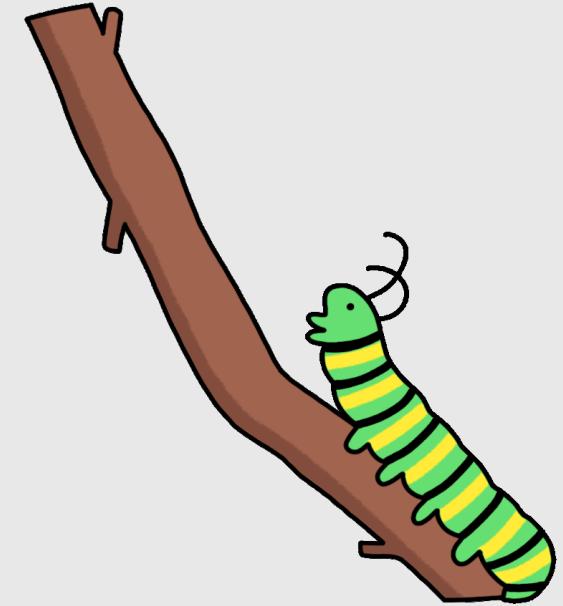


SharedPreferences Caterpillar to DataStore Butterfly: *How to Migrate and Make Your App Fly*



Viktor Arsovski
Senior Software Engineer @N47

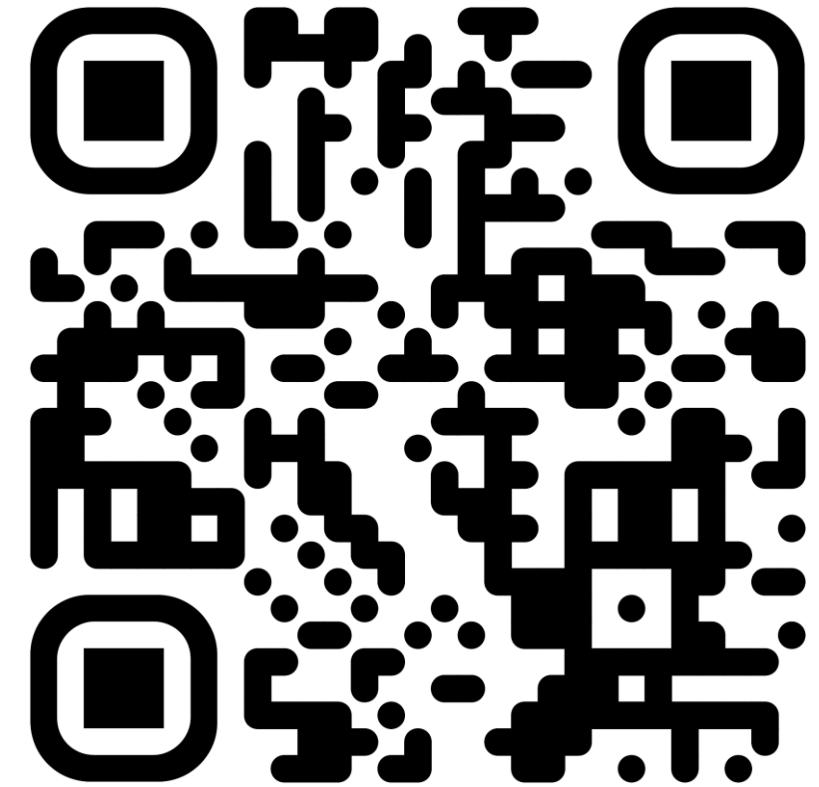
Agenda

- Me
- The caterpillar and the butterfly
- The case study
- The transformation
- Yeah, testing
- Tadaam
- Also, this
- Conclusion



Me

- Software Engineer @N47
- MSc in Intelligent Information Systems
- 10+ YoE
- Android.also {
 Python
 Java Spring Boot
}



#dc|x25

The caterpillar and the butterfly

SharedPreferences

- Added in API level 1
- Key-Value Storage
- Lightweight
- XML File Storage

```
class SimplePreferences(context: Context) {  
  
    private val prefs: SharedPreferences =  
        context.getSharedPreferences("my_prefs", Context.MODE_PRIVATE)  
  
    var isDarkTheme: Boolean  
        get() = prefs.getBoolean("is_dark_theme", true) // default = true  
        set(value) {  
            prefs.edit().putBoolean("is_dark_theme", value).apply()  
        }  
  
    var username: String?  
        get() = prefs.getString("username", null)  
        set(value) {  
            prefs.edit().putString("username", value).apply()  
        }  
}
```

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>  
└<map>  
    <string name="my_string">hello world!</string>  
    <int name="my_integer" value="124243" />  
    <boolean name="my_boolean" value="true" />  
└</map>
```

The caterpillar and the butterfly

```
implementation "androidx.datastore:datastore:1.0.0"
implementation "com.google.protobuf:protobuf-javalite:3.24.0"
```

```
val Context.dataStore by preferencesDataStore(name = "my_prefs")

class SimpleDataStore(context: Context) {

    private val ds = context.dataStore

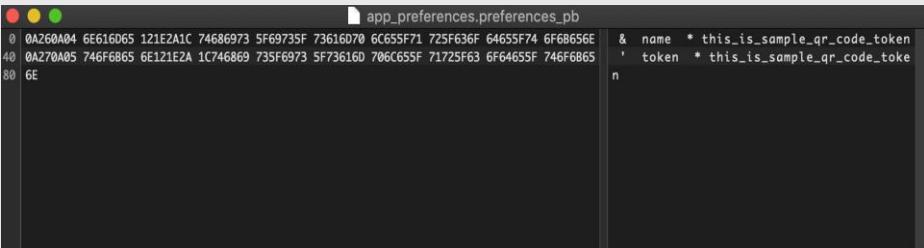
    companion object {
        private val IS_DARK_THEME = booleanPreferencesKey("is_dark_theme")
    }

    // Read value as Flow
    val isDarkTheme: Flow<Boolean> = ds.data.map { preferences ->
        preferences[IS_DARK_THEME] ?: true // default = true
    }

    // Write value
    suspend fun setDarkTheme(value: Boolean) {
        ds.edit { preferences ->
            preferences[IS_DARK_THEME] = value
        }
    }
}
```

DataStore

- part of Android Jetpack
- key-value pairs and/or custom data types
- **Preferences DataStore** stores and accesses data using keys
- **Preferences.Key<T>** is a type-safe key that you use to read and write values in a PreferencesDataStore.



```
app_preferences.preferences_pb
0A260A04 6E616065 121E2A1C 74686973 5F69735F 73616070 6C655F71 725F636F 64655F74 6F68656E
40 0A270A05 746F6865 6E121E2A 1C746869 735F6973 5F73616D 706C655F 71725F63 6F64655F 746F6865
80 6E
```

The caterpillar and the butterfly

Proto DataStore stores data as instances of a custom data type. This implementation requires you to define a schema using protocol buffers

Requires a **serializer** to read/write the protobuf to disk.

Supports atomic updates with `updateData { }` and flows for reactive reads.

```
proto

syntax = "proto3";

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

message UserPreferences {
    bool is_dark_theme = 1;
    string username = 2;
}
```

```
object UserPreferencesSerializer : Serializer<UserPreferences> {

    override val defaultValue: UserPreferences = UserPreferences.getDefaultInstance()

    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)
    }
}
```

The caterpillar and the butterfly

- PreferencesDataStore

```
val settingsDataStore: DataStore<Preferences> = PreferenceDataStoreFactory.create(  
    produceFile = { context.dataStoreFile("settings_prefs.preferences_pb") },  
    scope = CoroutineScope(Dispatchers.IO + SupervisorJob()),  
    migrations = listOf(  
        SharedPreferencesMigration(context, "old_shared_prefs")  
,  
        corruptionHandler = ReplaceFileCorruptionHandler(  
            produceNewData = { emptyPreferences() } // reset if corrupted  
)  
)
```

If you have content providers or app widgets

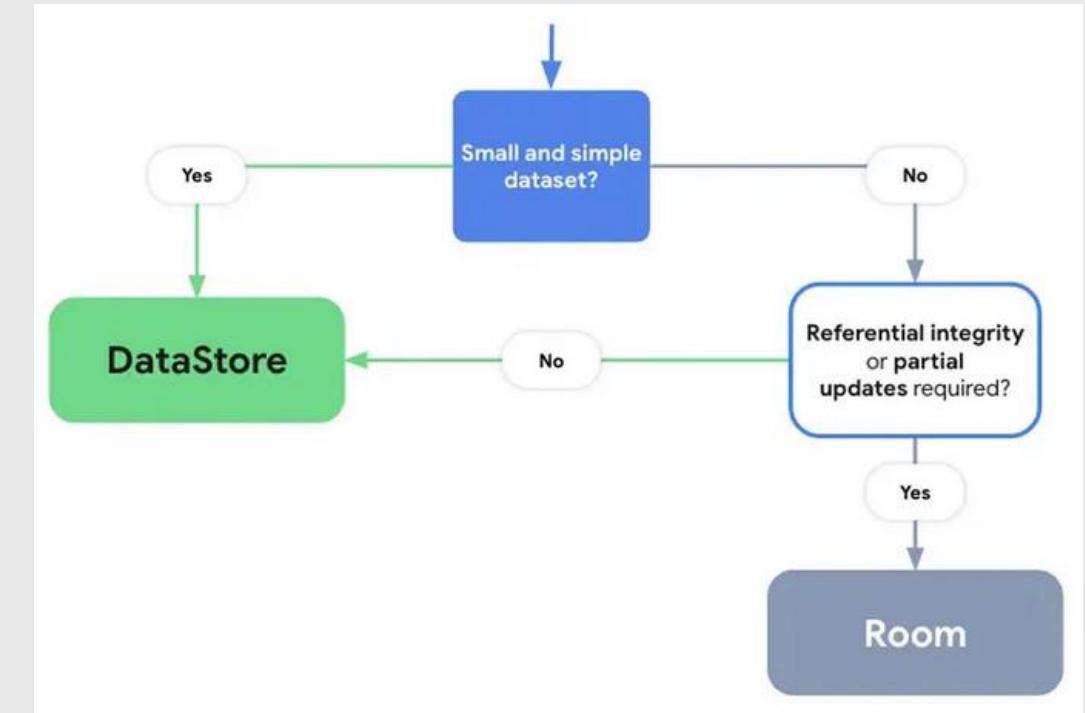
```
val settingsDataStore: DataStore<Preferences> = MultiProcessDataStoreFactory.create(  
    produceFile = { context.dataStoreFile("settings_prefs.preferences_pb") },  
    migrations = listOf(  
        SharedPreferencesMigration(context, "old_shared_prefs")  
,  
        corruptionHandler = ReplaceFileCorruptionHandler(  
            produceNewData = { emptyPreferences() } // reset if corrupted  
,  
        scope = CoroutineScope(Dispatchers.IO + SupervisorJob())  
)
```

- ProtoDataStore

```
val Context.userDataStore: DataStore<UserPreferences> by protoDataStore(  
    fileName = "user_prefs.pb",  
    serializer = UserPreferencesSerializer,  
    produceMigrations = { context ->  
        listOf(  
            SharedPreferencesMigration(  
                context,  
                sharedPreferencesName = "old_prefs",  
                keysToMigrate = listOf(  
                    booleanPreferencesKey("isDarkTheme"),  
                    stringPreferencesKey("user_json")  
,  
                    // optional: transform values if stored as JSON  
                )  
            )  
        )  
    }  
)
```

The caterpillar and the butterfly

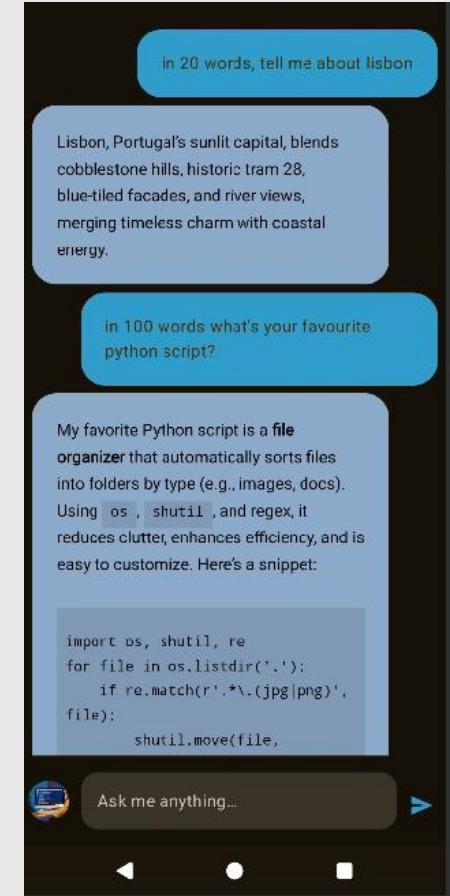
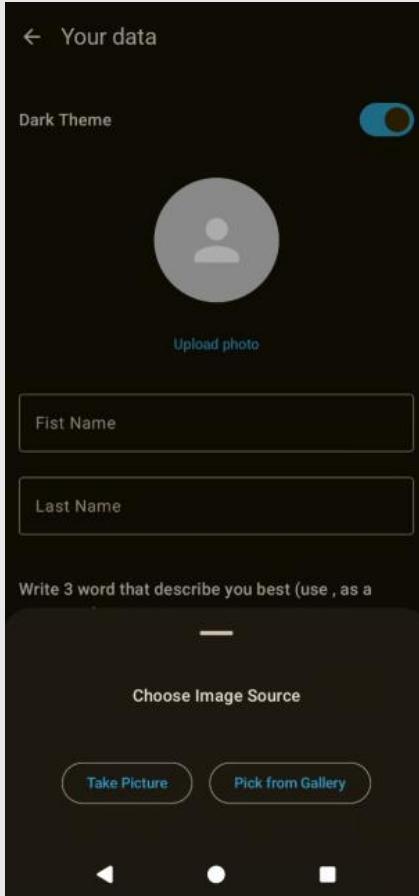
Feature	SharedPreferences	Preferences DataStore	Proto DataStore
Async API	✓ (only for reading changed values, via listener)	✓ (via Flow)	✓ (via Flow)
Synchronous API	✓ (but not safe to call on UI thread)	✗	✗
Safe to call on UI thread	✗*	✓ (work is moved to Dispatchers.IO under the hood)	✓ (work is moved to Dispatchers.IO under the hood)
Can signal errors	✗	✓	✓
Safe from runtime exceptions	✗**	✓	✓
Has a transactional API with strong consistency guarantees	✗	✓	✓
Handles data migration	✗	✓ (from SharedPreferences)	✓ (from SharedPreferences)
Type safety	✗	✗	✓ with Protocol Buffers



The case study

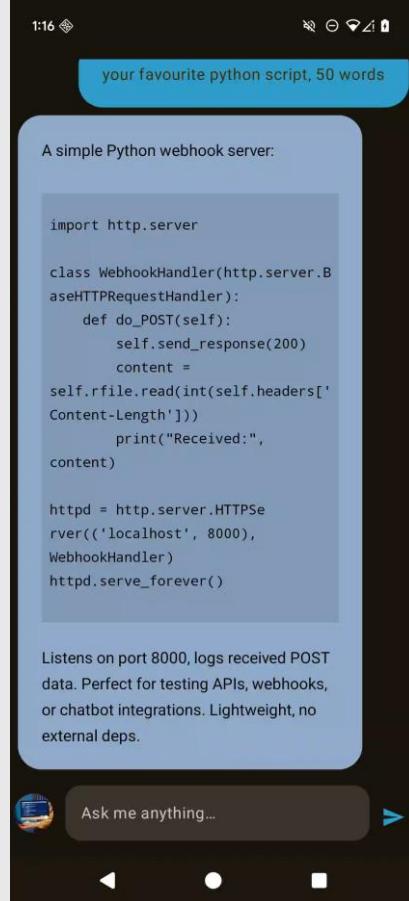


- develop_sp
- develop_ds



#dc|x25

The case study



#dc|x25

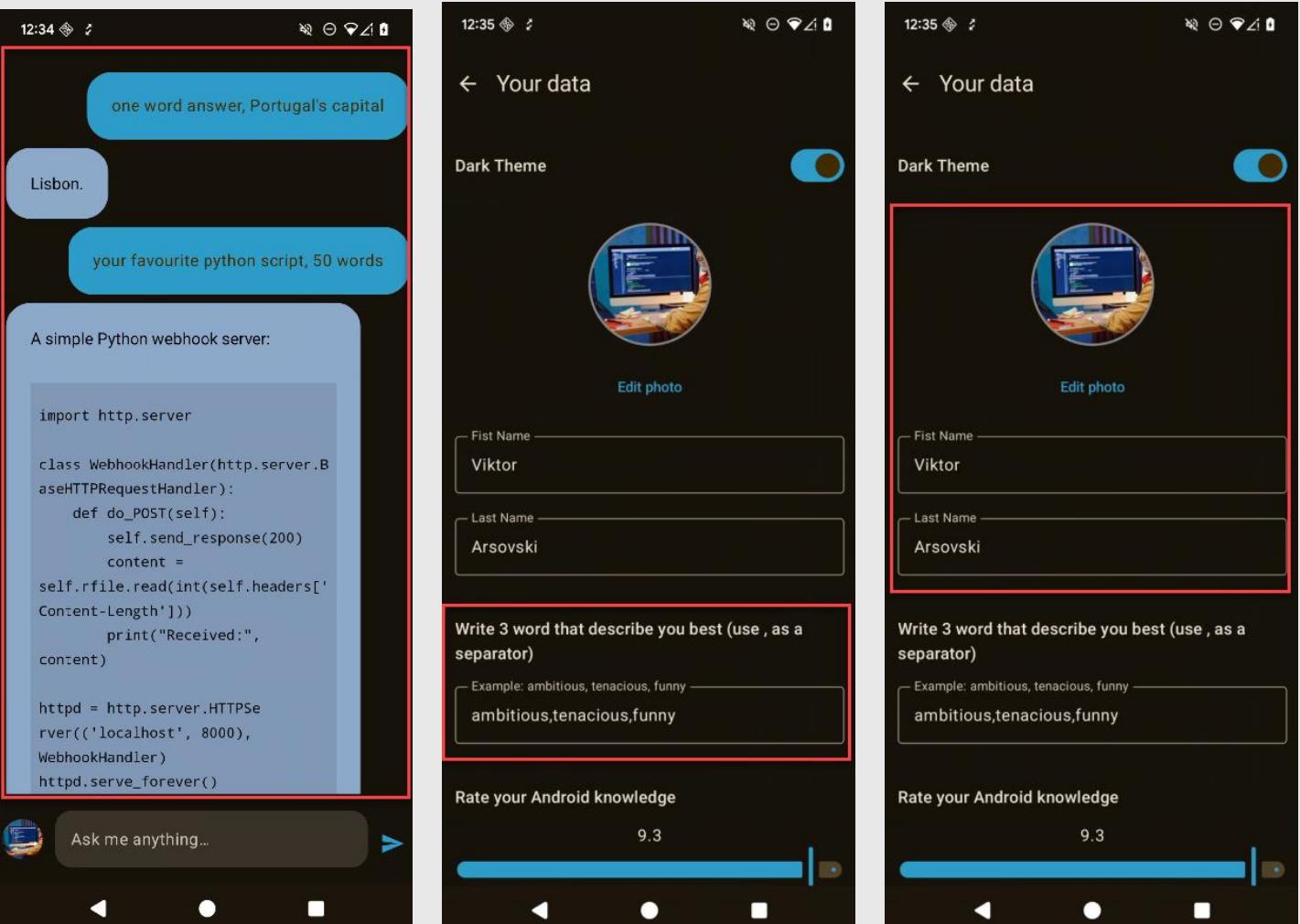
The case study

```
data class ChatMessage(val message: String, val messageType: MessageType)

enum class MessageType { 13 Usages
    ANSWER, 4 Usages
    QUESTION 4 Usages
}

fun MessageType.isAnswer() = this == MessageType.ANSWER 2 Usages
```

```
data class User( 12 Usages
    val firstName: String = "",
    val lastName: String = "",
    val picUri: String = ""
) {
    fun isValid() = firstName.isNotBlank() && 1 Usage
        lastName.isNotBlank() &&
        picUri.isNotBlank()
}
```



#dc|x25

The case study

- Unidirectional data flow (UDF) pattern
- SharedPreferencesManager

```
data class HomeScreenState( 4 Usages
    val hasBeenOnboarded: Boolean = false,
    val chatMessages: List<ChatMessage> = emptyList(),
    val timeOutTimestamp: String = "",
    val avatarUri: String = "",
    val errorMessage: String = "",
    val hasPopulatedData: Boolean = true,
    val isLoading: Boolean = false
)
```

```
data class DataScreenState( 7 Usages
    val user: User(),
    val description: List<String> = emptyList(),
    val androidRate: Float = 5.0f,
    val canClear: Boolean = false,
    val canSave: Boolean = false,
) {
    fun isValid() = 1 Usage
        user.picUri.isNotEmpty()
        && user.firstName.isNotEmpty()
        && user.lastName.isNotEmpty()
        && description.isNotEmpty()

    fun hasDataChangedFromDefault() = user.picUri.isNotEmpty() 1 Usage
        || user.firstName.isNotEmpty()
        || user.lastName.isNotEmpty()
        || description.isNotEmpty()
        || androidRate != DEFAULT_EXPERIENCE_LEVEL_VALUE
}

fun DataScreenState.hasDataChanged( 1 Usage
    u: User,
    desc: List<String>,
    rate: Float
) = user.picUri != u.picUri
    || user.firstName != u.firstName
    || user.lastName != u.lastName
    || description.toString() != desc.toString()
    || androidRate != rate
)
```

```
@Singleton 12 Usages
class Sp2DSharedPreferencesManager @Inject constructor(
    @ApplicationContext context: Context,
    private val moshi: Moshi
) {

    private val sharedPreferences = 17 Usages
        context.getSharedPreferences(p0 = SHARED_PREFERENCES_NAME, p1 = Context.MODE_PRIVATE)

    private val userAdapter = moshi.adapter(type = User::class.java) 2 Usages
    private val chatMessagesAdapter = moshi.adapter<List<ChatMessage>>() 2 Usages
        Types.newParameterizedType(
            rawType = List::class.java,
            ...typeArguments = ChatMessage::class.java
        )
    )

    var hasBeenOnboarded: Boolean 2 Usages
        get() = sharedPreferences.getBoolean(p0 = ONBOARDING_SHOWN_KEY, p1 = false)
        set(value) {...}

    var user: User
        get() {
            val json = sharedPreferences.getString(p0 = USER_KEY, p1 = null) ?: return User()
            return userAdapter.fromJson(string = json) ?: User()
        }
        set(value) {
            val json = userAdapter.toJson(value)
            sharedPreferences.edit { putString(p0 = USER_KEY, p1 = json) }
        }

    var threeWordDescription: List<String> 5 Usages
        get() {...}
        set(value) {
            val type = Types.newParameterizedType(rawType = List::class.java, ...typeArguments = String::class.java)
            val json = moshi.adapter<List<String>>(type).toJson(value)
            sharedPreferences.edit { putString(p0 = THREE_WORDS_DESCRIPTION_KEY, p1 = json) }
        }

    var isDarkTheme: Boolean 5 Usages
        get() = sharedPreferences.getBoolean(p0 = IS_DARK_THEME_KEY, p1 = true)
        set(value) {...}
}
```

The transformation

Mix of ProtoDataStore and PreferenceDataStore

Pros ✓

- Lightweight access – simple primitives in PreferencesDataStore are cheap to read/write.
- Modular – complex objects (User, ChatMessages) in proto, primitives separate.
- Better concurrency – small reads/writes don't interfere with larger proto transactions.
- Flexible schema evolution – changing proto messages doesn't affect primitive settings.

Cons ✗

- Split state – you have to manage consistency between stores manually.
- Migration complexity – you may need multiple migration steps.
- Partial backups – backing up one store doesn't include the other.
- Less atomicity – changing primitives and objects together isn't guaranteed to be transactional.



Single ProtoDataStore (everything in one proto)

Pros ✓

- Atomic updates (all properties saved together)
- Type-safe (no runtime cast issues)
- Single source of truth
- Easy backup/restore

Cons ✗

- Large file if many properties → harder to debug
- Schema changes can affect all data → risky migrations
- Tests and code tweaks are more intrusive
- Every primitive property now needs to be represented in the proto → testing and mocking become more involved
- Concurrency might be tricky if multiple flows update the same proto

The transformation

#dcIx25

- DI with Hilt

```
interface Sp2DsDataStore { 1 Implementation

    var hasProtoBeenMigrated: Boolean 2 Usages 1 Implementation
    var isOnboardingShownFlow: Flow<Boolean> 2 Usages 1 Implementation
    var userFlow: Flow<User> 7 Usages 1 Implementation
    var threeWordDescriptionFlow: Flow<List<String>> 6 Usages 1 Implementation
    var isDarkTheme: Boolean 5 Usages 1 Implementation
    var questionsLeft: Int 3 Usages 1 Implementation
    var timeoutTimestamp: String 8 Usages 1 Implementation
    var androidRate: Float 4 Usages 1 Implementation
    var chatMessagesFlow: Flow<List<ChatMessage>> 4 Usages 1 Implementation
    suspend fun hasStoredValidData(): Boolean 1 Usage 1 Implementation
    fun clearData(): Unit 1 Usage 1 Implementation
}
```

```
interface Sp2DsMigrator { 10 Usages 1 Implementation
    suspend fun migrateToProtoStore() 1 Usage 1 Implementation
}
```

```
@Keep 8 Usages
@Module
@InstallIn( ...value = SingletonComponent::class)
class ApplicationModule {

    @InternalCoroutinesApi 1 Usage
    @Provides
    fun provideCoroutineProvider(): CoroutineProvider = CoroutineProviderImpl()

    @Provides 1 Usage
    @Singleton
    fun provideDataStore(
        @ApplicationContext applicationContext: Context,
        coroutineProvider: CoroutineProvider
    ): DataStore<Preferences> = PreferenceDataStoreFactory.create(
        corruptionHandler = ReplaceFileCorruptionHandler(
            produceNewData = { emptyPreferences() }
        ),
        migrations = listOf(
            SharedPreferencesMigration(
                context = applicationContext,
                sharedPreferencesName = SHARED_PREFERENCES_NAME
            )
        ),
        scope = coroutineProvider.createCoroutineScope(),
        produceFile = { applicationContext.preferencesDataStoreFile( name = DATASTORE_NAME ) }
    )
}
```

The transformation

- `ReadWriteProperty`

```
interface ReadWriteProperty<in T, V> : ReadOnlyProperty<T, V>
```

[\(source\)](#)

Base interface that can be used for implementing property delegates of read-write properties.

This is provided only for convenience; you don't have to extend this interface as long as your property delegate has methods with the same signatures.

Since Kotlin
1.0

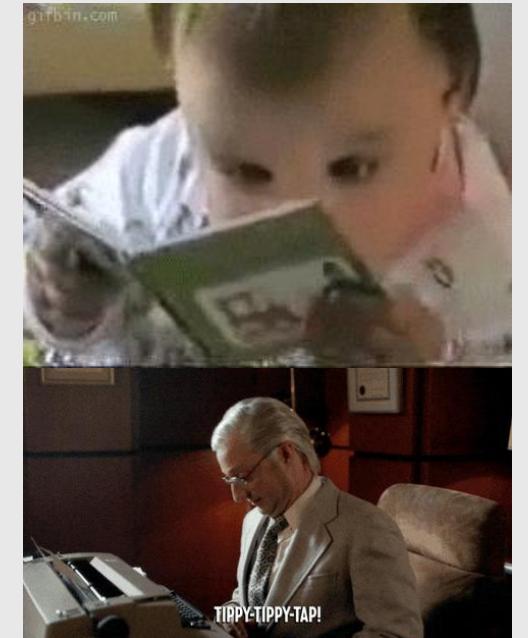
Parameters

T the type of object which owns the delegated property.

V the type of the property value.

```
abstract operator override fun getValue(thisRef: T, property: KProperty<*>): V
```

```
abstract operator fun setValue(thisRef: T, property: KProperty<*>, value: V)
```



The transformation

- Delegated property with blocking and with flow

```
class DataStoreProperty<T>{
    private val coroutineScope: CoroutineScope,
    private val dataStore: DataStore<Preferences>,
    private val key: Preferences.Key<T>,
    private val default: T
) : ReadWriteProperty<Any, T?> {

    override fun getValue(thisRef: Any, property: KProperty<*>): T = runBlocking {
        dataStore.data
            .map { preferences ->
                preferences[key] ?: default
            }.first()
    }

    override fun setValue(thisRef: Any, property: KProperty<*>, value: T?) {
        coroutineScope.launch {
            dataStore.edit { preferences ->
                value?.let {
                    preferences[key] = it
                }
            }
        }
    }
}
```

```
class DataStorePropertyFlow<T>(
    private val coroutineScope: CoroutineScope,
    private val dataStore: DataStore<Preferences>,
    private val key: Preferences.Key<T>,
    private val default: T
) : ReadWriteProperty<Any, Flow<T?>> {

    override fun getValue(thisRef: Any, property: KProperty<*>): Flow<T> =
        dataStore.data
            .map { preferences ->
                preferences[key] ?: default
            }

    override fun setValue(thisRef: Any, property: KProperty<*>, value: Flow<T?>) {
        coroutineScope.launch {
            dataStore.edit { preferences ->
                value.first()?.let {
                    preferences[key] = it
                }
            }
        }
    }
}
```



The transformation

- Let's optimize

#dcIx25

```
class SimplifiedDataStoreProperty<T>{ 2 Usages
    private val clazz: Class<T>,
    private val coroutineScope: CoroutineScope,
    private val dataStore: DataStore<Preferences>,
    private val keyString: String,
    private val default: T
) : ReadWriteProperty<Any, T?> {

    override fun getValue(thisRef: Any, property: KProperty<*>): T = runBlocking {
        dataStore.data.map { preferences ->
            Timber.d(
                message = "getValue() | migrateToProtoStore() keyString = $keyString, default = $default," +
                    "value = ${preferences[clazz.toPreferenceKey( key = keyString)]}"
            )
            preferences[clazz.toPreferenceKey( key = keyString)] ?: default
        }.first()
    }

    override fun setValue(thisRef: Any, property: KProperty<*>, value: T?) {
        coroutineScope.launch {
            dataStore.edit { preferences ->
                value?.let {
                    preferences[clazz.toPreferenceKey( key = keyString)] = it
                }
            }
        }
    }
}
```

```
@Suppress( ...names = "UNCHECKED_CAST") 5 Usages
fun <T> Class<T>.toPreferenceKey(key: String): Preferences.Key<T> {
    return when (this) {
        Boolean::class.java, java.lang.Boolean.TYPE -> booleanPreferencesKey( name = key) as Preferences.Key<T>
        Int::class.java, Integer.TYPE -> intPreferencesKey( name = key) as Preferences.Key<T>
        Long::class.java, java.lang.Long.TYPE -> longPreferencesKey( name = key) as Preferences.Key<T>
        String::class.java -> stringPreferencesKey( name = key) as Preferences.Key<T>
        Float::class.java, java.lang.Float.TYPE -> floatPreferencesKey( name = key) as Preferences.Key<T>
        Set::class.java, MutableSet::class.java -> stringSetPreferencesKey( name = key) as Preferences.Key<T>
        else -> throw IllegalArgumentException( s = "Unsupported type: ${this.simpleName}")
    }
}
```

```
class SimplifiedPropertyFlow<T>{ 2 Usages
    private val clazz: Class<T>,
    private val coroutineScope: CoroutineScope,
    private val dataStore: DataStore<Preferences>,
    private val keyString: String,
    private val default: T
) : ReadWriteProperty<Any, Flow<T?>> {

    override fun getValue(thisRef: Any, property: KProperty<*>): Flow<T> =
        dataStore.data
            .map { preferences ->
                preferences[clazz.toPreferenceKey( key = keyString)] ?: default
            }

    override fun setValue(thisRef: Any, property: KProperty<*>, value: Flow<T?>) {
        coroutineScope.launch {
            dataStore.edit { preferences ->
                value.first()?.let {
                    preferences[clazz.toPreferenceKey( key = keyString)] = it
                }
            }
        }
    }
}
```

The transformation

Let's make it easier to implement

```
inline fun <reified T> simplifiedDataStoreProperty( 5 Usages
    key: String,
    default: T,
    dataStore: DataStore<Preferences>,
    coroutineScope: CoroutineScope,
): SimplifiedDataStoreProperty<T> {
    return SimplifiedDataStoreProperty(
        clazz = when (T::class) {
            Boolean::class -> Boolean::class.java
            Int::class -> Int::class.java
            Long::class -> Long::class.java
            Float::class -> Float::class.java
            String::class -> String::class.java
            Set::class -> Set::class.java
            else -> throw IllegalArgumentException( s = "Unsupported type")
        } as Class<T>,
        coroutineScope = coroutineScope,
        dataStore = dataStore,
        keyString = key,
        default = default
    )
}
```

```
inline fun <reified T> simplifiedDataStorePropertyFlow( 1 Usage
    key: String,
    default: T,
    dataStore: DataStore<Preferences>,
    coroutineScope: CoroutineScope,
): SimplifiedPropertyFlow<T> {
    return SimplifiedPropertyFlow(
        clazz = when (T::class) {
            Boolean::class -> Boolean::class.java
            Int::class -> Int::class.java
            Long::class -> Long::class.java
            String::class -> String::class.java
            Set::class -> Set::class.java
            else -> throw IllegalArgumentException( s = "Unsupported type")
        } as Class<T>,
        coroutineScope = coroutineScope,
        dataStore = dataStore,
        keyString = key,
        default = default
    )
}
```

The transformation

- The implementation

```
class Sp2DsDataStoreManager @Inject constructor( 7 Usages
    @ApplicationContext val context: Context,
    coroutineProvider: CoroutineProvider,
    private val dataStore: DataStore<Preferences>
) : Sp2DsDataStore {

    val coroutineScope = coroutineProvider.createCoroutineScope() 10 Usages

    override var hasProtoBeenMigrated: Boolean by simplifiedDataStoreProperty(
        coroutineScope = coroutineScope,
        dataStore = dataStore,
        key = HAS_BEEN_MIGRATED_TO_PROTO_KEY,
        default = false
    )
    override var isOnboardingShownFlow: Flow<Boolean> by simplifiedDataStorePropertyFlow(
        coroutineScope = coroutineScope,
        dataStore = dataStore,
        key = ONBOARDING_SHOWN_KEY,
        default = false
    )
}
```

```
override var isDarkTheme: Boolean by simplifiedDataStoreProperty(
    coroutineScope = coroutineScope,
    dataStore = dataStore,
    key = IS_DARK_THEME_KEY,
    default = true
)

override var questionsLeft: Int by simplifiedDataStoreProperty(
    coroutineScope = coroutineScope,
    dataStore = dataStore,
    key = QUESTIONS_TO_ASK_KEY,
    default = DEFAULT_QUESTIONS_TO_ASK_VALUE
)

override var timeoutTimestamp: String by simplifiedDataStoreProperty(
    coroutineScope = coroutineScope,
    dataStore = dataStore,
    key = TIMEOUT_TIMESTAMP_KEY,
    default = ""
)
```

The transformation

- Protobuf

```
protobuf {
    protoc {
        artifact = "com.google.protobuf:protoc:3.25.1"
    }

    // Generates the java Protobuf-lite code for the Protobufs in this project. See
    // https://github.com/google/protobuf-gradle-plugin#customizing-protobuf-compilation
    // for more information.
    generateProtoTasks {
        all().forEach { task ->
            task.builtins {
                create(name = "java") {
                    option(option = "lite")
                }
            }
        }
    }
}
```

The screenshot shows the Android Studio project structure for a module named 'SharedPreferences2DataStore'. The 'app' folder contains 'src' and 'build' directories. Under 'src', there's an 'androidTest' folder and a 'main' folder which includes 'assets', 'java', and 'proto'. The 'proto' folder contains a file named 'sp2ds_data_store.proto'. A red arrow points from this file to the generated Java code on the right. The code is a snippet of a Java file generated from a protobuf definition, showing messages like 'UserProto' and 'MessageTypeProto'.

```
syntax = "proto3"; Viktor, 06.8.2025 19:35 • Initial commit

import "google/protobuf/wrappers.proto";

option java_package = "com.droidconlisbon.sp2ds.proto";
option java_multiple_files = true;

message ListOfString {
    repeated string item = 1;
}

message UserProto {
    string firstName = 1;
    string lastName = 2;
    string picUri = 3;
}

enum MessageTypeProto {
    MESSAGE_TYPE_UNSPECIFIED = 0;
    ANSWER = 1;
    QUESTION = 2;
}

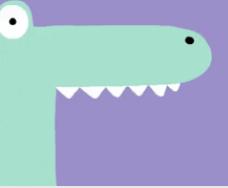
// Individual chat message
message ChatMessageProto {
    string message = 1;
    MessageTypeProto messageType = 2;
}

// Wrapper for list of chat messages
message ChatMessageList {
    repeated ChatMessageProto messages = 1;
}
```



#dc|x25

The transformation



- Model mapper extension methods

#dc|x25

```
fun ChatMessage.toProto(): ChatMessageProto { 2 Usages
    return ChatMessageProto.newBuilder()
        .setMessage(message)
        .setMessageType(
            when (messageType) {
                MessageType.ANSWER -> MessageTypeProto.ANSWER
                MessageType.QUESTION -> MessageTypeProto.QUESTION
            }
        )
        .build()
}

fun List<ChatMessage>.toProtoList(): List<ChatMessageProto> { 2 Usages
    return this.map { it.toProto() }
}

fun ChatMessageProto.toDomain(): ChatMessage { 3 Usages
    return ChatMessage(
        message = message,
        messageType = when (messageType) {
            MessageTypeProto.ANSWER -> MessageType.ANSWER
            MessageTypeProto.QUESTION -> MessageType.QUESTION
            else -> MessageType.ANSWER
        }
    )
}
```

```
fun User.toProto(): UserProto { 2 Usages
    return UserProto.newBuilder()
        .setFirstName(firstName)
        .setLastName(lastName)
        .setPicUri(picUri)
        .build()
}

fun UserProto.toDomain(): User { 3 Usages
    return User(
        firstName = firstName,
        lastName = lastName,
        picUri = picUri
    )
}
```

The transformation

- Proto Serializer and Delegate

```
private class UserSerializer : Serializer<UserProto> { 1 Usage
    override val defaultValue: UserProto = UserProto.getDefaultInstance()

    override suspend fun readFrom(input: InputStream): UserProto {
        return try {
            UserProto.parseFrom(input)
        } catch (exception: Exception) {
            throw CorruptionException( message = "Cannot read proto User", cause = exception)
        }
    }

    override suspend fun writeTo(t: UserProto, output: OutputStream) {
        t.writeTo(output)
    }
}
```

```
class UserDataStorePropertyFlow( 2 Usages
    val context: Context,
    val dataStoreName: String,
    private val coroutineScope: CoroutineScope
) : ReadWriteProperty<Any, Flow<User?>> {

    private val Context.userDataStore: DataStore<UserProto> by dataStore( 1 Usage
        fileName = dataStoreName.toProtoStoreName(),
        serializer = UserSerializer(),
    )

    private val protoStore: DataStore<UserProto> 2 Usages
        get() = context.userDataStore

    override fun getValue(thisRef: Any, property: KProperty<*>): Flow<User> =
        protoStore.data.map { it.toDomain() }

    override fun setValue(thisRef: Any, property: KProperty<*>, value: Flow<User?>) {
        coroutineScope.launch( context = Dispatchers.IO ) {
            val user = value.firstOrNull() ?: User()
            protoStore.updateData {
                user.toProto()
            }
        }
    }
}
```

```
override var userFlow: Flow<User> by UserDataStorePropertyFlow( 2 Usages
    context = context,
    dataStoreName = USER_KEY,
    coroutineScope = coroutineScope
)
```

The transformation

- Proto Serializer and Delegate

```
private class ChatMessagesListSerializer : Serializer<List<ChatMessageProto>> { 1 Usage
    override val defaultValue: List<ChatMessageProto>
        get() = emptyList()

    override suspend fun readFrom(input: InputStream): List<ChatMessageProto> {
        try {
            val chatMessagesList = ChatMessageList.parseFrom(input)
            return chatMessagesList.messagesList
        } catch (exception: Exception) {
            throw CorruptionException(message = "Cannot read proto.", cause = exception)
        }
    }

    override suspend fun writeTo(
        t: List<ChatMessageProto>,
        output: OutputStream
    ) {
        val chatMessagesList = ChatMessageList.newBuilder()
            .addAllMessages(values = t)
            .build()
        chatMessagesList.writeTo(output)
    }
}
```

```
class ChatMessagesListDataStorePropertyFlow( 2 Usages
    private val context: Context,
    dataStoreName: String,
    private val coroutineScope: CoroutineScope
) : ReadWriteProperty<Any, Flow<List<ChatMessage>>> {

    private val Context.chatMessagesListDataStore: DataStore<List<ChatMessageProto>> by dataStore(
        fileName = dataStoreName.toProtoStoreName(),
        serializer = ChatMessagesListSerializer(),
    )

    private val protoStore: DataStore<List<ChatMessageProto>> 2 Usages
        get() = context.chatMessagesListDataStore

    override fun getValue(thisRef: Any, property: KProperty<*>): Flow<List<ChatMessage>> =
        protoStore.data.map { protoList ->
            protoList.map { proto ->
                proto.toDomain()
            }
        }

    override fun setValue(thisRef: Any, property: KProperty<*>, value: Flow<List<ChatMessage>>) {
        coroutineScope.launch(context = Dispatchers.IO) {
            val messages = value.firstOrNull() ?: emptyList()
            protoStore.updateData {
                messages.toProtoList()
            }
        }
    }
}
```

```
override var chatMessagesFlow: Flow<List<ChatMessage>> by ChatMessagesListDataStorePropertyFlow( 1 Usage
    context = context,
    dataStoreName = CHAT_MESSAGES_LIST_KEY,
    coroutineScope = coroutineScope
)
```

The transformation

- Manual Proto migration - implementation

```
interface Sp2DsMigrator { 10 Usages 1 Implementation
    suspend fun migrateToProtoStore() 1 Usage 1 Implementation
}
```

```
class Sp2DsMigratorManager @Inject constructor( 6 Usages
    @ApplicationContext val context: Context,
    private val sp2DsDataStore: Sp2DsDataStore,
    private val dataStore: DataStore<Preferences>,
    moshi: Moshi
) : Sp2DsMigrator {

    private val userAdapter = moshi.adapter( type = User::class.java) 1 Usage
    private val chatMessagesAdapter = moshi.adapter<List<ChatMessage>>(
        Types.newParameterizedType(
            rawType = List::class.java,
            ...typeArguments = ChatMessage::class.java
        )
    )
    private val listOfStringsAdapter = moshi.adapter<List<String>>(
        Types.newParameterizedType(
            rawType = List::class.java,
            ...typeArguments = String::class.java
        )
    )

    override suspend fun migrateToProtoStore() {
        coroutineScope {
            val jobs = listOf(
                async {
                    migrateProperty( key = USER_KEY) { jsonValue ->
                        val userValue = if (jsonValue.isNotEmpty()) {
                            userAdapter.fromJson( string = jsonValue ) ?: User()
                        } else {
                            User()
                        }
                        sp2DsDataStore.userFlow = flowOf(userValue)
                    },
                    async {
                        migrateProperty( key = CHAT_MESSAGES_LIST_KEY) { jsonValue ->
                            val messages = if (jsonValue.isNotEmpty()) {
                                chatMessagesAdapter.fromJson( string = jsonValue ) ?: emptyList()
                            } else {
                                emptyList()
                            }
                            sp2DsDataStore.chatMessagesFlow = flowOf( value = messages )
                        },
                        async {
                            migrateProperty( key = THREE_WORDS_DESCRIPTION_KEY) { jsonValue ->
                                val list = if (jsonValue.isNotEmpty()) {
                                    listOfStringsAdapter.fromJson( string = jsonValue ) ?: emptyList()
                                } else {
                                    emptyList()
                                }
                                sp2DsDataStore.threeWordDescriptionFlow = flowOf( value = list )
                            }
                        }
                    }.jobs.awaitAll()
                }
            )
        }
    }
}
```

The transformation



- Manual Proto migration – trigger the migration

#dc|x25

```
@HiltAndroidApp 9 Usages
class Sp2DsApplication : Application() {
    @Inject 3 Usages
    lateinit var sp2DsDataStore: Sp2DsDataStore

    @Inject 2 Usages
    lateinit var sp2DsMigrator: Sp2DsMigrator

    @Inject 2 Usages
    lateinit var coroutineProvider: CoroutineProvider

    private val scope: CoroutineScope 1 Usage
        get() = coroutineProvider.createCoroutineScope()

    override fun onCreate() {
        super.onCreate()
        if (BuildConfig.DEBUG) {
            Timber.plant( tree = Timber.DebugTree())
        }
        migrateProto()
    }

    private fun migrateProto() { 1 Usage
        if (sp2DsDataStore.hasProtoBeenMigrated) {
            return
        }
        scope.launch( context = Dispatchers.IO) {
            sp2DsMigrator.migrateToProtoStore()
            sp2DsDataStore.hasProtoBeenMigrated = true
        }
    }
}
```

The transformation

- Keep the same pattern
- Accommodate for the load the params in the data state object



```
@HiltViewModel 14 Usages
class DataViewModel @Inject constructor(
    private val sp2DataStore: Sp2DsDataStore
) : IDataViewModel() {

    private val _dataScreenStateFlow = MutableStateFlow( value = DataScreenState() ) 9 Usages
    override val dataScreenStateFlow = _dataScreenStateFlow.asStateFlow()

    init {
        viewModelScope.launch {
            with( receiver = sp2DataStore ) {
                val pair = combine(
                    sp2DataStore.userFlow,
                    flow2 = sp2DataStore.threeWordDescriptionFlow,
                ) { user, description ->
                    Pair( first = user, second = description )
                }.first()
                val baseState = DataScreenState(
                    user = pair.first,
                    description = pair.second,
                    androidRate = androidRate
                )
                val canSave = calculateCanSave(baseState)
                val canClear = baseState.hasDataChangedFromDefault()
                _dataScreenStateFlow.emit(
                    value = _dataScreenStateFlow.value.copy(
                        user = pair.first,
                        description = pair.second,
                        androidRate = androidRate,
                        isInitialized = true,
                        canSave = canSave,
                        canClear = canClear
                    )
                )
            }
        }
    }
}
```

```
@OptIn( ...markerClass = ExperimentalCoroutinesApi::class)
class ThemeViewModelTest {

    @MockK 3 Usages
    private lateinit var mockSp2DsStore: Sp2DsDataStore

    private lateinit var viewModel: ThemeViewModel 3 Usages

    @Before
    fun setUp() {
        MockKAnnotations.init( ...obj = this, relaxed = true)
        with( receiver = mockSp2DsStore) {
            every { isDarkTheme } returns false
            every { isDarkTheme = any() } just Runs
        }
        viewModel = ThemeViewModel( sp2DataStore = mockSp2DsStore)
    }

    @Test
    fun `initial themeState matches DataStore value`() = runTest {
        viewModel.isDarkThemeStateFlow.test {
            val initial = awaitItem()
            assertEquals( expected = false, actual = initial)
            cancelAndIgnoreRemainingEvents()
        }
    }
}
```

```
@OptIn( ...markerClass = ExperimentalCoroutinesApi::class)
class ThemeViewModelTest {

    @MockK 3 Usages
    private lateinit var mockSharedPreferences: Sp2DsSharedPreferencesManager

    private lateinit var viewModel: ThemeViewModel 3 Usages

    @Before
    fun setUp() {
        MockKAnnotations.init( ...obj = this)
        with( receiver = mockSharedPreferences){
            every { isDarkTheme } returns false
            every { isDarkTheme = any() } just Runs
        }
        viewModel = ThemeViewModel( sharedPreferencesManager = mockSharedPreferences)
    }

    @Test
    fun `initial themeState matches SharedPreferences value`() = runBlocking {
        viewModel.isDarkThemeStateFlow.test {
            val initial = awaitItem()
            assertEquals( expected = false, actual = initial) // initial value matches mocked getter
            cancelAndIgnoreRemainingEvents()
        }
    }
}
```

Yeah, testing



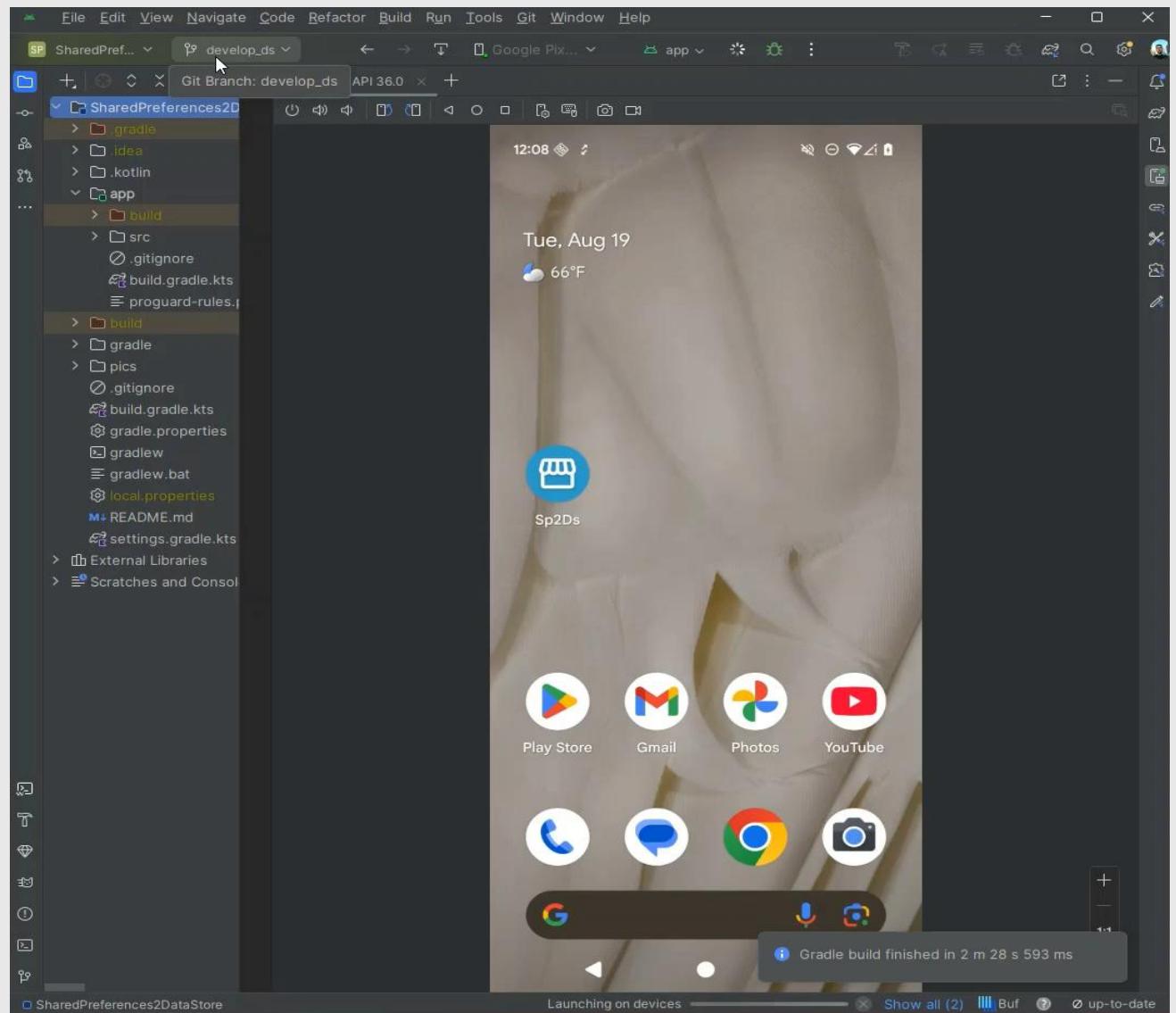
#dcIx25



Tadaam

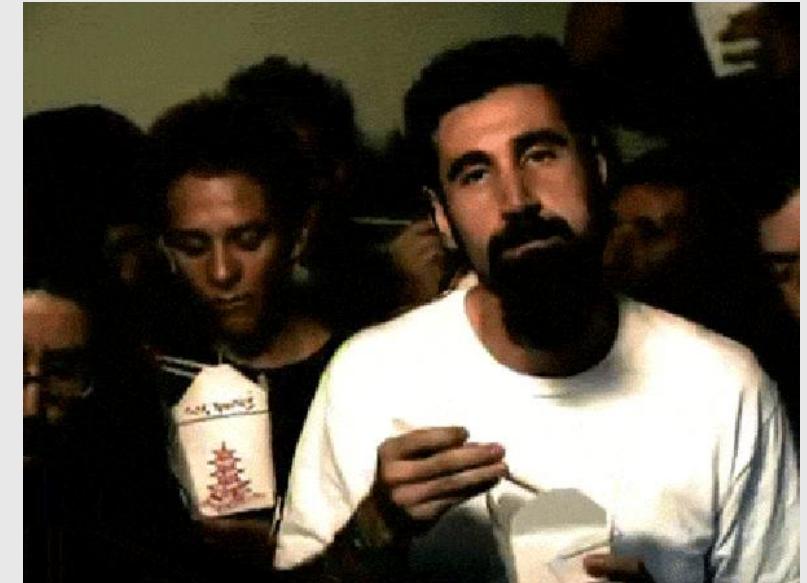


#dc|x25



Also, this

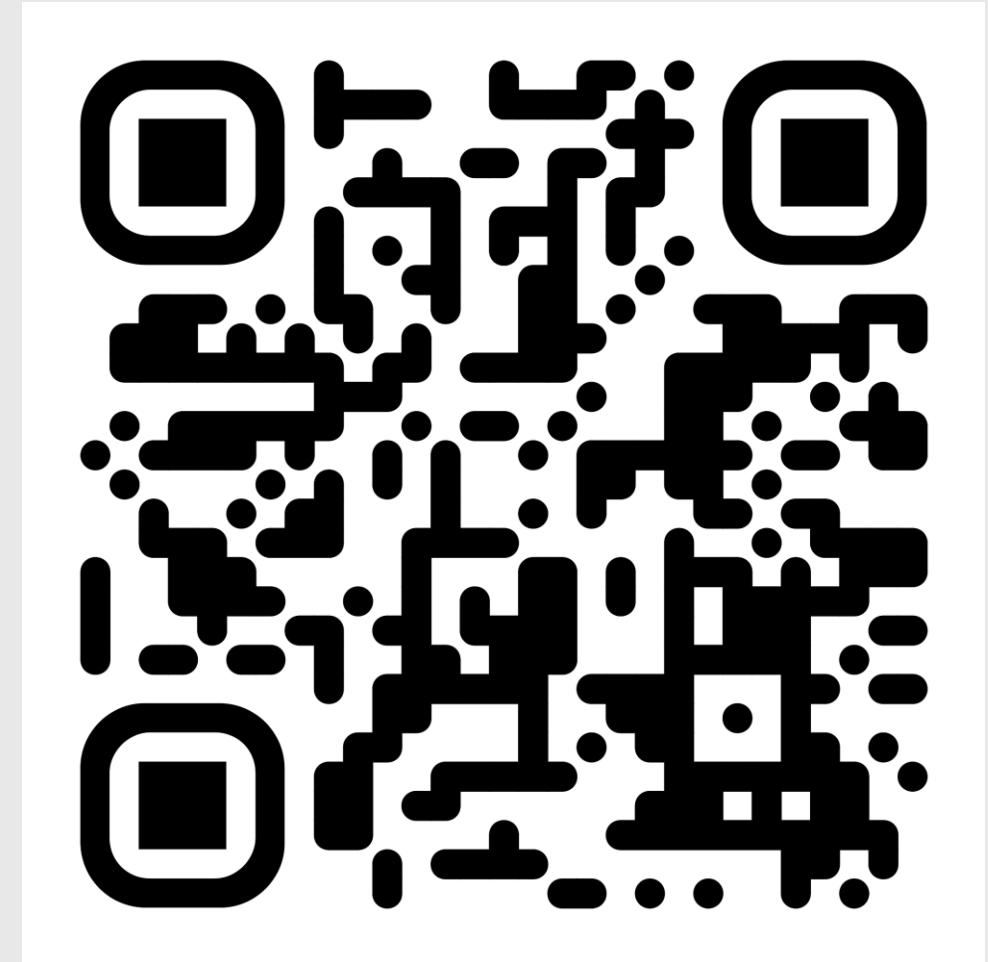
- Android Jetpack DataStore throws `java.lang.NoClassDefFoundError` for `proto.ListOfStrings` only in instrumentation tests
- DI, DataStore and UI tests
- Multiple SharedPreferences files



#dcIx25

Conclusion

- In complex legacy solutions there's no silver bullet
- The art of finding a perfect solution: knowing what's possible and what suits your implementation
- Be prepared for some trial&error and code tweaks along the way
- lots of independent primitives + a few complex objects -> mixed stores
- a tightly-coupled set of objects that always need to be saved together -> single ProtoStore



#dc|x25

References

- [1] Android Developers. *DataStore*. Available: <https://developer.android.com/jetpack/androidx/releases/datastore>. [Accessed: 16-Aug-2025].
- [2] S. Milanović, *Introduction to Jetpack DataStore*. Available: <https://tinyurl.com/2btr3he2>. [Accessed: 16-Aug-2025].
- [3] I. Dimitrov, *SharedPreferences vs. DataStore: Evolving Data Management*. Available: <https://tinyurl.com/29jknr32>. [Accessed: 16-Aug-2025].
- [4] A. Dyas, *Migrating SharedPreferences with Jetpack DataStore*. Available: <https://tinyurl.com/2a97qoxr>. [Accessed: 16-Aug-2025].
- [5] F. Muntenescu and R. Sathyanarayana, *Prefer Storing Data with Jetpack DataStore*. Available: <https://tinyurl.com/264zyo6c>. [Accessed: 16-Aug-2025].