



30-Day Android Challenge

From Basics to Advanced

(Expert-Level Deconstruction 📱)

Day 16

Jetpack DataStore

Welcome to Modern, Asynchronous, and Safe Data Storage in Android

(Full Coverage of Preferences & Proto DataStore) 🚀



Esmaeel Moradi

@esmaeel-moradi700

Day 16



Jetpack DataStore: Farewell to SharedPreferences, Welcome to Modern, Asynchronous, and Safe Data Storage in Android (Full Coverage of Preferences & Proto DataStore) 🚀

Greetings again to all professional colleagues, Android engineers, and architects across the globe!

If you've been wrestling with Android for years, the name *SharedPreferences* is undoubtedly familiar – a simple, handy tool for storing small key-value data. But, let's be honest and, without mincing words, look at its limitations and problems. As someone who has witnessed the evolution (and sometimes the headaches) of data persistence in Android for over a decade, I can confidently say that the era of relying on SharedPreferences for serious tasks is over.

Today, on the sixteenth day of our challenge, we turn to the new champion in the field: Jetpack DataStore. This modern and powerful solution from Google, built upon Kotlin Coroutines and Flow, is here not only to solve all the problems of SharedPreferences but also to provide far more capabilities for storing simple and even structured (typed) data. Still relying on the aging SharedPreferences for your app's settings or simple data persistence? It's time to upgrade to Jetpack DataStore and enjoy asynchronous, transactional, and (if using Proto) type-safe data storage, and handle potential errors with peace of mind!



Esmaeel Moradi

@esmaeeil-moradi700

Introduction: Beyond SharedPreferences – The Modern Data Persistence Solution in Android with DataStore.

SharedPreferences served us for years, but its inherent problems were always like a shadow accompanying us:

- **Synchronous API:** Calling `apply()` or `commit()` (the latter being even worse as it runs on the UI thread!) could lead to blocking the UI thread and causing ANRs (Application Not Responding), especially if I/O operations were slow.
- **No Built-in Error Signaling:** If a problem occurred during reading or writing, SharedPreferences wouldn't simply throw an exception, making it difficult to ascertain if the operation was successful.
- **No Transactional API Support:** If you needed multiple atomic changes, there was no guarantee of complete execution or no execution at all.
- **Lack of Type Safety Beyond Primitives:** It was easy to work with the wrong key or data type and encounter runtime errors.
- **Performance Issues:** With large amounts of data or frequent calls, performance could be severely impacted.

Jetpack DataStore addresses all these issues with a modern approach.



Esmaeel Moradi

@esmaeel-moradi700

Understanding DataStore: Two Different Flavors for Different Needs. →

DataStore comes in two flavors:

- **Preferences DataStore:**
 - Purpose: Storing simple key-value pairs (similar to SharedPreferences but much better).
 - Key Advantages: Fully asynchronous API using Flow for reading and suspend functions for writing, transparent error signaling via exceptions, and transactional updates with edit().
 - No need to define a predefined schema.
- **Proto DataStore:**
 - Purpose: Storing typed, structured data using Protocol Buffers (Protobuf).
 - Key Advantages: Full type safety at compile time, clear and powerful schema definition via .proto files, highly efficient and compact serialization, and built-in support for safe data schema migration.
 - Requires defining a data schema with Protocol Buffers.
- **When to Choose Which?**
 - **Preferences DataStore:** For simple app settings, flags, user-selected themes, and any small key-value data that doesn't require a complex schema.
 - **Proto DataStore:** For storing more complex data objects, typed data models, or when type safety and advanced schema migration capabilities are highly important to you.



Esmaeil Moradi
@esmaeil-moradi700



Preferences DataStore (The Modern Replacement for *SharedPreferences*):

Let's see how to use this powerful replacement.

- **Setup:** Add the necessary dependency to build.gradle:

```
// build.gradle (Module)
dependencies {
    implementation "androidx.datastore:datastore-preferences:1.1.1" // Check
}
```

- **Creating a DataStore Instance:** Usually done using the preferencesDataStore delegate at the file level (outside a class) or as a Singleton for the entire application:

"settings" is the name of the file DataStore uses for storing data.

```
// Add the necessary imports at the top of your Kotlin file
import android.content.Context
import androidx.datastore.core.DataStore
import androidx.datastore.preferences.core.Preferences
import androidx.datastore.preferences.preferencesDataStore

// Create the DataStore instance using the Context extension
// (Typically at the top level of your Kotlin file)
val Context.settingsDataStore: DataStore<Preferences>
    by preferencesDataStore(name = "settings")
```



Esmaeel Moradi

@esmaeeil-moradi700



- **Defining Keys:** For each value, we define a key with a specific type:

```
// Define keys for your preferences
object UserPreferencesKeys {
    val THEME_MODE = intPreferencesKey("theme_mode")
    val USER_NAME = stringPreferencesKey("user_name")
    val IS_LOGGED_IN = booleanPreferencesKey("is_logged_in")
}
```

- **Writing Data (Edit Operation - edit):** Write operations in Preferences DataStore are transactional and asynchronous, performed using the suspend function edit:

```
// Writing to Preferences DataStore (inside a coroutine scope)
suspend fun saveThemeMode(context: Context, themeMode: Int) {
    context.settingsDataStore.edit { settings ->
        settings[UserPreferencesKeys.THEME_MODE] = themeMode
    }
}
// Call this from a lifecycleScope.launch, viewModelScope.launch, etc.
```

The *edit* block is executed atomically. If an error occurs during the operation, no changes will be applied.



Esmaeel Moradi
@esmaeel-moradi700



- **Reading Data:** Data from Preferences DataStore is read as a *Flow<Preferences>*. This means you can reactively listen to changes.

```
// Reading from Preferences DataStore
val themeModeFlow: Flow<Int> = context.settingsDataStore.data
    .map { preferences ->
        // Provide a default value if the key does not exist
        preferences[UserPreferencesKeys.THEME_MODE] ?: ThemeManager.DEFAULT_THEME_MODE
    }

// Collect the flow in a coroutine scope (e.g., in a ViewModel or Composable)
// lifecycleScope.launch {
//     themeModeFlow.collect { themeMode ->
//         // Update UI or use the themeMode
//     }
// }
```



Esmaeel Moradi
@esmaeeil-moradi700



Error Handling:

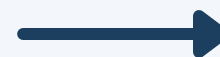
Read (*collect* on the *data* Flow) and write (*edit*) operations can throw *IOException*. Be sure to handle them in a *try-catch* block.

```
Error handling when reading  
context.settingsDataStore.data  
.catch { exception ->  
if (exception is IOException) {  
Log.e("DataStore", "Error reading preferences.", exception)  
emit(emptyPreferences())  
Emit empty preferences on error  
} else {  
throw exception  
}  
}  
.map { ... }  
...
```



Esmaeel Moradi

@esmaeel-moradi700



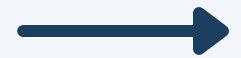
Migrating from *SharedPreferences*:

DataStore provides a way to migrate from existing SharedPreferences:

```
val Context.settingsDataStore: DataStore<Preferences> by preferencesDataStore(  
    name = "settings",  
    produceMigrations = { context ->  
        listOf(SharedPreferencesMigration(context, "your_shared_prefs_name"))  
    }  
)
```



Esmaeel Moradi
@esmaeeil-moradi700



Proto DataStore (For Typed, Structured Data):

When you need to store typed objects, Proto DataStore comes into play.

- **Setup:**

Dependencies:

```
// build.gradle (Module Level - Groovy DSL)

dependencies {
    // For Proto DataStore. IMPORTANT: Always check for the latest stable version!
    // The version "1.1.1" is likely outdated as of May 2025.
    implementation "androidx.datastore:datastore-core:1.1.1"

    // For Protobuf with Kotlin. IMPORTANT: Always check for the latest stable version!
    // The version "3.25.3" is likely outdated.
    // Use "protobuf-javalite" for Java-based projects or specific Java interop needs.
    implementation "com.google.protobuf:protobuf-kotlin-lite:3.25.3"
}

// --- Essential Configuration for Protobuf ---
// The following sections are typically commented out in snippets but are CRUCIAL.
// You MUST uncomment and configure them correctly.

// 1. Add Protobuf plugin definition to your project-level build.gradle(.kts)
// IMPORTANT: Use the latest stable plugin version. "0.9.4" is likely outdated.
// plugins {
//     id("com.google.protobuf") version "0.9.4" apply false
// }
```



Esmaeel Moradi
@esmaeeil-moradi700



Define Data Schema (.proto file):

Create a .proto file (e.g., user_prefs.proto) in the *app/src/main/proto* directory:

```
syntax = "proto3";

option java_package = "com.example.myapp.datastore";
option java_multiple_files = true;

message UserPreferences {
    string user_name = 1;
    int32 theme_color_argb = 2;
    bool pro_member = 3;
}
```

After defining, build the project to generate the corresponding Kotlin/Java classes by the Protobuf compiler.



Esmaeel Moradi
@esmaeeil-moradi700



Create a Serializer:

Implement the `Serializer<T>` interface (where T is your Protobuf data type):

```
object UserPreferencesSerializer : Serializer<UserPreferences> {
    override val defaultValue: UserPreferences = UserPreferences.getDefaultI

    override suspend fun readFrom(input: InputStream): UserPreferences {
        try {
            return UserPreferences.parseFrom(input)
        } catch (exception: InvalidProtocolBufferException) {
            throw CorruptionException("Cannot read proto.", exception)
        }
    }

    override suspend fun writeTo(t: UserPreferences, output: OutputStream) =
        t.writeTo(output)
}
```

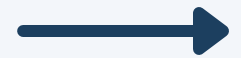
Create a Proto DataStore Instance:

```
// Create the Proto DataStore instance
val Context.userPreferencesDataStore: DataStore<UserPreferences>
    by datastore(
        fileName = "user_prefs.pb", // .pb extension is conventional
        serializer = UserPreferencesSerializer
    )
```



Esmaeel Moradi
@esmaeeil-moradi700

Writing Data (Update Operation - `updateData`):



```
// Writing to Proto DataStore (inside a coroutine scope)
suspend fun updateUserPreferences(context: Context, newName: String) {
    context.userPreferencesDataStore.updateData { currentPreferences ->
        currentPreferences.toBuilder()
            .setUserName(newName)
            .build()
    }
}
```

`updateData` is also transactional and a *suspend* function.

Reading Data: Data is read as a `Flow<MyProtoType>`:

```
// Reading from Proto DataStore
val userPreferencesFlow: Flow<UserPreferences> =
    context.userPreferencesDataStore.data
        .catch { exception ->
            /* Handle IOException, CorruptionException */
            // TODO: Add actual error handling logic here,
            // e.g., emit a default state or log the error.
        }

// Collect in a coroutine scope
// lifecycleScope.launch {
//     userPreferencesFlow.collect { prefs ->
//         val userName = prefs.userName
//         // ...
//     }
// }
```

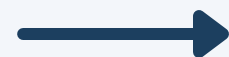
Data Schema Migrations: If your .proto schema changes (e.g., a field is added or removed), you must implement a `DataMigration` to convert old data to the new format. This is one of the strengths of Proto DataStore. (Implementing `DataMigration` is slightly more advanced and requires defining the precise transformation logic).



Esmaeel Moradi

@esmaeeil-moradi700

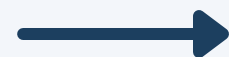
Key Advantages of DataStore (Reiteration & Deepening):



- **Asynchronous API (Coroutines & Flow):** I/O operations are moved off the Main Thread, making your app smoother.
- **Built-in Error Handling:** IOExceptions are standardly catchable and manageable.
- **Transactional API:** Ensures operations are either fully completed or not at all.
- **Type Safety (with Proto DataStore):** Say goodbye to runtime errors from wrong keys or types!
- **Observability with Flow:** Easily listen to data changes and update your UI reactively.



Esmaeel Moradi
@esmaeel-moradi700



Best Practices & A Veteran's Advanced Tips:

- **Dependency Injection (DI):** Define DataStore as a Singleton in your DI graph (Hilt, Koin) and inject it where needed.
- **Testing DataStore:** Use TestCoroutineDispatcher (or StandardTestDispatcher in newer tests) and Flow testing utilities to test read/write logic and migrations.
- **Choosing Between Preferences and Proto:** If you only need simple key-values, Preferences is sufficient. For anything more complex or typed, undoubtedly go for Proto.
- **Managing Coroutine Scopes:** For collecting Flows, use appropriate Scopes (like lifecycleScope in UI Controllers or viewModelScope in ViewModels, or a custom Scope) to prevent Memory Leaks.
- **Advanced Error Handling Strategies:** Beyond simple try-catch, you can wrap DataStore operation results in a Result class (containing success or error) for cleaner handling in higher layers.
- **DataStore in Multi-Module Projects:** Place DataStore in a :data or a Core module and access it via interfaces.
- **Data Security in DataStore:** Remember that DataStore itself does not encrypt data. If you need to store sensitive data, you should use the Jetpack Security library to encrypt the DataStore file or similar solutions.



Esmaeel Moradi

@esmaeeil-moradi700

Conclusion:



Future-Proofing Your App's Data Persistence with DataStore.

Jetpack DataStore is a major step forward in data persistence on Android. By providing asynchronous, transactional, and (in the case of Proto) type-safe APIs, it helps us build more robust, reliable, and better-performing applications. My strong recommendation as a seasoned developer is to fully migrate from SharedPreferences to DataStore for all new projects and even consider refactoring and replacing it in older projects at the earliest possible opportunity. DataStore not only solves past problems but also prepares you for the future.



Esmaeeil Moradi

@esmaeeil-moradi700

Looking forward to your
insights and experiences!
Let's elevate Android
development together.



LEAVE A COMMENT BELOW



Esmaeeil Moradi

@esmaeeil-moradi700