Advanced Mobile Application Development

Recipe App

We will be building an app to search and save recipes using the Spoonacular API. Eventually, you'll need to sign up for a free account so that you can get a key, but for now we'll just be working with local data so you can hold off on that if you want. The documentation for the API can be found here.

Throughout the process of creating this app, we will learn about a variety of advanced Android concepts including tables/grids, different types of navigation, working with JSON, async tasks, API requests, data persistence, and more. We will continue to build out this app over the next several weeks and I'll be creating branches for each checkpoint on my GitHub repo so you won't be sure to check those.

Project Setup

Our main layout will be based on a bottom navigation bar so we'll use that template as a starting point and build upon it. I've already created the skeleton for the project and it is available on my GitHub repo in the branch "01-recyclerview".

Checkout this branch and then open up Android Studio.
On the splash screen, **select** *Import project (Gradle, Eclipse ADT, etc.)* navigate to the recipe-app folder in my repo and select open.
Let Android Studio do it's initial project sync with Gradle and then **build and run** it to make sure everything is working properly.

Project Layout

Fragments

Navigate to the following folder: *recipe-app/app/src/main/java/com/isaac/recipes/ui/search*

Fragments are a concept in Android that represent a portion of a UI and they can be combined within an activity to create a multi-pane layout. They can also be reused in different activities throughout your application. Essentially, you can think of it as a sub-activity. If you're familiar with React.js you can think of fragments as React components that can be rendered and reused as much as you like.

A fragment has its own lifecycle that is seperate from the parent activity's lifecycle. However, the parent activity's lifecycle events affect all of the fragments within it. Pausing or destroying the parent activity pauses or destroys all the fragments within it.

You can either use a fragment in the activity's xml layout or load it programmatically.

ViewModels

The ViewModel class is designed to store and manage UI-related data in a lifecycle conscious way. The ViewModel class allows data to survive configuration changes such as screen rotations.

This means that any data we want to represent in our activities or fragments should live in our ViewModel including things like collections, UI selections, or any other data that represents the state of our view.

Raw JSON File

We'll start building our application using local data before we move on to making API requests. Our static sample data can be found in *recipe-app/app/src/main/assets.*

Layout

Our main activity's layout includes a fragment that will show the currently selected tab and a bottom navigation element to navigate between our two views. All of this was included in the Bottom Navigation template project provided by Android Studio.

Each fragment's layout contains an empty textview. The text is populated at runtime by the Fragment. The actual text is requested from the ViewModel by the fragment.

Build and run the app to see the tab bar functionality.

Reading and Parsing the JSON

Model Data Class

We'll use the Kotlin Data Class to model out JSON data. As mentioned in the Intro to Kotlin videos, data classes are useful for data transfer objects because they have a better toString() method and auto getters and setters, among other advantages.

To start, create a new package under your base package. If you're working from my repo the base package is com.isaac.recipes

Ctrl-Click the base package and **select** New->Package
Name it *com.isaac.recipes.data*
Create a new class in the package by **ctrl-clicking** the *data* package and **selecting** New->Kotlin File/Class

Name it *Recipe* and set the type to *Class* and press enter

Add keyword *data* to the beginning of the class declaration
We don't need a class body at the beginning so **remove** the curly braces
The properties of the class will match the names and types of the JSON object keys/values and we'll **place** them in the primary constructor like so

```
data class Recipe (
    val id: Int,
    val title: String,
    val readyInMinutes: Int,
    val servings: Int,
    val image: String,
    val imageUrls: Set<String>
)
```

Read File from Assets

We'll create a helper class to load in our JSON from the file in the assets folder.
To do this, let's **create** another sub package of our main package called *utils*

Ctrl-click *com.isaac.recipe* and select New->Package
Name it *com.isaac.recipe.utils*
Ctrl-click on the new utils package and **select** New->Kotlin File/Class
Name it *FileHelper* and **select** *Class* for the type

We'll create a function inside a companion object to get and return the raw text from a file. In Kotlin, a function inside of a companion object of a class is much like a static method of a class in Java — it belongs to the class and not it's instances. This function needs to know the get our application context which is an interface to global information about our app that is created by the operating system.

```
class FileHelper {
    //functions like a static method
    companion object {
        fun readTextFromAssets(context: Context, filename: String): String
{
            //the .use() methods to make sure that all of my streams are
closed appropriately
            return context.assets.open(filename).use { inputStream ->
                inputStream.bufferedReader().use {
                    it.readText()
                }
            }
```

```
        }
    }
}
```

Next, we need to utilize this method in our SearchViewModel. We'll need to get the application context so we can pass it to the readTextFromAssets() function which isn't exposed to a ViewModel by default. The simplest way to fix this is to **change** the superclass of SearchViewModel to AndroidViewModel() and **change** it to receive a reference to the application which will be passed to the parent class. We'll also add an init block to load the text.

```
class SearchViewModel(app: Application) : AndroidViewModel(app) {

    //load the data as pure text when the ViewModel is instantiated
    init {
        val dataText = FileHelper.readTextFromAssets(app,
"recipe-data.json")
        Log.i("recipeLogging", dataText)
    }

}
```

If you run the app at this point, you should be able to pull up the log and filter for "recipeLogging" and see the text from our JSON file.

Parse JSON

To parse our raw JSON string into an out data model, we'll use a library called Moshi. This is a newer library that a lot of developers are using instead of the older Google alternative, GSON. Moshi was developed by Square, the payment processing platform.

First, we need to add a dependency on Moshi. **Add** the following line to the build.gradle file for the app module and then **sync** gradle using the notification at the top of the editor.

```
implementation("com.squareup.moshi:moshi-kotlin:1.9.2")
```

A well architected android application will make use of a concept called repository classes. There's really nothing unique about these classes — they are merely intended to hide the data source and parsing logic from the rest of the application. We'll go ahead and move the data logic to one of these classes.

Additionally, modern android applications use a publisher/subscriber pattern to notify various components of the app about changes to the data. In Kotlin, this is usually accomplished using a LiveData object.

To do so, first **create** a new class in our *data* package by **ctrl-click** the package name select New->Kotlin File/Class and **name** it *RecipeRepository* with type *Class*

Make the following changes/additions to the repository class:

1. **Take** in a property in the primary constructor to represent the application context
2. To use Moshi, we need to **create** a parameterized type so that the JSON adapter knows what we will be using to parse the data. In our case this will be a list of recipes. **Set** this as a private class property as we will use it multiple times.
3. **Instantiate** a property of type MutableLiveData<List<Recipe>>() to publish our results to any and all subscribers
4. **Create** a function to get and parse the recipe data and then update the LiveData object
5. **Call** that function from an init block

When all is said and done the repository class should look like this:

```
class RecipeRepository(val app: Application) {
    //parameterized type property for Moshi
    private val listType = Types.newParameterizedType(List::class.java,
Recipe::class.java)

    //LiveData object consisting of our recipe data
    //we will publish from this class and subscribe from our fragment
    val recipeData = MutableLiveData<List<Recipe>>()

    //fetch the data when the class is instantiated
    init {
        getRecipeData()
    }

    //get the raw text from our json file and update the LiveData object
with the parsed data
    private fun getRecipeData() {
        val dataText = FileHelper.readTextFromAssets(app,
"recipe-data.json")

        val moshi = Moshi.Builder().add(KotlinJsonAdapterFactory()).build()
        val adapter: JsonAdapter<List<Recipe>> = moshi.adapter(listType)

        //update our LiveData object with the results of our parsing
        recipeData.value = adapter.fromJson(dataText) ?: emptyList()
    }
}
```

Now, we need to refactor our SearchViewModel to incorporate the repository class. **Remove** the init block, **instantiate** our repository class, and **create** a property that will pass along the LiveData object and expose it to the fragment.

This is what the entire SearchViewModel should look like after this:

```
class SearchViewModel(app: Application) : AndroidViewModel(app) {
    //instantiate repository class
    private val recipeRepo = RecipeRepository(app)

    //get reference to LiveData object with a value of type List<Recipe>
    val recipeData = recipeRepo.recipeData

    private val _text = MutableLiveData<String>().apply {
        value = "This is the Search Fragment"
    }
    val text: LiveData<String> = _text
}
```

Finally, we can *observe* the changes to the LiveData object in our SearchFragment class via the SearchViewModel. Add the code in bolds to the onCreateView() function:

```
override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
): View? {
    //instance of ViewModel
    searchViewModel =
ViewModelProvider(this).get(SearchViewModel::class.java)

    //subscribe to data changes in the repository class via the ViewModel
    searchViewModel.recipeData.observe(viewLifecycleOwner, Observer {
        for(recipe in it) {
            Log.i("recipeLogging", "${recipe.title} serves
${recipe.servings}")
        }

    })

    val root = inflater.inflate(R.layout.fragment_search, container, false)
    val textView: TextView = root.findViewById(R.id.text_search)
    searchViewModel.text.observe(viewLifecycleOwner, Observer {
```

```
        textView.text = it
    })
    return root
}
```

The *viewLifecycleOwner* argument tells the LiveData object to remove our class from the list of observers when it is killed. This helps avoid memory leaks within our application.

Now when we **run** our app, we should be able to see our data in the Log (make sure your filter for “recipeLogging”

RecyclerView

Android List/Grid Layout Overview

Android has a few different ways to display information in vertical lists or grids

- A **ListView** is used to for a scrollable, vertical collection of data
    - Simple to use but not really efficient (creates and destroys views for each cell as you scroll)
- A **GridView** is used for horizontal or vertical collection of data organized in columns and rows
    - Like ListView, this has fallen out of style since it’s not performant
- A **RecyclerView**
    - The modern solution for displaying collections of data
    - Highly performant since the views for each cell are recycled as the user scrolls
    - Supports both lists and grids
    - Animations of modifying the items in a collection
    - Somewhat complex to implement

To summarize, *GridView* and *ListView* should be used carefully, especially when the data in your collection is not static. I recommend only using these when you know all of the data you’ll be displaying at design time — think of them like static Table Views in Swift/iOS. A *RecyclerView* is the best solution for dynamic data.

RecyclerView Overview

RecyclerView is the container used to display a scrolling list of elements based on large data sets or data that may change frequently.

ViewHolder

The views in the list are represented by view holder objects which are instances of a subclass of RecyclerView.ViewHolder. Each view holder is in charge of displaying a single item with a view.

The RecyclerView creates only as many view holders as are needed to display the on-screen portion of the dynamic content, plus a few extra which are preloaded to anticipate scrolling. As the user scrolls, the RecyclerView takes views which are moving off-screen and rebinds them to the data which is coming on-screen. The view holder objects are managed by an adapter.

Adapter

Adapters are a subclass of RecyclerView.Adapter and they bind application specific data to views. The data source can vary widely (lists, arrays, sets, database query, etc.). The Adapter takes the data and converts each entry into a view that can be added to a layout. There are a variety of ready to use adapters that provide some basic functionality. For more flexibility, there is the option to create a custom subclass.

LayoutManager

A LayoutManager is responsible for determining the size and position of items within a RecyclerView as well as when to reuse views that are no longer visible to the user. To recycle a view, the layout manager may ask the Adapter to replace the contents of a view with a different element from the dataset. There are some stock LayoutManagers available that work for most layouts, and there is the option to create a custom layout manager.

- LinearLayoutManager supports horizontal or vertical scroll lists. Most common LayoutManager used with RecyclerView
- StaggeredGridLayout creates staggered lists similar to the Pinterest layout
- GridLayoutManager displays a content in a traditional grid

ItemAnimator

When changes are made to the data (add, move, delete), the RecyclerView uses an extension of the abstract class RecyclerView.ItemAnimator to change the item's appearance. There is a default implementation called RecyclerView.DefaultItemAnimator that is used to provide stock animations. For custom animations, define your own animator object by extending RecyclerView.ItemAnimator.

Add RecyclerView

Let's add a recycler view to our search fragment. To do so, **go** to the *frament_search.xml* file in the *res/layout* folder.

Delete the TextView that is currently inside the view and **drag** out a RecyclerView on to the screen.
In the Attributes pane under the Declared Attributes section, **change** the values for *layout_width* and *layout_height* to *0dp (match parent)*
Set constraints in the layout section to *0* for top and bottom and *8* for each side

Switch to XML view by **pressing** the Code icon at the top left of the editor (hamburger menu looking thing)
Add a property to the XML for the RecyclerView to set the LayoutManager
Add an ID *@+id/recyclerView*
The XML for the RecyclerView should now look something like this:

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recyclerView"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_marginStart="8dp"
    android:layout_marginEnd="8dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager" />
```

Create Item Layout

To create the layout for each item in our RecyclerView, we need a new layout file. **Create** it by **ctrl-clicking** on the layout folder in the project pane and **selecting** New->Layout Resource File. **Name** it *recipe_grid_item* and make sure the root item is **set** to *androidx.constraintlayout.widget.ConstraintLayout.* **Click** OK.

Switch to the Design view of the layout file using the icon at the top right of the editor
Drag out two TextViews and align one to the top left of the view and the other to the top right
For the left one, **set** the id to *titleTextView* and set textAppearance to *Medium*
For the right one, **set** the id to *prepTextView*
Set top, left, and bottom constraints for the *titleTextView* to *8*
Set for *prepTextView*, set the right constraint to *8* and the bottom constraint to the bottom of *titleTextView* so that the baselines are aligned
Set the right constraint of *titleTextView* to the *prepTextView* and **give** it a constant of 16 and change the *layout_width* value to *0dp (match parent)*

Next, **switch** to XML view and **set** the *android:layout_height* property of the top level constraint layout to *wrap_content*

Create Adapter and ViewHolder

Finally, to show data within our RecyclerView, we need to create a custom Adapter that uses a custom ViewHolder to inflate views for each element in the RecyclerView and binds the appropriate data to each view.

We'll start by **creating** a new Kotlin class in our *ui.search* package and **name** it *SearchRecyclerAdapter* and subclass RecyclerView.Adapter. Take in a context and the data list in the primary constructor. We'll implement three required methods for the Adapter to describe how many items are in the list, inflate a view, and bind data to the view.

The ViewHolder will be an inner class of our adapter and its purpose is to get references to each of the TextViews in our item layout.

The end product of this class should look like this:

```
class SearchRecyclerAdapter(val context: Context, val recipeList:
List<Recipe>) : RecyclerView.Adapter<SearchRecyclerAdapter.ViewHolder>() {

    //custom ViewHolder
    inner class ViewHolder(itemView: View) :
RecyclerView.ViewHolder(itemView) {
        val titleText = itemView.findViewById<TextView>(R.id.titleTextView)
        val prepText = itemView.findViewById<TextView>(R.id.prepTextView)
    }

    //inflate the view for the item
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ViewHolder {
        val inflater = LayoutInflater.from(parent.context)
        val view = inflater.inflate(R.layout.recipe_list_item, parent,
false)
        return ViewHolder(view)
    }

    //number of items in RecyclerView
    override fun getItemCount() = recipeList.count()

    //set the data for the view
    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val curRecipe = recipeList[position]

        holder.titleText.text = curRecipe.title
        holder.prepText.text = "${curRecipe.readyInMinutes} min"
    }
}
```

Lastly, we need to implement this in our SearchFragment. Make the following changes:

```
class SearchFragment : Fragment() {

    private lateinit var searchViewModel: SearchViewModel
    private lateinit var recyclerView: RecyclerView

    override fun onCreateView(
            inflater: LayoutInflater,
            container: ViewGroup?,
            savedInstanceState: Bundle?
    ): View? {
        //instance of ViewModel
        searchViewModel =
ViewModelProvider(this).get(SearchViewModel::class.java)

        val root = inflater.inflate(R.layout.fragment_search, container,
false)
        recyclerView = root.findViewById(R.id.recyclerView)

        //subscribe to data changes in the repository class via the
ViewModel
        searchViewModel.recipeData.observe(viewLifecycleOwner, Observer {
            //instantiate adapter
            val adapter = SearchRecyclerAdapter(requireContext(), it)
            //set the adapter to the recyclerview
            recyclerView.adapter = adapter
        })

        return root
    }
}
```

Run the app and verify that the data is showing up!

We did a lot of work to set up the architecture of our application so that in the future we can make changes more easily. Next time, we'll be taking a look using images and creating a detail view to see more info about the recipe such as ingredients and directions. Reach out if you have any questions about what we've done today.