
// UserPreferenceSerializer
object UserPreferencesSerializer:
   Serializer<UserComicPreference>{
    override val defaultValue: UserComicPreference
        get() = UserComicPreference.getDefaultInstance()
    override suspend fun readFrom(input: InputStream):
UserComicPreference {
      return try {
          UserComicPreference.parseFrom(input)
        } catch (e:InvalidProtocolBufferException){
          throw CorruptionException(
"Cannot read from the proto file because of $e")
        }
    }
    override suspend fun writeTo(t: UserComicPreference, output:
OutputStream) = t.writeTo(output)
}
```

droidcon academy

Creating a Proto DataStore

- 2 ways to create an instance of a Proto DataStore
  - Using the protoDataStore property delegate with a Context Receiver
    - used when you are providing your datastore instance manually
  - Using the DataStoreFactory API
    - used with dependency injection frameworks such as Hilt

Using Property Delegate

Construct a proto datastore instance using property delegate

```
// at the top level of your file

private const val DATA_STORE_FILE_NAME =
"user_prefs.pb"
private const val SORT_ORDER_KEY = "sort_order"

private val Context.userProtoDatastore:
DataStore<UserPreferences> by dataStore(
    fileName = DATA_STORE_FILE_NAME,
    serializer = UserPreferencesSerializer
)
```

Using DataStoreFactory

- Create a Proto DataStore instance: use DataStoreFactory.create()
- Parameters:
  - serializer
  - produceFile

```
// creating a proto data-store instance using
DataStoreFactory
 @Provides
 @Singleton
 fun provideProtoDataStore(@ApplicationContext
appContext: Context): DataStore<UserComicPreference>
{
        return DataStoreFactory.create(
            serializer = UserPreferencesSerializer)
            {
               File(appContext.filesDir,
  USER_PROTO_PREFERENCES_NAME)
        }
    }
```

Reading from Proto DataStore

- Read via DataStore<Preferences>.data property
- The read data will be exposed via Flow<Preference>
  - Latest changes stored in datastore are emitted
- The returned flow will always either:
  - Emit a value or
  - Throw an exception

Writing data to Proto Data-store

- The DataStore<Preference>.updateData() function used to : write data to Proto DataStore
- The DataStore<Preference>.updateData() function is suspending
- The function updates data transanctionally

```
   suspend fun filterByCategory(
comicCategory:UserComicPreference.ComicCategory
  ) {
     userPreferenceProtoStore.updateData {currentPreferences ->
                currentPreferences.toBuilder()
.setComicCategory(comicCategory).build()
 }
}
```

Handling Read Exceptions

- Proto DataStore throws an IOException whenever an error occurs during the reading/writing operation
- Handling of the exception during read is done using the Flow's catch() operator

```
 fun getUserPreferences(): Flow<UserComicPreference> {
        return userPreferenceProtoStore.data.catch
{exception->
            if (exception is IOException){
            Log.e(LoggingTag,"An error occurred while
reading from the proto data store $exception")

emit(UserComicPreference.getDefaultInstance())
            }
            else throw exception
        }
}
```

Handling Write Exceptions

- Proto DataStore throws an IOException whenever an error occurs during the reading/writing operation
- Handling of exceptions during writing is done via a try-catch block

```
 suspend fun filterByCategory(
comicCategory:UserComicPreference.ComicCategory) {
    try{
            userPreferenceProtoStore.updateData
{currentPreferences ->
            currentPreferences
            .toBuilder()
            .setComicCategory(comicCategory)
            .build()
        }
     }catch(ioException:IOException){
         // do something with the exception
         }
}
```

Code Challenge

- Add a new sort order option

    - The new sort order must enable the user to sort the comics based on their names

- Implement read and write functionality of the new sort order option in the **UserProtoPreferencesRepository.kt**.

droidcon academy

Section Summary

- Define protobuf objects
- Implement a protobuf serializer
- Create a Proto Datastore instance
- Read from and write to the Proto DataStore
- Handle read and write exceptions thrown by Proto DataStore

droidcon academy

Proto DataStore in Action

Up Next