# Recipe App: API Integration

Today we will be working with the Spoonacular API to dynamically search for recipes based on user input. We will use a few different external tools: Coroutines which is a JetBrains supported library to spawn and manage background threads for async tasks, Retrofit which is an HTTP client created by Square that works well with Moshi, and Glide which is framework for loading images. By the end of this demo we will be loading all of our data from the Spoonacular server.

## Spoonacular Account

You can create a free account to use the Spoonacular API at this link. After you sign up, you can get your API key from the console. You'll need this key to make requests to the api so keep note of it. The full documentation for the API can be found here. We'll be working with the `recipes/search` and the `recipes/{id}/information` endpoints today so feel free to browse the docs to see all the different ways we can use these.

## Dependencies

Open the project in Android Studio and make sure you are on the *03-api-START* branch. Check out the app modules *build.gradle* file as I've added a few dependencies for the external tools we'll be working with to get a head start. You should see lines for added dependencies of Glide, Retrofit, and Coroutines in addition to the one we already added for Moshi. I just got these from the documentation for each tool which is linked to from the lecture assignment on Canvas.

## Global Constants

Also note the new *Global.kt* file I created to hold some constants we want to use throughout our app. These constants can be used by anything in our base package.

## Displaying Images

It is super simple to load images from the server using Glide so we'll start with that. We'll be loading these in our *RecipeDetailFragment* so open that file in the editor. The first time we get access to the url for the recipe's image is when our *selectedRecipe* LiveData object is updated, so we can use that Observer to start loading the image. Behind the scenes, Glide is making an HTTP request to fetch the image in the background using a Coroutine. Once the response is received, Glide will put it in our ImageView.

Before we jump into the code, we need to set up our permissions so that the app can use the internet.

**Open** the *AndroidManifest.xml* which can be found in the *manifests* folder at the top level of our project.
**Add** the following permissions at the top level of the manifest (just anywhere inside the *<manifest>* tag):

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Spoonacular has a guide for loading images which can be found here (open this to talk about the process). We need to construct the url using the id from the selected recipe. As mentioned before, we will do this in the Observer for our *selectedRecipe* LiveData object from our *SharedViewModel*.

**Add** the following code inside the Observer lambda which can be found in the *onCreateView* method of our *RecipeDetailFragment*:

```
//images can be fetched using the following pattern:
https://spoonacular.com/recipeImages/{ID}-{SIZE}.{TYPE}
Glide.with(this)
     .load("${IMAGE_BASE_URL}/${it.id}-556x370.jpg")
     .into(imageView)
```

We'll also need to make some changes to our ImageView to get it to display the image properly so **open** up *fragment_recipe_detail.xm*l and do the following:

- **Change** *layout_height* to *wrap_content*
- **Change** *scaleType* to *centerInside*

Now we can **run the app**, select a recipe from the list, and see the image loaded (may be a short delay).

If your image isn't loading and you see an error in the log that's something like:

```
java.net.SocketException: socket failed: EPERM (Operation not permitted)
```

Then you may need to uninstall the app from your emulator or device and then re-run it from Android Studio. Seems like this is due to adding permissions after the app has already been installed.

## Integrate API for Recipe Searching

### Adjust Data Model

Take a look at what an actual response from the server will look like if we use the recipes/search endpoint. The easiest way to do this is to type the following pattern into the browser: *https://api.spoonacular.com/recipes/search?apiKey=**{insert key here}**&query=**{insert search term}***

You should see something like this:

```
{
  "results": [
    {
      "id": 723984,
      "title": "Cabbage Salad with Peanuts",
      "readyInMinutes": 15,
      "servings": 2,
      "image": "cabbage-salad-with-peanuts-723984.jpg",
      "imageUrls": [
        "cabbage-salad-with-peanuts-723984.jpg"
      ]
    },
    {
      "id": 667917,
      "title": "Cilantro Salsa",
      "readyInMinutes": 20,
      "servings": 2,
      "image": "Cilantro-Salsa-667917.jpg",
      "imageUrls": [
        "Cilantro-Salsa-667917.jpg"
      ]
    }
  ],
  "baseUri": "https://spoonacular.com/recipeImages/",
  "offset": 0,
  "number": 10,
  "totalResults": 370987,
  "processingTimeMs": 198,
  "expires": 1586129321877,
  "isStale": false
}
```

As you can see the structure is as follows: JSON object with some metadata for the response at the top level which the actual results stored in an array of objects with key “results”. Our RecipeData class only addresses the results array so we’ll need to make another simple data class to accommodate the new level above that.

**Open** up the *Recipe.kt* file in our data package and **add the following data class** to capture what we want from the response JSON:

```
@JsonClass(generateAdapter = true)
data class SearchResponse (
    val results: Set<Recipe>,
    val number: Int,
    val totalResults: Int
)
```

You can see that, in addition to getting the results array, I included some of the info about the number of results available and the number of results returned just in case that proves useful.

The line above the data class that starts with JsonClass is called an annotation processor and it is used by Moshi to generate an adapter at compile time. The reason we need this now is because we’ll be using the Moshi factory converter to parse the response from the API and it requires that we have adapters for each of our custom types.

**Add the JsonClass annotation processor** above each of our data classes so we don’t forget later!

## Define Retrofit Interface

A Retrofit interface represents a call to a web service and is necessary to work with the Retrofit HTTP client. To use it, we define a custom Interface that uses Annotation Processors and abstract methods to make different types of requests to our REST API.

To get started, **create a new interface** in our *data* package (ctrl-click, new->kotlin file/class, select interface for the type) and name it Spoonacular Service. I used the Retrofit documentation to determine what my interface should look like. For now, we’ll just include one abstract method with an annotation so **add this to the interface:**

```
interface SpoonacularService {
    @GET("recipes/search?apiKey=${API_KEY}")
    fun searchRecipes(@Query("query") searchTerm: String):
Call<SearchResponse>
}
```

It uses a GET annotation to determine the endpoint for the query and include the apiKey since that doesn't change and then the method itself takes in a parameter that will be passed along as the "query" parameter in our request. The method returns a Retrofit Call object that we can execute to get a response.

## Utility Function for Checking Network

Before we can execute our Retrofit calls, it's a good idea to make sure the device has a working internet connection. Because we'll need to do this multiple times throughout our application, I created a new utility class called NetworkHelper with a method in the companion object to test the connection.

**Create** a new class in our *utils* package called *NetworkHelper*
**Add** the following companion object:

```
companion object {
    //activeNetworkInfo has been deprecated but we can't use the new
alternative because our min API is too low
    @Suppress("DEPRECATION") //suppress the deprecation warnings
    fun networkConnected(app: Application): Boolean {
        val connectivityManager =
            app.getSystemService(Context.CONNECTIVITY_SERVICE) as
ConnectivityManager
        val networkInfo = connectivityManager.activeNetworkInfo
        return networkInfo?.isConnectedOrConnecting ?: false
    }
}
```

You'll notice I added a suppression annotation because we have to work with some deprecated methods in order to accomodate our min API level. All the annotation does is get rid of warnings and you should only use if you're certain that you need to use deprecated features in your function.

## Implement Retrofit Interface in Repository Class

With our interface set up, we implement it in our Repository class to get the data. We do this in our repository class since that is how we are sourcing all of our data.

To get started, we need to create a Retrofit instance as a class property using a builder pattern. We'll use this to set the base URL and the Moshi converter for all the calls that are made using the instance. **Add the following class property to RecipeRepository**:

```
private var retrofit: Retrofit = Retrofit.Builder()
  .baseUrl(BASE_URL)
```

```
   .addConverterFactory(MoshiConverterFactory.create())
   .build()
```

Next, we need to implement our SpoonacularService interface which is done using the create method of our retrofit instance. We'll define the property but do the initialization in the init block since we can't call the create method until the retrofit instance is created.

**Add the property** at the class level:

```
private var service: SpoonacularService
```

And **add the code** to create it in the init block:

```
//init the service instance
service = retrofit.create(SpoonacularService::class.java)
```

At this point we've successfully configured our retrofit instance and we can use it throughout the class to make our API calls based on the abstract method we created.

We'll change the way our getRecipeList method works in the following ways:

- It needs to be async
  - Add suspend keyword and @WorkerThread annotation to denote that this is an async function which should not be called from the UI thread
- Take in a parameter for the search term
- Check the network connection
- Execute the searchRecipes method from our service
- Post the response to the value of our recipeData LiveData object

**Here's the final method:**

```
//search the API for recipes based on a searchTerm
@WorkerThread
private suspend fun getRecipeList(searchTerm: String) {
    if(NetworkHelper.networkConnected(app)) {
        val response = service.searchRecipes(searchTerm).execute()
        if(response.body() != null) {
            //successful request
            val responseBody = response.body()
            recipeData.postValue(responseBody?.results?.toList())
        } else {
            //there was an error with the request (or server)
            Log.e(LOG_TAG, "Could not search recipes. Error code:
${response.code()}")
        }
    }
}
```

We should see an error in our init block where we call the getRecipeList() method. This is because it is now an async method and thus needs to be called from a CoroutineScope. We can use a built in dispatcher to run our async function and which manages a thread pool so we don’t have to worry about anything.

**Change** the simple method call **to the following call** using *CoroutineScope*:

```
//async request for getting the recipes
CoroutineScope(Dispatchers.IO).launch {
    getRecipeList("pasta")
}
```

Now we can **run** our application and we should see our data getting fetched from the API and populated in the RecyclerView!

## Integrate API for Recipe Details

Using the API to fetch the recipe details will now be easy since we’ve done all the configuration work upfront. We just need to add a method to our SpoonacularService interface for fetching recipe details and then make the changes to our repository class.

### Add an Abstract Method to the Interface

First step is to add the method to SpoonacularService. It will look very similar to the method for searching recipes and will also use a GET annotation with a placeholder for the id. The id will need to be passed in as an argument. I got the following pattern from the API docs:

```
https://api.spoonacular.com/recipes/{id}/information
```

**Add the method** to SpoonacularService:

```
@GET("recipes/{id}/information?apiKey=${API_KEY}")
fun recipeDetails(@Path("id") id: Int): Call<RecipeDetails>
```

### Make the changes in the Repository class

Next, we just have to make some changes to the *getRecipeDetails* method of our *RecipeRepository* class. Instead of loading a parsing raw JSON, we will make a request using Retrofit and parse the response based on the id of the selected recipe from the search results. Then all we have to do is post the response to the recipeDetails LiveData object.

**Change** the *getRecipeDetails* method accordingly:

```
@WorkerThread
private suspend fun getRecipeDetails(forRecipe: Recipe) {
    if(NetworkHelper.networkConnected(app)) {
        val response = service.recipeDetails(forRecipe.id).execute()
        if(response.body() != null) {
            recipeDetails.postValue(response.body())
        } else {
            Log.e(LOG_TAG, "Could not find details for ${forRecipe.title}.
Error code ${response.code()}")
        }
    }
}
```

Make sure that your change the Observer that calls this method to handle the new asynchronous nature of the function:

```
val recipeSelectedObserver =  Observer<Recipe> {
    CoroutineScope(Dispatchers.IO).launch {
        getRecipeDetails(it)
    }
}
```

Now we can **run** the app and select any recipe from the list to see the details!

## User Input for Search

We are now able to make API requests and parse the data on the way back asynchronously. Most of the technical work is done for today, but we need to take in user input for the search term and then display the results so we still have some work to do on the view which will require some app architecture changes.

First, we'll create a new fragment that we'll use to display the results and add it to our navigation graph. **Go to** *mobile_navigation.xml* in the *res/navigation* folder. **Remove the action** that currently connects the *navigation_search* fragment to the *recipeDetailFragment*. **Create a new destination** by clicking the icon in the top left of the editor. **Name** it *SearchResultsFragment*. **Add an action** from *navigation_search* to *searchResultsFragment*. **Add an action** from *searchResultsFragment* to *recipeDetailFragment*.

Now, we'll **move the layout xml for the RecyclerView** from the *fragment_search.xml* to our new layout file for the new fragment *fragment_search_results.xml* (cut/paste).

Let’s fix our project structure by **dragging** the new *SearchResultsFragment.kt* file into our *ui/search* package (select Refactor in the dialog that pops up).

Now let’s **move the code for the RecyclerView** from the *SearchFragment* to the *SearchResultsFragment*. **Create properties** in *SearchResultsFragment* for the *recyclerView*, the *SharedViewModel*, and the *Nav Controller* and **instantiate** them *onCreateView*. Also, **add the implementation code** to the class for *SearchRecyclerAdapter.RecipeItemListener* and **use the error to implement** the abstract method.
The SearchResultsFragment should now looks like this:

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

    private lateinit var recyclerView: RecyclerView
    private lateinit var sharedSearchViewModel: SharedSearchViewModel
    private lateinit var navController: NavController

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {

        //instantiate nav controller reference using its id from the xml of the
main activity layout
        navController = Navigation.findNavController(requireActivity(),
R.id.nav_host_fragment)
//        get shared instance of view model
        sharedSearchViewModel =

ViewModelProvider(requireActivity()).get(SharedSearchViewModel::class.java)

        // Inflate the layout for this fragment
        val root = inflater.inflate(R.layout.fragment_search_results, container,
false)

        //find the recyclerview
        recyclerView = root.findViewById(R.id.recyclerView)

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

        return root
    }

    override fun onRecipeItemClick(recipe: Recipe) {
        TODO("not implemented") //To change body of created functions use File |
Settings | File Templates.
    }
}
```

Next, we'll **implement the onRecipeItemClick method** — it will be essentially the same as it was in SearchFragment, but the id for our action has changed. We still need to update the selectedRecipe LiveData object from our shared ViewModel.

```
override fun onRecipeItemClick(recipe: Recipe) {
    Log.i(LOG_TAG, recipe.toString())
    sharedSearchViewModel.selectedRecipe.value = recipe

navController.navigate(R.id.action_searchResultsFragment_to_recipeDetailFra
gment)
}
```

Now let's modify the view for our fragment_search.xml file before we update the class. We'll use the following steps to create a simple layout with an EditText and a button that allows users to search for recipes:

- **Go to** fragment_search.xml
- **Drag** out a plain text editText (text tab of the Palette)
- **Drag** out button
- **Change** editText id to *searchInput*
    - **Change** hint to "Search recipes"
        - Use errors to create string resource
    - **Center** in the parent using constraints (drag each handle to top,bottom,left,right sides)
- **Change** id of button to searchButton
    - **Change** text to SEARCH
    - **Constrain** to left and right sides in center and top to searchInput

With our view properly layout out, we can start to think about how we'll navigate and pass data around. We'll need a new LiveData object in the *SharedSearchViewModel* to store the text the user inputs into the *searchInput*. Add it as follows:

```
val searchUserInput = MutableLiveData<String>()
```

Next, let's modify the SearchFragment class to update this object with the user's input when the search button is pressed using the following steps:

- **Remove** the listener extension and the onRecipeItemClick method since these are no longer needed in this fragment
- **Create** private function called *searchRecipes* to search recipes that will update the searchUserInput liveData object in our ViewModel and fire the navigation to the results fragment
- **Add** class properties for button and edittext in Search
    - Initialize these in the onCreateView
    - Set onClickListener for button to call the *searchRecipes* method

Now we just need to change the repository class to make the API request when the searchUserInput LiveData object is updated instead of just in the init block. We'll do this by creating an Observer function that we will add in the ViewModel.

**Create the observer** in RecipeRepository:

```
val searchTermEntered =  Observer<String> {
    CoroutineScope(Dispatchers.IO).launch {
        getRecipeList(it)
    }
}
```

**Set searchTermEntered Observer as a callback** for the *searchUserInput* LiveData object in the *init* block of *SharedSearchViewModel*:

```
searchUserInput.observeForever(recipeRepo.searchTermEntered)
```

Don't forget to **remove it** in the *onCleared* method:

```
searchUserInput.removeObserver(recipeRepo.searchTermEntered)
```

Now we should be able to **run** the app, enter a search term

You'll notice that data gets left over as we navigate back and forth between fragments. We'll be fixing this during the UI design demo in a couple weeks or maybe before — we'll see.