Recipe App: Detail View

Today we will be adding a new Fragment to show the details of a recipe that the user will navigate to by selecting a recipe from our RecyclerView in the search tab.

Overview and new JSON

Take a look in the *assets* folder to see some sample JSON for the details of a recipe. This is exactly what the response will look like when we integrate the Spoonacular API into our project next week, and you can see there is quite a bit of info. For our purposes, we'll be focusing on the image url string, the summary string, the extendedIngredients array, and the instructions string. Working with this more complex JSON structure should give us a more real-world understanding of how to work with JSON in Android.

Handle Click Events on RecyclerView

When the user clicks on an item in the RecyclerView, a click event is dispatched. We'll be taking this event in the Adapter, but the action needs to eventually be handled by the parent which is a Fragment in our case. In order for the Adapter and Fragment to communicate we'll need to use an Interface.

If you're not familiar, Interfaces are basically stateless abstract classes that contain abstract methods or actual method implementations. Since they are stateless, interface properties need to be either abstract or you have to provide implementation for the getter method. Our Interface will be very basic and it will have a single abstract method inside it.

Add the interface to the bottom of the SearchRecyclerAdapter:

```
interface RecipeItemListener {
    fun onRecipeItemClick(recipe: Recipe)
}
```

Next, **navigate** to the SearchFragment and **implement** the listener in the class definition:

```
class SearchFragment : Fragment(), SearchRecyclerAdapter.RecipeItemListener
{
```

You should get an error since we need to implement the abstract onRecipeItemClick() method. **Select** "Implement Members" from the error and it will provide us with the stub for the required method. For now, we'll just **log the recipe** that was clicked. It should look like this:

```
override fun onRecipeItemClick(recipe: Recipe) {
    Log.i("recipeLogging", recipe.toString())
}
```

Next, we need to set the fragment as the listener in the SearchRecyclerAdapter class. To do this, we need to **take in an instance of our RecipeItemListener interface** in the primary constructor.

```
class SearchRecyclerAdapter(val context: Context, val recipeList:
List<Recipe>, val itemListener: RecipeItemListener)
```

Now we need to set this new itemListener property as the actual click listener in the onBindViewHolder method of SearchRecyclerAdapter. There is a property of the holder parameter called itemView that represents the view of the item we're working with. We will call the setOnClickListener method and use a lambda to pass the data item to the Fragment. **Add** the following code block to the bottom of onBindViewHolder

```
//pass the data item to the fragment click listener
holder.itemView.setOnClickListener {
    itemListener.onRecipeItemClick(curRecipe)
}
```

The last step is to **pass in the itemListener argument** when we instantiate the SearchRecyclerAdapter in the SearchFragment in the onCreateView method:

```
val adapter = SearchRecyclerAdapter(requireContext(), it, this)
```

Now we can **run** our app and click on items in the RecyclerView to see the recipe printed to the Log.

Detail Fragment

Create Detail View Fragment and Navigate

When we create apps using a navigation component, we can define each screen as a Fragment instead of an Activity. This makes our app extremely modular and creates less overhead. Since our Recipe App uses a bottom navigation, we can take advantage of this fact.

To get started, **navigate** to the *res/navigation* folder and **open** the mobile_navigation.xml

Click on this icon (New Destination) in the top left of the editor area and **select** 'Create new destination'

Name it RecipeDetailFragment and **make sure** “Create layout XML” is checked and fragment name is fragment_recipe_detail_view
Uncheck “include factor fragment methods”. Then **click** finish.

Then, **click and drag** from the circle on the right side of navigation_search (home) to the new fragment to create an *action* that can be triggered from our code. **Switch** to the XML view and **note** the id for the action.

Next, we need a reference to the application’s nav controller in our SearchFragment so that we can handle navigation events within that class. To do this, **go to** our Search fragment and **add** a property to store that reference:

```
private lateinit var navController: NavController
```

Instantiate this property somewhere near the top of the onCreateView() method:

```
 //instantiate nav controller reference using its id from the xml of the
main activity layout
navController = Navigation.findNavController(requireActivity(),
R.id.nav_host_fragment)
```

Lastly, we **fire the navigation action** in our onRecipeItemClick method:

```
override fun onRecipeItemClick(recipe: Recipe) {
    Log.i("recipeLogging", recipe.toString())
    navController.navigate(R.id.action_navigation_search_to_recipeDetailFragment)
}
```

Run the app and select any item from the RecyclerView to see the navigation in action. You’ll notice that there is a back arrow in the app bar of the detail fragment but it doesn’t work. We’ll implement this next. For now, you can use the device’s back button to return to the SearchFragment.

Add Up Navigation

Before we add logic for the up navigation, let’s fix our app structure. We want to make sure that our new detail fragment is in the correct package. Right now, it’s in the base package, but we should move it to the search package since that’s what it’s coupled with. **Drag and drop** the detail fragment into the search package and **select** *Refactor* on the dialog that pops up.

Since we are loading the detail fragment into the activity using the navController, we can put the logic for navigating up through the fragment stack in the MainActivity class and it will apply to all of the screens in our application. This greatly reduces the amount of code needed to handle upwards navigation throughout the entire app.

Navigate to our MainActivity and **implement** the FragmentManager interface:

```
class MainActivity : AppCompatActivity(),
FragmentManager.OnBackStackChangedListener {
```

You should get an error because we need to implement the required method. Use the Implement Methods action to create the method stub for *onBackStackChanged*. This function is called anytime a fragment is pushed onto or popped off of the fragment back stack. We will override this to enable and disable the up navigation arrow based on if it's possible to up nav — we need to have a stack of at least 2 in order to up nav.

Implement *onBackStackChanged*:

```
override fun onBackStackChanged() {
    shouldDisplayHomeUp()
}

private fun shouldDisplayHomeUp() {
    val canGoBack = supportFragmentManager.backStackEntryCount > 0
    supportActionBar?.setDisplayHomeAsUpEnabled(canGoBack)
}
```

Next, we need to **pull navController reference out** of the *onCreate* method and set it as a lateinit property so that we have access to it throughout the class:

```
private lateinit var navController: NavController
```

Make sure you're still initializing in the *onCreate* method tho!

Lastly, we will override a method of AppCompatActivity to called onSupportNavigateUp to call the navigateUp() method on our navController property:

```
override fun onSupportNavigateUp(): Boolean {
    navController.navigateUp();
    return true;
}
```

Run the app and verify that the up navigation works from the DetailFragment to the main SearchFragment.

You'll probably notice that the transitions could use a little work and they are currently very abrupt. It's not that hard to define some simple transition animations and we'll be taking a look at those later in this unit.

Pass Data to Detail View

Now that we have our navigation linked, we can think about how to pass data between the SearchFragment and the RecipeDetailFragment. Keep in mind that we have yet to actually get the data from our local JSON yet, but we can still think about how we want to construct the passing of data down to the detail fragment.

There are a few different ways to do this, but the easiest for us is going to be through the use of a Shared View Model because we are using a . Since we already have a ViewModel that is used by our SearchFragment, we can think about pulling this ViewModel up a level to be controlled by the activity. Once we do this, it will be easy to make data available to all of the fragments that are managed by our activity.

To start, let's rename our SearchViewModel to reflect the changes we are going to make. Since this ViewModel will be shared by all of the fragments in our search package, we'll call it SharedSearchViewModel.
Ctrl-click on the name SearchViewModel in the class declaration and **select** *Refactor->Rename…* then change its name to SharedSearchViewModel and **press** enter*.* Be sure to rename anything it asks you to in the next dialog that pops up.

Now **navigate** to our *SearchFragment.kt* file and **find** where we initialize the ViewModel in our *onCreateView* method. You can see that we are currently passing *this* to the ViewModelProvider so it is owned by this fragment. Instead, we want it to be owned by the activity. To make this change, we simply need to pass the activity instance to ViewModelProvider which we can get with our handy *requireActivity* function.
Change the Initialization to look like this:

```
searchViewModel =
ViewModelProvider(requireActivity()).get(SearchViewModel::class.java)
```

Now that our ViewModel is provided by the activity, the same instance can be used throughout our application. We just need to get a reference to the instance in our *RecipeDetailFragment* in order to use. We'll do this the same way we do it in *SearchFragment*

Add a class property to story the reference in RecipeDetailFragment:

```
private lateinit var sharedSearchViewModel: SharedSearchViewModel
```

And **add** the initialization in onCreateView:

```
sharedSearchViewModel =
ViewModelProvider(requireActivity()).get(SharedSearchViewModel::class.java)
```

Now we just need to add a new LiveData object that will store the recipe that is selected in the RecyclerView. We will publish to this LiveData object from our search fragment in the on click method and use an observer to display the details in RecipeDetailFragement.

Add the LiveData object to the SharedSearchViewModel:

```
val selectedRecipe = MutableLiveData<Recipe>()
```

Then, in the onRecipeItemClick method of our SearchFragment, lets **update** our live data object with the recipe that the user selected from the RecyclerView. Make sure that this code goes before the navigation code:

```
sharedSearchViewModel.selectedRecipe.value = recipe
```

For now, to make sure this is working we'll simply add an observer in the RecipeDetailFragment that logs the selected recipe. Add the following code to the onCreateView method of RecipeDetailFragement:

```
sharedSearchViewModel.selectedRecipe.observe(viewLifecycleOwner, Observer{
    Log.i("recipeLogging", "Selected recipe: ${it.title}")
})
```

We can now **run** the app and **check** the logs to make sure that everything looks right.

Load JSON

Load and Parse Data from Sample Recipe JSON

Now that we've gotten our navigation working and figured out how we'll pass data between fragments in our navigation hierarchy, let's work on how we'll load the details for the selected recipe. Eventually, this will involve another API call to fetch the details from the server, but for now we can use the local sample JSON file to mock this response.

Model Class

As you may have guessed, the logic and source for getting and parsing this data will go in our *RecipeRepository* class, but first we need to model our data using another data class. **Navigate** to the Recipe.kt file in our data package.

As I mentioned earlier, our JSON has a ton of data in it, but we'll be primarily concerned with a small subset of the data at this point: title, summary, image, readyInMinutes, servings, instructions, and extendedIngredient. Most of this are basic types (String, Int, etc.) but you'll notice that extendedIngredient is an array of objects, so we'll need to model those objects with a data class as well.

To get started, add a new data class called RecipeDetails below the existing Recipe data class. In the construtor, we'll set the properties we're concerned with. This class should look like this:

```
data class RecipeDetails (
    val title: String,
    val summary: String,
    val image: String,
    val readyInMinutes: Int,
    val servings: Int,
    val instructions: String,
    val extendedIngredients: Set<Ingredient>
)
```

You'll get an error on the Set<Ingredient> line since we haven't created the Ingredient class yet. **Do that** now below *RecipeDetails:*

```
data class Ingredient(
    val originalString: String,
    val name: String,
    val amount: Double,
    val unit: String
)
```

We could probably get away with just using the originalString key but I added the others just in case we need them later.

Load and Parse in Repository Class

Now we'll go to our RecipeRepository class in the data package, and add a method that will load the raw JSON string from our local file and then parse it and update a LiveData object. We will observe the LiveData object in our RecipeDetailsFragment via the SharedSearchViewModel.

To start, **add** the method and LiveData object to RecipeRepository:

```
//LiveData for the recipe details
val recipeDetails = MutableLiveData<RecipeDetails>()

//get the raw text from our sample recipe details json
private fun getRecipeDetails(forRecipe: Recipe) {
    val detailsText = FileHelper.readTextFromAssets(app,
"sample-recipe.json")
```

```
    val moshi = Moshi.Builder().add(KotlinJsonAdapterFactory()).build()
    val adapter: JsonAdapter<RecipeDetails> =
moshi.adapter(RecipeDetails::class.java)

    //update our LiveData object with the results of our parsing
    recipeDetails.value = adapter.fromJson(detailsText)
}
```

We now have the logic to load, parse, and publish the details for a recipe, but we have yet to determine when this should happen. To do this, we can observe the selectedRecipe LiveData object that is currently in our SharedSearchViewModel class in our RecipeRepository class. There are a few steps to this that we need to take since our Repository class is NOT lifecycle aware and thus we need to take extra care to ensure that we don't create a memory leak.

We will **create** an Observer callback function in our RecipeRepository and then **add** it to the LiveData object in our SharedSearchViewModel class:

```
val recipeSelectedObserver =  Observer<Recipe> {
    getRecipeDetails(it)
}
```

Then, in SharedSearchViewModel, **add the observer** to the LiveData object in an init block:

```
init {
    selectedRecipe.observeForever(recipeRepo.recipeSelectedObserver)
}
```

Since we used the observeForever function, it is very important that we manually remove the observer when we will no longer need it. To do this, we can override the onCleared() method of the ViewModel class which is called when the ViewModel is no longer in use:

```
override fun onCleared() {
    //remove observers added with observe forever to prevent memory leak
    selectedRecipe.removeObserver(recipeRepo.recipeSelectedObserver)
    super.onCleared()
}
```

The last thing to do in the SharedSearchViewModel is **add a reference to the recipeDetails** LiveData object from our RecipeRepository so that we can access it from the RecipeDetailFragment:

```
val recipeDetails = recipeRepo.recipeDetails
```

Finally, we can **add an Observer** in the onCreateView method of the RecipeDetailFragment that logs the RecipeDetail for the selected recipe when it gets updated:

```
sharedSearchViewModel.recipeDetails.observe(viewLifecycleOwner, Observer {
    Log.i("recipeLogging", "Selected recipe instructions:
${it.instructions}")
})
```

We can now **run** the application and verify that everything is working together

This part was somewhat complicated in my opinion and involved a lot of app architecture. I encourage you to spend some time understanding exactly what is happening and how all the interactions between the Fragment, the ViewModel, and the Repository class are working together to deliver the data.

Define Layout for RecipeDetailFragment

Replace the xml for *fragment_recipe_detail* with the content of this gist and take a look at the changes. The layout isn't that complicated but I thought I would just streamline the process by providing the finished product with all the constraints and the proper IDs.

Display Details

RecyclerView Adapter

You'll notice that there is a RecyclerView to display the list of ingredients, so we'll need to create a layout for the items and an adapter to populate the data.

First, create a new Layout Resource File for the items in the RecyclerView. Do this by **ctrl-click** on the layout folder and **selecting** New->Layout Resource File. Our layout will be very simple:

- Add a text view and pin to top and left
    - Change id to ingredientTextView
- Change layout_height of the Constraint layout to wrap_content

Now we can create the adapter. It will look very similar to the SearchRecyclerAdapter. Create a new Kotlin class named IngredientsRecyclerAdapter in the search package and **add the following code**:

```
class IngredientsRecyclerAdapter(val context: Context, val ingredientList:
List<String>) :
RecyclerView.Adapter<IngredientsRecyclerAdapter.ViewHolder>() {

    inner class ViewHolder(itemView: View):
RecyclerView.ViewHolder(itemView) {
        val ingredientTextView: TextView =
itemView.findViewById<TextView>(R.id.ingredientTextView)
    }

    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ): IngredientsRecyclerAdapter.ViewHolder {
        val inflater = LayoutInflater.from(parent.context)
        val view = inflater.inflate(R.layout.ingredient_list_item, parent,
false)
        return ViewHolder(view)
    }

    override fun getItemCount(): Int = ingredientList.count()

    override fun onBindViewHolder(holder:
IngredientsRecyclerAdapter.ViewHolder, position: Int) {
        val curIngredient = ingredientList[position]

        holder.ingredientTextView.text = curIngredient
    }
}
```

Now we can **modify the onCreateView method** of our RecipeDetailFragment to use the new adapter and update the other TextViews

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    //hide the bottom nav since we've moved down in the view hierarchy
    activity?.findViewById<BottomNavigationView>(R.id.nav_view)?.visibility =
android.view.View.GONE

    val root = inflater.inflate(R.layout.fragment_recipe_detail, container, false)

    //references to the necessary views
    val ingredientListView =
root.findViewById<RecyclerView>(R.id.ingredientsListView)
    val recipeTitleTextView = root.findViewById<TextView>(R.id.recipeTitleTextView)
    val instructionsTextView =
root.findViewById<TextView>(R.id.instructionsTextView)
```

```
    sharedSearchViewModel =
ViewModelProvider(requireActivity()).get(SharedSearchViewModel::class.java)

    //we'll populate the details in this Observer as soon as we get them
    sharedSearchViewModel.recipeDetails.observe(viewLifecycleOwner, Observer {
        Log.i("recipeLogging", "Selected recipe instructions:
${it.extendedIngredients}")

        //get ingredient names
        val ingredientList = mutableListOf<String>()
        for(ingredient in it.extendedIngredients) {
            ingredientList.add(ingredient.originalString)
        }
        //add instantiate and use adapter for recyclerview
        val adapter = IngredientsRecyclerAdapter(requireContext(), ingredientList)
        ingredientListView.adapter = adapter

        //set instructions textview
        instructionsTextView.text = it.instructions
    })

    //we'll use this Observer to set the titles ASAP
    sharedSearchViewModel.selectedRecipe.observe(viewLifecycleOwner, Observer{
        //set the title in the action bar
        (activity as AppCompatActivity?)?.supportActionBar?.title = it.title
        //set the title textview
        recipeTitleTextView.text = it.title
    })

    return root
}
```

Now when we **run** our app we should see the detail fragment populated with our sample data.

I was thinking we would load in the image from the server today but I realized you'll need an API key to do that. I'll walk through the API setup next week and we'll get the image displayed in addition to the data fetching.