Persistent Storage

Files and Room

Persistence

- Viewmodels are lifecycle aware, so data stored in a viewmodel persists as long as the application is running, across screen rotations/app backgrounding, etc
- But what should we do about data that needs to be stored even when the app is closed?
- We'll look at a few options for simple data (files, small string/simple object data) as well as for more complex data

Small strings/objects: DataStore

- Provides a small amount of “key value storage” in persistent storage, so basically an on-SSD-Bundle
- A new API addressing some deficiencies of it's predecessor (`SharedPreferences`)
- Those deficiencies are related to concurrency issues, which show up at the worst times without lots of extra care
- As a result, the API is not super user friendly, but is safe and reliable
- It's based on `Flows` so we'll use coroutines (suspend functions) to interact with it
- There's a type safe version available, but it's quite a bit of work to set up, so I won't cover it

DataStore setup code

```
//extension property, in this package, Context objects will now have a datastore property
//that  calls  preferencesDataStore.getValue/setValue
val Context.dataStore: DataStore< Preferences>
        by preferencesDataStore(name = "preferenceFilename")
```

```
val prefs: Flow< Preferences> = dataStore.data

//Key with int value
val keyForIntValue: Preferences.Key< Int> = intPreferencesKey("some_key")

//Another key with string value
val keyForStringValue = stringPreferencesKey("another_key")
```

DataStore read/write operations

```
//a suspend func, so run in a coroutine scope
datastore.edit { prefs: MutablePreferences ->
        //can read from prefs too
    prefs[myKey] = myValue
}

//also in a coroutine scope:
prefs.collect(){ //flow operator
    //it = preferences
    //use default if there is no entry
    someData = it[someKey] ?: defaultValue
}
```

When to use DataStore

- Simple, small key-value data
- Requires no permissions, but is slow-ish... the whole hash table is read/written to disk on updates/reads

Files

- We can use all the Java APIs for dealing with files
- There's a few different directories we have access to with different properties/requirements
- But once we've opened a file in the right directory, there's nothing Android specific

App-specific files

- Folders for storing files that no other apps can access
- Can be in internal (smaller, built in) or external (larger, probably an SD card, though these are rare now)
- `getFilesDir()` and `getExternalFilesDir()` return the `File` objects corresponding to those folders
- Replace `Files` with `Cache` to get folders which will be automatically cleaned up if storage space gets used up. Use it for caching or maybe something like logs
- Example: `File(context.filesDir, "myFilename")`

Shared “Media”

- Used for storing stuff like images that should be shared among other apps (any apps that want to open/share images, audio, etc)
- Access to this is via the Media Store API or Photo Picker API
- The MediaStore API requires you to specify some app permissions in your `AndroidManifext.xml` file, which the user sees/approves when they install your app

Example

Basic DataStore/File app

Relational Data

- How do we store relational data?
- Android (and ios) include the SQLite relational database library in all(?) apps
- It supports lots of the stuff we saw this summer in mysql/mariadb, but stores its data in local files
- Interacting with it via prepared statements is straightforward, but error prone and annoying
- We'll talk about Android's analogue to LINQ (sort of) called `Room` which is a much nicer way to interact with data stored in a local DB

Quick Aside: Tweaks to MVVM

- Recall, in MVVM the viewmodel is reponsible for being the mediator between the views and the Model (our actual data)
- For all previous examples, the “model” was simple enough to live in our viewmodel class, but if we add a DB, it won't make sense to store it there
- We'll actually split the “model” into 2 pieces to help keep our code more organized

Repository pattern

- The “Repository” is basically the “in-memory” part of the model
- It acts as sort of a “viewmodel for the viewmodel” in that it presents a simplified version of the data for the VM, while potentially doing some hard work under the hood
- The “simplest” version of a repository is basically moving the LiveData we had stored in the VM into a separate class
- A more complex implementaiton might be reponsible for getting data from a database or via HTTP requests and presenting it to the VM (likely through LiveData)

Room

- Room is an abstraction layer over SQLite, similar to LINQ
- Lots of scaffolding automatically converts between table rows and kotlin objects
- Mostly powered via Java `@Annotations` to help Room figure out that scaffolding
- Room actually runs a compiler that writes classes for you based on abstract classes and interfaces that you write/annotate

Room components

- A Database. This is a class annotated with `@Database`
- Entities: correspond to tables, annotated with `@Entity`
- Data Access Objects (DAOs): Interfaces annotated with SQL code. Room will automatically create classes implementing this interface. This is essentially LINQ's scaffolding annotated with `@Dao`
- DAO queries are checked at compile time, so we get errors sooner

Database class

- An `abstract` class that inherits from `RoomDatabase` and is annotated as `@Database`
- Needs to specify its entities in the `@Database` annotation
- Needs a getter for its `@Dao` interface
- Since usually we only have one DB, often it's a singleton (more on that later)

Entity classes

- These are your data classes
- If you're careful, all you need to do is stick an `@Entity` annotation on the class and a `@field:PrimaryKey` annotation on one field
- Entities are our “Model” classes (the real data we're tracking) with some annotations to specify how they should get turned into DB tables

DAO classes

- These are the links between your Kotlin objects and SQL rows
- These are Interfaces annotated with `@Dao` which Room will create implementing classes
- Methods are annotated to specify what kind of SQL matches the kotlin method signature
- For complex queries, you actually write the SQL code in your query annotation: `@Query("SELECT * FROM my_table ORDER BY birthday DESC")`

DAO Examples

```
@Insert(onConflict = OnConflictStrategy.REPLACE) fun addUser(user:
User): Void
@Delete fun deleteUsers(vararg users: User)
```

- The first one adds a user, overwriting an existing row if the primary key already exists
- The second deletes rows. vararg means you can pass any number of Users to the method
- Again, you just write the signatures in your interface and Room writes classes for you

Details: Singletons

- Our app should only have a single Database/Repository object
- It's common to want exactly one instance of a class, and the software engineering term for this is the “Singleton Pattern”
- There's some trickyness about efficiently getting exactly one initialized instance in multithreaded programs. We need to make sure that if 2 threads try to access the instance for the first time together, that only one of them actually creates it
- In kotlin an `object` handles all this synchronization for us automatically. Unless...

Pain point: Singletons with constructor params

- If we need to pass constructor params, we have to implement the initialization stuff manually
- It's still nicer in Kotlin than in Java
- We have a static reference to the one object, and we make a getter method for it
- In that method we safely create it if necessary, and otherwise return the existing instance

Android specific stuff

- You can write a class that inherits from `Application` which will be a singleton with a lifetime outliving all Activities.
- You can handle a fair bit of Singleton management stuff with that class and the `lazy` delegate
- Unfortunately, `Room` and `ViewModels` require some extra set up, so we'll have to write some really ugly code for them...

Room + Coroutines

- Room is built with coroutines in mind
- DAO methods can (should) be marked as suspend so that the coroutine can yield before doing a slow DB query, and let another task do some work
- Methods that return flows should not be marked as suspend since the suspending stuff will happen in the code that watches the Flow

Example: Simulated “weather” app

- Simulate a slow network request with the `delay` suspend function
- View asks the VM to get weather data for a city
  - VM asks the Repo to do it
    - Repo does a “network request”
    - Repo eventually calls a DAO method to insert the just-retreived data into the DB
  - Room notices that the insert may impact some of the DAO's `Flow` method, new values are emitted to the flows
  - The Repository has wrapped the flow as a liveData so the VM and eventually view get the data
- The view's observer finally updates the display to show the new data

Note: Sneak peak at “Jetpack Compose”

- My demo doesn't use a Views based UI, it uses the modern “Jetpack Compose” system
- The ONLY thing that's different from what we've seen is how my activity looks. All the VM + Repo + DB + DAO + Etc would be exactly the same in a views based app
- We don't call “observe” here, but it works basically the same way, updating the UI when the Flow/Live data changes
- I replaced tons of recycler view code with about 5 lines