Application Development II

420-5A6-AB

Instructor: Talib Hussain

Day 23:

Flows, ViewModels and DataStore: Persisting Data

Persist Data

- So far, we've been able to store information in our viewModel and display it as it changes.
- But, what happens if you re-run the application?
  - Gone!
- We would like to be able to persist data.
- One approach is to store the information on the device.
- Android offers the DataStore to do this.
- Two types:
  - Preferences DataStore – simple key-value pairs
  - Proto DataStore – more complex data

- Let's try the Preferences DataStore
  - See the "persistData" branch in Teacher's code

DataStore

- Note: Many of these examples use automated Dependency Injection (typically with Hilt).
- Simple walkthrough: https://medium.com/jetpack-composers/android-jetpack-datastore-5dfdfea4a3ea
- https://developer.android.com/courses/pathways/android-basics-compose-unit-6-pathway-3
- https://developer.android.com/codelabs/android-preferences-datastore#7
- https://medium.com/androiddevelopers/all-about-preferences-datastore-cc7995679334
- https://developer.android.com/topic/libraries/architecture/datastore?gclid=CjwKCAiA55mPBhBOEiwANmzoQtX8aFaxx5WFTDOpYVN429tF3U8X3BnZu8ZMfJhRqGtyme_PzaypHhoCQDsQAvD_BwE&gclsrc=aw.ds#datastore-typed

- https://android-developers.googleblog.com/2020/09/prefer-storing-data-with-jetpack.html
  - Nice walkthrough
- Room:
  - https://developer.android.com/courses/pathways/android-basics-compose-unit-6-pathway-2

- Firebase:
  - https://firebase.google.com/codelabs/build-android-app-with-firebase-compose#2

DataStore = Asynchronous Local Data Storage

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

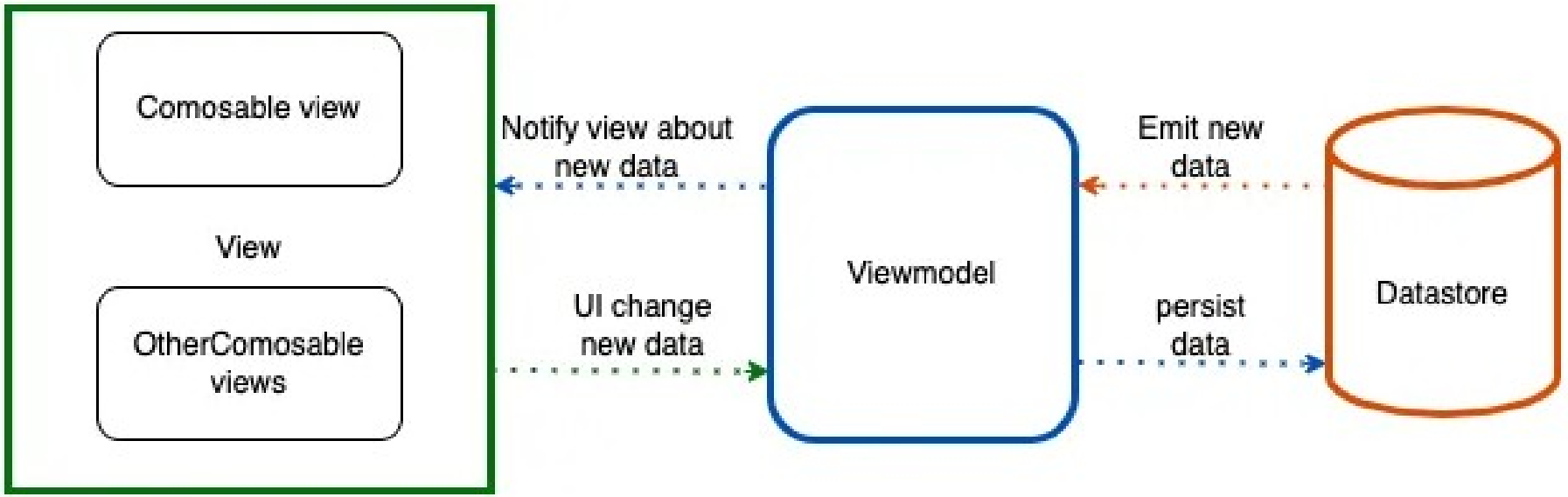
Preferences DataStore

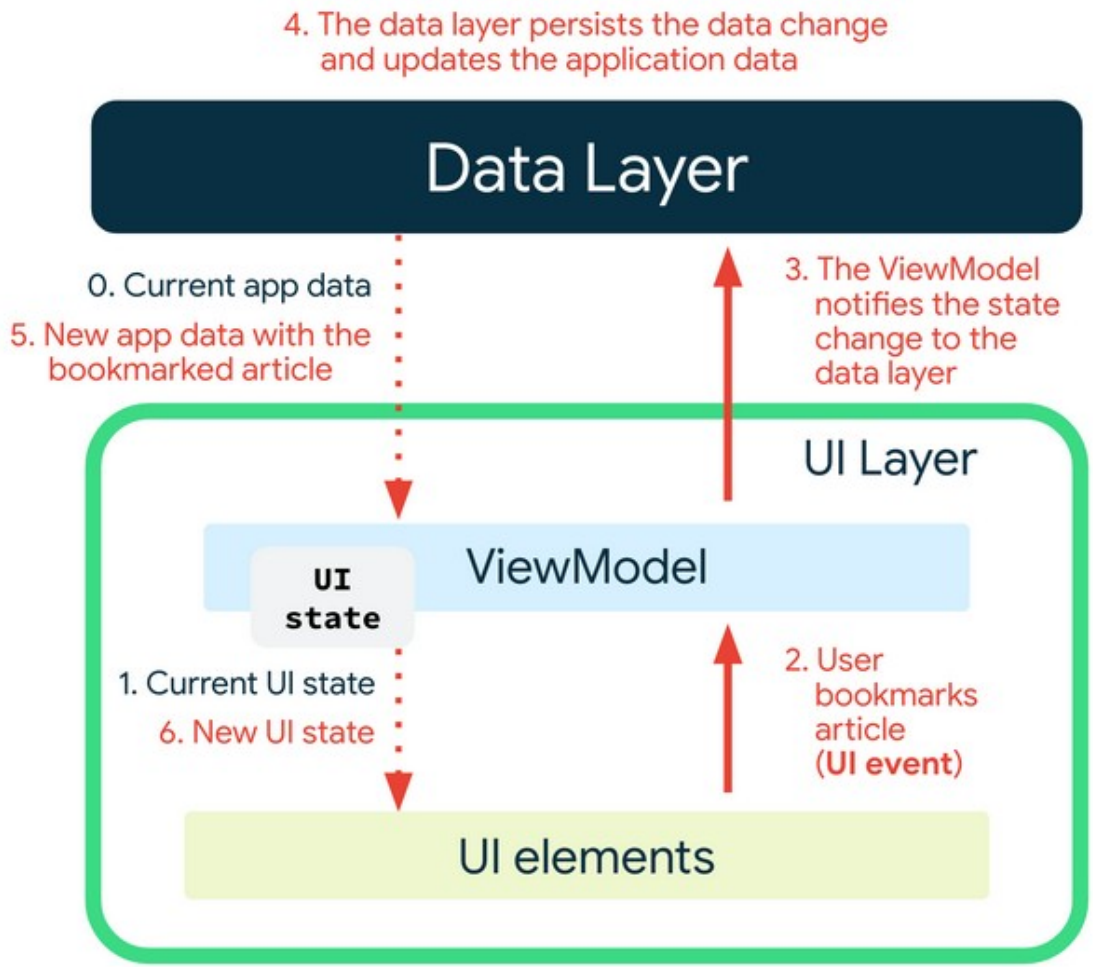
- Preferences DataStore stores and accesses data using key-value pairs
  - Useful for a small dataset
- This implementation does not require a predefined schema
- It does not provide type safety.
- DataStore uses Kotlin coroutines and Flow to store data asynchronously, consistently, and transactionally.
  - It directly avoids doing any work that blocks the UI thread

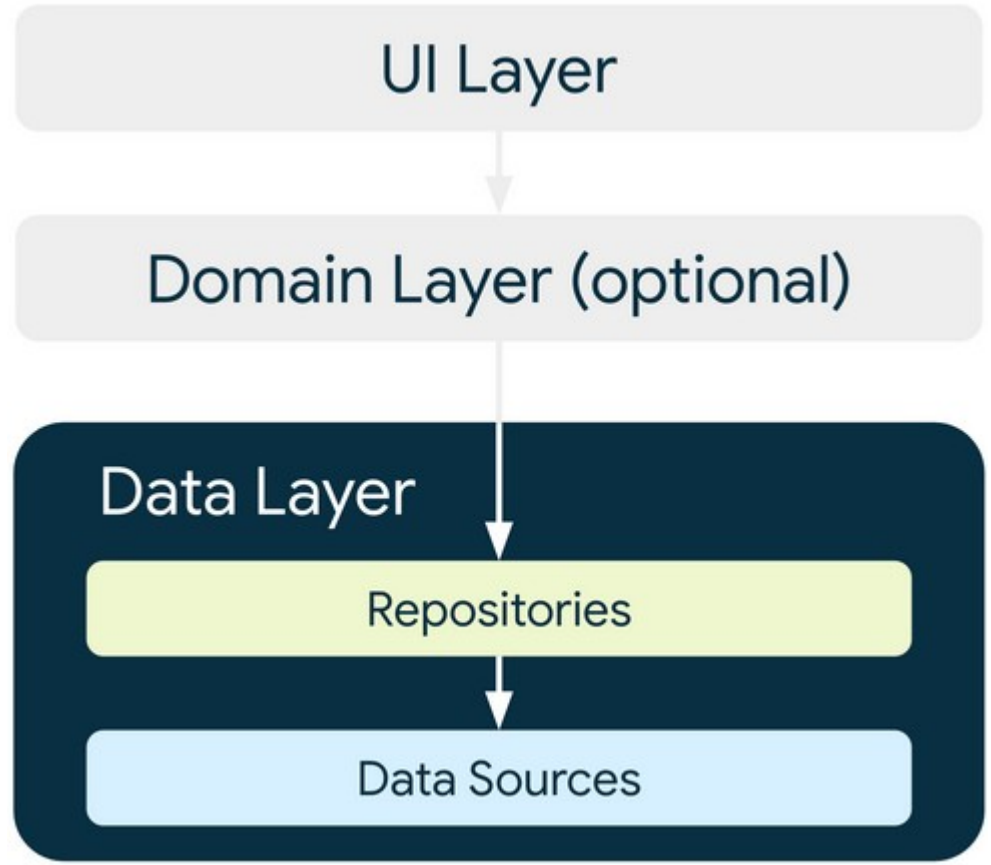
- https://developer.android.com/topic/libraries/architecture/datastore

Separation of Concerns

- The idea is to separate the UI from the data storage, using the ViewModel as an intermediary to hide the specific details of the data layer.
  - https://betterprogramming.pub/2-way-flow-using-jetpack-compose-and-datastore-36305301347d?gi=034df17be5a4




4. The data layer persists the data change and updates the application data
Data Layer
0. Current app data
5. New app data with the bookmarked article
3. The ViewModel notifies the state change to the data layer
UI Layer
ViewModel
UI state
1. Current UI state
6. New UI state
2. User bookmarks article (UI event)
UI elements


UI Layer
Domain Layer (optional)
Data Layer
Repositories
Data Sources

4. Repository-based approach using Datastore

- We will create a Repository interface for persisting our ProfileData
  - Save our data, get our data, clear our data
- We will create a Repository implementation that is backed by a Preferences DataStore
  - Save our data to file, get our data from file, clear the data in the file
- We will create a ViewModel that uses the repository to get/update the profile data
- We will create a Screen that uses that ViewModel

4. Repository Interface

- We need asynchronous-based operations. Thus, the use of suspend and Flows.

```
interface ProfileRepository {
    suspend fun saveProfile(profileData: ProfileData)
    fun getProfile(): Flow<ProfileData>
    suspend fun clear()
}
```

4. Using Preferences DataStore

- In build.gradle, add the following dependency
  - implementation("androidx.datastore:datastore-preferences:1.0.0")
- Create a DataStore file containing a DataStore class. The data store needs to accept a Context as a parameter.
- A given DataStore is saved using a filename (e.g., profile_datastore)

```
const val PROFILE_DATASTORE ="profile_datastore"
private val Context.dataStore: DataStore<Preferences> by preferencesDataStore(name = PROFILE_DATASTORE)


class ProfileRepositoryDataStore (private val context: Context) : ProfileRepository  {
```

- The set of information that will be stored in the DataStore are indicated using "preferencesKeys". For convenience, these can be defined in a companion object.

```
companion object {
    val NAME = stringPreferencesKey("NAME")
    val COUNTER = intPreferencesKey("COUNTER")
  }
```

- A preferences datastore provides an edit() function that transactionally updates the data in a DataStore

```
/** Update the values in the DataStore. */
  override suspend fun saveProfile(profileData: ProfileData) {
    context.dataStore.edit {
      it[NAME] = profileData.name
      it[COUNTER] = profileData.counter
    }
  }
```

- We read information from the datastore as a Flow.

```
Override fun getProfile(): Flow<ProfileData> = context.dataStore.data.map {
    ProfileData(
      name = it[NAME] ?: "",
      counter = it[COUNTER] ?: 0
    )
  }
```

- Other operations include clear()

```
 override suspend fun clear() {
    context.dataStore.edit {
      it.clear()
    }
```

DataStore

```
// Preferences DataStore: It stores and accesses data using keys.
// This implementation does not require a predefined schema
//   and does not provide type safety.
const val PROFILE_DATASTORE ="profile_datastore"
private val Context.dataStore: DataStore<Preferences> by preferencesDataStore(name = PROFILE_DATASTORE)
class ProfileRepositoryDataStore (private val context: Context) : ProfileRepository  {
    companion object {
        val NAME = stringPreferencesKey("NAME")
        val COUNTER = intPreferencesKey("COUNTER")
    }

    /** Update the values in the DataStore. */
    override suspend fun saveProfile(profileData: ProfileData) {
        context.dataStore.edit {
            it[NAME] = profileData.name
            it[COUNTER] = profileData.counter
        }
    }
```

```
    /** Get the data in the DataStore as a flow.  Since the store may have never
     *     been used yet, handle the null case with default values. */
    override fun getProfile(): Flow<ProfileData> = context.dataStore.data.map {
        ProfileData(
            name = it[NAME] ?: "",
            counter = it[COUNTER] ?: 0
        )
    }

    override suspend fun clear() {
        context.dataStore.edit {
            it.clear()
        }
    }

}
```

ViewModel backed by DataStore

```
/** Simple view model that keeps track of a single value (count in this case) */

class MyViewModelSimpleSaved(private val profileRepository: ProfileRepository) :
ViewModel() {

    // private UI state (MutableStateFlow)

    private val _uiState = MutableStateFlow(ProfileData())

    // public getter for the state (StateFlow)

    val uiState: StateFlow<ProfileData> = _uiState.asStateFlow()


    /* Method called when ViewModel is first created */

    init {

        // Start collecting the data from the data store when the ViewModel is created.

        viewModelScope.launch {

            profileRepository.getProfile()collect { profileData ->

                _uiState.value = profileData

            }

        }

    }
```

```
fun setName(newName: String) {

    viewModelScope.launch {

        _uiState.update { it.copy(name = newName) }

        profileRepository.saveProfile (_uiState.value)

    }

}
```

```
/* Increments the value of the counter stored in the state flow */

fun increment() {

    viewModelScope.launch {

        var count = _uiState.value.counter;

        _uiState.update { currentState ->

            currentState.copy(counter = count + 1)

        }

        profileRepository.saveProfile (_uiState.value)

    }

}


}
```

Only One Instance!

- Never create more than one instance of DataStore for a given file in the same process.
- Doing so can break all DataStore functionality.
  - If there are multiple DataStores active for a given file in the same process, DataStore will throw IllegalStateException when reading or updating data.
- This means that we need to "inject" a single DataStore object into our ViewModel.
  - As the application is recomposed and reconfigured, we don't want that object to change
  - This is called Dependency Injection
  - Recall: We did a simplified example of dependency injection with our normal and test databases in the Web 2 course.

Manual Dependency Injection

- This is a good video that walks through manual injection.
  - Note: Hard to find good online descriptions of manual dependency injection in Compose.
  - Automated injection using tools like Hilt can be very confusing to learn…
  - https://www.youtube.com/watch?v=eX-y0IEHJjM
- There are a few steps we need to take to ensure that the object we want to inject is created only once

- First, we create a new "AppModule" file.

```
/** This module provides the specific object(s) we will inject */
class AppModule(
    private val appContext: Context
) {
    /* Create appropriate repository (backed by a DataStore) on first use.
       Only one copy will be created during lifetime of the application. */
    val profileRepository : ProfileRepository by lazy {
        ProfileRepositoryDataStore(appContext)
    }
}
```

"App"

- Create new class at root level called MyApp

```
/** This file allows us to provide a single ("static") module that can be accessed
 *   everywhere in the code, and in turn provide the specific (singleton) objects we will inject.
 */
class MyApp: Application() {

    /* Always be able to access the module ("static") */
    companion object {
        lateinit var appModule: AppModule
    }

    /* Called only once at beginning of application's lifetime */
    override fun onCreate() {
        super.onCreate()
        appModule = AppModule(this)
    }
}
```

- Also, add "MyApp" to manifest (AndroidManifest.xml)

```
<application
    android:name=".MyApp"
    ...
```

ViewModel Factory

- The viewModel() function that we have been using in the constructor calls to our Screens does an important job – it creates a viewModel of the right type when first accessed, and always maps to that single viewModel when the composable is recreated/recomposed.
  - But, it does not "accept parameters" directly.
- We need to use a "factory" to create the view model with parameters.
  - We need the parameter since this is the "injection" of the DataStore that will back our ViewModel.

```
/* ViewModel Factory that will create our view model by injecting the
    ProfileDataStore from the module.
 */
class MyViewModelSimpleSavedFactory : ViewModelProvider.Factory {
   override fun <T : ViewModel> create(modelClass: Class<T>): T {
      return MyViewModelSimpleSaved(MyApp.appModule.profileRepository) as T
   }
}
```

- You can put this in its own class or in the same file as the MyViewModelSimpleSaved viewmodel.

- Regardless of whether you use viewModel() or with Hilt hiltViewModel() to retrieve your ViewModel, they both will call onCleared() when the NavHost finishes transitioning to a different route
- So whenever you navigate to another Composable, it will be cleaned up. This is achieved by defining a DisposableEffect on the navigation route when the NavHost is created and you can mimic the behavior even if you're not using the navigation library and need to clean up the ViewModel yourself.
  - https://www.droidcon.com/2021/10/25/jetpack-compose-navigation-architecture-with-viewmodels/

- https://olshevski.github.io/compose-navigation-reimagined/shared-view-models/

Sharing Views Across Navigation

- https://www.youtube.com/watch?v=FIEnIBq7Ups
- https://medium.com/@ffvanderlaan/navigation-in-jetpack-compose-using-viewmodel-state-3b2517c24dde
- https://mahan-yt.medium.com/compose-navigation-with-viewmodel-9f5ba9013975

More Notes on View Model Factory

- The function viewModel(...) will create a new HomeViewModel if it's the first time you request the ViewModel, or it will return the previous instance of HomeViewModel if it already exists. That's one of the advantages of using ViewModels, because on configuration change (or on recomposition) your ViewModel should be reused, not created again. And the way it works is by using a ViewModelProvider.Factory to create the ViewModel when it's necessary. Your ViewModel has a parameter on its constructor, there's no way the default Android classes would know how to create your ViewModel and pass that parameter (i.e. the repository) without you providing a custom ViewModelProvider.Factory. If your ViewModel doesn't have any parameters, the default ViewModelProvider.Factory uses reflection to create your class by using the no-argument constructor.
- By using viewModels and ViewModelProvider.Factory, the framework will take care of the lifecycle of the ViewModel. It will survive configuration changes and even if the Activity is recreated, you'll always get the right instance of the WordViewModel class.
- Factory is used because ViewModel creation is conditional - it will only happen if ViewModelProvider does not contain given ViewModel.
- Implementations of ViewModelProviders.Factory interface are responsible to instantiate ViewModels. That means you write your own implementation for creating an instance of ViewModel.
- When to use ViewModelProvider.Factory?
    - If your ViewModel have dependencies then you should pass this dependencies through the constructor (It is the best way to pass your dependencies), so you can mock that dependencies and test your ViewModel.
- When not to use ViewModelProvider.Factory
    - If your ViewModel have no dependencies then you will not require to create your own ViewModelProvider.Factory. The default implementation is enough to create ViewModel for you.
- https://stackoverflow.com/questions/67985585/why-do-we-need-viewmodelprovider-factory-to-pass-view-model-to-a-screen
- https://medium.com/koderlabs/viewmodel-with-viewmodelprovider-factory-the-creator-of-viewmodel-8fabfec1aa4f
- https://developer.android.com/topic/libraries/architecture/viewmodel/viewmodel-factories

ViewModels with Dependencies

- https://developer.android.com/topic/libraries/architecture/viewmodel/viewmodel-factories
- Following dependency injection's best practices, ViewModels can take dependencies as parameters in their constructor.
  - These are mostly of types from the domain or data layers.
- Because the framework provides the ViewModels, a special mechanism is required to create instances of them.
  - That mechanism is the ViewModelProvider.Factory interface.
  - Only implementations of this interface can instantiate ViewModels in the right scope.
- (Note: When injecting ViewModels using Hilt as a dependency injection solution, you don't have to define a ViewModel factory manually)

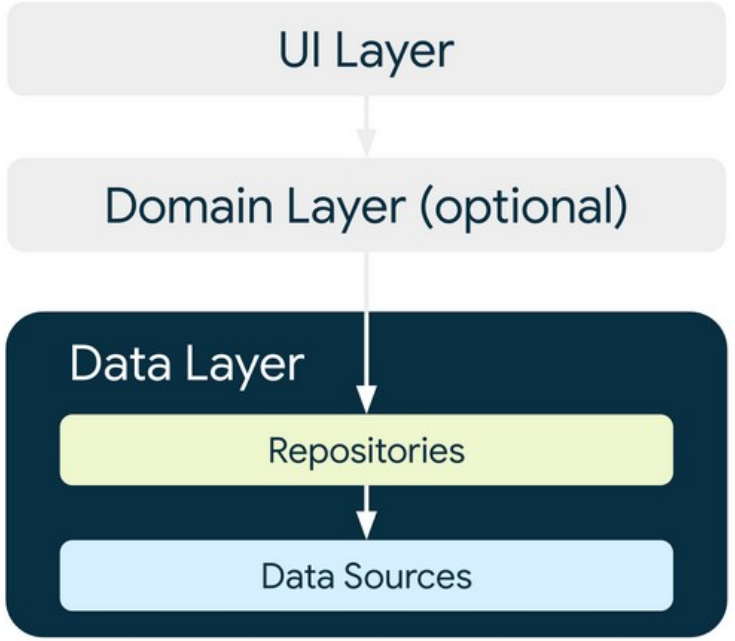
Use Factory

- Finally, once we have all the above, we can call the factory in the constructor of our Screen as follows:

```
/* This constructor will new up our view model using injection as appropriate (in Factory) */
@Composable
fun MySimpleSavedScreen(myViewModel: MyViewModelSimpleSaved =
                        viewModel(factory=MyViewModelSimpleSavedFactory())) {
```

- This should now create an appropriate ViewModel backed by a DataStore, and only one of these objects will exist for the lifetime of the application.
  - This is important for data integrity.

Reading / Codelab



- Read these links
  - https://developer.android.com/topic/architecture/ui-layer
  - https://developer.android.com/topic/architecture/domain-layer
  - https://developer.android.com/topic/architecture/data-layer
  - https://medium.com/@bhavnathacker14/viewmodels-in-clean-architecture-dos-and-donts-part-1-3f8d7fe43aa2

- This optional codelab will walk you through using a preferences DataStore
  - https://developer.android.com/codelabs/android-preferences-datastore#5

Notes: Share a Viewmodel

- viewModel() returns an existing ViewModel or creates a new one in the given scope.
  - The ViewModel is retained as long as the scope is alive.
  - For example, if the composable is used in an activity, viewModel() returns the same instance until the activity is finished or the process is killed.
- https://www.appsloveworld.com/kotlin/100/85/how-can-i-share-view model-from-one-screen-to-another-screen-in-jetpack-compose

- https://issuetracker.google.com/issues/188693123?pli=1

Notes: ViewModel & Navigation

- https://stackoverflow.com/questions/69002018/why-a-new-viewmodel-is-created-in-each-compose-navigation-route

- Usually view model is shared for the whole composables scope, and init shouldn't be called more than once.
- But if you're using compose navigation, it creates a new model store owner for each destination. If you need to share models between destination, you can do it like in two ways:
- By passing it directly to viewModel call. In this case only the passed view model will be bind to parent store owner, and all other view models created inside will be bind(and so destroyed when route is removed from the stack) to current route.
- By proving value for LocalViewModelStoreOwner, so all composables inside will be bind to the parent view model store owner, and so are not gonna be freed when route is removed from the stack.

Notes: Hilt Dependency Injection

- Hilt is the recommended solution for dependency injection in Android apps, and works seamlessly with Compose.
  - https://developer.android.com/jetpack/compose/libraries#hilt
  - https://levelup.gitconnected.com/dependency-injection-with-hilt-in-android-73921b76c661
  - https://medium.com/@Shvet5/datastore-in-compose-c28504958552
  - https://medium.com/@ramg7/android-user-preferences-simplified-preferences-datastore-with-hilt-c08da9691667
- The viewModel() function mentioned in the ViewModel section automatically uses the ViewModel that Hilt constructs with the @HiltViewModel annotation.
- It actually turns out to be complex to figure out, so let's skip it...

```
@HiltViewModel
class MyViewModel @Inject constructor(
    private val savedStateHandle: SavedStateHandle,
    private val repository: ExampleRepository
) : ViewModel() { /* ... */ }

// import androidx.lifecycle.viewmodel.compose.viewModel
@Composable
fun MyScreen(
    viewModel: MyViewModel = viewModel()
) { /* ... */ }
```

- Dependencies for hilt and navigation

```
dependencies {
    implementation("androidx.hilt:hilt-navigation-compose:1.0.0")
}
```