Recipe App: Favorites Functionality

We have our Room database to persist the user's favorite recipes all set up, now we just need to implement it in the code. We'll do some intermediate transforms between the model class that we use to parse the JSON returned by the API and the Entities we constructed to work with our database. Beyond that, we'll get more comfortable with passing data around between fragments using shared view models and live data.

Changes Since Last Demo

Since the last demo, I made the following minor/repetitive changes to the project:

- Added a field to the Favorite entity title *date_added* to keep track of when a user added a particular favorite
    - Since the Date type is not compatible directly with Room, I created a simple type converter class to transform Date objects to Long types and vice versa for the database to use when persisting and retrieving data
- Did some project restructuring in order to make sure all was packaged well

Demo

I'll be working from the 05-favorites-START branch throughout the demo, so be sure to pull that down before you start working. The finished demo can be found in the 05-favorites-END branch.

Menu Item

We'll need to add a menu item to the action bar in the Detail fragment so that the user can save (or remove) the recipe to their favorites. In order to do this, we will create a new menu resource XML file and inflate it in the fragment.

Create XML

To start, **create a new Menu Resource file**. **Ctrl-click** on the *menu* folder in *res* and **select** New -> Menu Resource File. **Name** it *detail_menu* and **press** OK

Drag a new Menu Item from the palette into the layout and apply the following changes to its attributes:

- **Change showAsAction** to *ifRoom* or *Always*
- **Change id** to favoriteRecipe
- **Change icon** to @android:drawable/btn_star_big_off
- **Change title** to Save (generate the string resource with warning)

Next, **go to** *values/strings.xml* and **add a new String Resource**

```
<string name="remove">Remove</string>
```

Inflate the Menu

Next, we need to **inflate the new menu xml** in the *RecipeDetailsFragment*

Add the following line to enable the options menu inside the onCreateView() method:

```
setHasOptionsMenu(true)
```

Then, **override the onCreateOptionsMenu method** to inflate the xml layout:

```
override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.detail_menu, menu)
    super.onCreateOptionsMenu(menu, inflater)
}
```

Handle Item Selection

Now we need to **override the onOptionsItemSelected() method** in order to listen for when the user presses the favorite button. We'll **toggle the icon and title for the button** based on whether they are saving or removing the recipe from their favorites:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    //toggle icon and title when pressing the star
    if(item.itemId == R.id.favoriteRecipe) {
        if (item.title == getString(R.string.save)) {
            item.icon = ResourcesCompat.getDrawable(
                resources,
                android.R.drawable.btn_star_big_on,
                null
            )
            item.title = getString(R.string.remove)
        } else {
            item.icon = ResourcesCompat.getDrawable(
                resources,
                android.R.drawable.btn_star_big_off,
                null
            )
            item.title = getString(R.string.save)
        }
    } else {
        Log.i(LOG_TAG, item.itemId.toString())
    }
    return super.onOptionsItemSelected(item)
}
```

At this point, we could **run the application** and verify that the button's icon is toggling correctly.

Conversion Between Room Entities and Model Types

Right now, most of our application uses the model classes from Recipe.kt file to represent our data. However, we have separate structures for our database due to the constraints of Room and SQLite.

I decided that only the FavoritesRepository and the Database should work with the Room types while the rest of the application is restricted to the model classes we defined in Recipe.kt. In order for this to work smoothly, let's add some methods to our Recipe and RecipeDetails classes to help us convert to these types from the associated Room types and vice versa.

Let's start with some **methods to convert to/from the RecipeDetails class**:

```
@JsonClass(generateAdapter = true)
data class RecipeDetails (
    val id: Int,
    val title: String,
    val summary: String,
    val image: String,
    val readyInMinutes: Int,
    val servings: Int,
    val analyzedInstructions: List<Steps>,
    val extendedIngredients: Set<Ingredient>
) {
    //methods for converting to and from room entities
    fun getRoomFavorite(): Favorite {
        return Favorite(id, title, summary, image, readyInMinutes, servings,
Date())
    }

    fun getRoomIngredients():
List<com.isaac.recipes.data.database.ingredient.Ingredient> {
        var roomIngredientList =
mutableListOf<com.isaac.recipes.data.database.ingredient.Ingredient>()


        for(ingredient in extendedIngredients) {

roomIngredientList.add(com.isaac.recipes.data.database.ingredient.Ingredient(recipe
_id = id,
                name=ingredient.name,
                unit = ingredient.unit,
                amount = ingredient.amount))
        }

        return roomIngredientList
    }

    fun getRoomInstructions():
List<com.isaac.recipes.data.database.instruction.Instruction> {
        var roomInstructionList =
mutableListOf<com.isaac.recipes.data.database.instruction.Instruction>()

        for(instruction in analyzedInstructions[0].steps) {

roomInstructionList.add(com.isaac.recipes.data.database.instruction.Instruction(rec
ipe_id = id,
                number = instruction.number,
                step = instruction.step))
        }

        return roomInstructionList
    }

    companion object {
        fun fromRoomTypes(fav: Favorite,
                          roomInstructions:
List<com.isaac.recipes.data.database.instruction.Instruction>,
                          roomIngredients:
List<com.isaac.recipes.data.database.ingredient.Ingredient>): RecipeDetails {
            //create ingredient set
            var ingredientSet = mutableSetOf<Ingredient>()

            for(roomIngredient in roomIngredients) {
                ingredientSet.add(Ingredient("${roomIngredient.amount}
${roomIngredient.unit} ${roomIngredient.name}",
                    roomIngredient.name,
                    roomIngredient.amount,
                    roomIngredient.unit))
            }

            //create instruction list
            var instructionList = mutableListOf<Instruction>()

            for(roomInstruction in roomInstructions) {
                instructionList.add(
                    Instruction(
                    roomInstruction.number,
                    roomInstruction.step
                ))
            }

            //construct and return the converted object
            return RecipeDetails(fav.recipe_id,
                fav.title,
                fav.summary,
                fav.image,
                fav.ready_in_minutes,
                fav.servings,
                listOf(Steps(instructionList)),
                ingredientSet)

        }
    }
}
```

While this is a fair amount of work, it will pay off down the road when we want to seamlessly work between the database entities and our data models.

Next, let's **add a companion object to the Recipe class** to convert to that type from a Room Favorite:

```
@JsonClass(generateAdapter = true)
data class Recipe (
    val id: Int,
    val title: String,
    val readyInMinutes: Int,
    val servings: Int,
    val image: String
) {
    companion object {
        fun fromRoomFavorite(fav: Favorite): Recipe {
            return Recipe(fav.recipe_id, fav.title, fav.ready_in_minutes,
fav.servings, fav.image)
        }
    }
}
```

Once again, this will become very useful in a minute.

FavoritesRepository Class

We'll use our new conversion methods to insert recipes into our database and retrieve them in the FavoritesRepository. This method is very simple now that we have some conversion methods to use.

Add the following method to the FavoritesRepository class:

```kotlin
fun addFavorite(recipe: RecipeDetails) {
    CoroutineScope(Dispatchers.IO).launch {
        //insert the favorite
        favoriteDAO.insertFavorite(recipe.getRoomFavorite())
        //insert each instruction
        for(instruction in recipe.getRoomInstructions()) {
            instructionDAO.insertInstruction(instruction)
        }
        //insert each ingredient
        for(ingredient in recipe.getRoomIngredients()) {
            ingredientDAO.insertIngredient(ingredient)
        }
    }
}
```

Shared View Model

We'll use another shared View Model to provide a data bridge between the *RecipeDetailFragment* and the *FavoritesFragment* so let's **change the name** of the *FavoritesViewModel* to *SharedFavoritesViewModel*.

We'll do this by refactoring the name of the class. **Select it and press Shift-F6** then type *SharedFavoritesViewModel* in the red box that shows up. **Press Enter**.

And now since the RecipeDetailFragment will be connected to both the *Search* and the Favorites tabs, **let's move the details package** up a level to be equal with the the *favorites*, *search*, and *adapters* packages (a direct child of the *ui* package)

Finally, we can **create a method** that utilizes our addFavorite() repository method:

```kotlin
fun addFavorite(recipe: RecipeDetails) {
    favRepo.addFavorite(recipe)
}
```

Changes to RecipeDetailFragment

The last step before we can test all the work we've done so far is to implement the *SharedFavoritesViewModel* in our *RecipeDetailFragment.*

Create a lateinit property to store the instance of *SharedFavoritesViewModel*:

```
private lateinit var favoritesVM: SharedFavoritesViewModel
```

Instantiate the property in onCreateView():

```
favoritesVM =
ViewModelProvider(requireActivity()).get(SharedFavoritesViewModel::class.ja
va)
```

Add a property to keep track of the recipe we are displaying:

```
private lateinit var currentRecipe: RecipeDetails
```

Set it in the recipeDetails Observer:

```
currentRecipe = it
```

And **call the addFavorites() method** in our *onOptionsItemSelected()* method:

```
if (item.title == getString(R.string.save)) {
    item.icon = ResourcesCompat.getDrawable(
        resources,
        android.R.drawable.btn_star_big_on,
        null
    )
    item.title = getString(R.string.remove)
     //pass new favorite to view model
     favoritesVM.addFavorite(currentRecipe)
}
```

If we want, we can add some Log statements and run the app to verify that the addFavorites() methods are being called, or we can just move on to the next step and set up the RecyclerView for the favorites to see it in action.

FavoritesFragment Layout

We'll be using another RecyclerView to show our list of Favorites. This

Create the Layout

Open the fragment_favorites layout file
Remove the TextView

Add a RecyclerView using the following steps:

- **Add new recyclerview** from palette
- **Set id** to *favoritesRecyclerView*
- **Add constraints**
  - 0 for top and bottom
  - 8 for each side
- **Set layout_width and layout_height** to *match constraints*
- **Set layout manager** to *LinearLayoutManager*

```
app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"/>
```

Reuse the SearchRecyclerAdapter

In order to populate the new RecyclerView with data, we'll reuse our *SearchRecyclerAdapter* since it's close to what we need.

Let's **refactor the name of the class** from *SearchRecyclerAdapter* to *RecipeRecyclerAdapter*

In order to make this work for our FavoritesFragment without refactoring the Adapter and the SearchResultsFragment, we need to supply the adapter with a *List<Recipe>*. Right now we don't have that data, so we'll make some changes to our view model and repository classes before we can put data into the Favorites RecyclerView.

List of Favorites Live Data

Repository

First, let's get a reference to a LiveData object that holds all the Favorites in our database.

Add the following property to *FavoritesRepository*:

```
val favoriteRoomList: LiveData<List<Favorite>> =
favoriteDAO.getAllFavorites()
```

View Model

Now, we'll make use of this in the FavoritesViewModel to observe changes to Favorites entity and then convert the Favorites to Recipes.

Create a new LiveData object that will store the converted List:

```
val favoriteRecipeList: MutableLiveData<List<Recipe>> = MutableLiveData()
```

Next, we'll **create an Observer** that will take in a *List<Favorite>*, convert to *List<Recipe>*, and update *favoriteRecipeList*:

```
private val favoriteListObserver =  Observer<List<Favorite>> {
    val allRecipes = mutableListOf<Recipe>()
    for(fav in it) {
        allRecipes.add(Recipe.fromRoomFavorite(fav))
    }

    favoriteRecipeList.value = allRecipes
}
```

All that's left is to **add and remove the Observer** appropriately:

```
init {
    favRepo.favoriteRoomList.observeForever(favoriteListObserver)
}

override fun onCleared() {
    favRepo.favoriteRoomList.removeObserver(favoriteListObserver)
    super.onCleared()
}
```

FavoriteRecyclerView Adapter

Now that we have the data in the right format, we can add the adapter to the Favorites RecyclerView in Favorites Fragment.

Create class properties for recyclerView and adapter:

```
private lateinit var favRecyclerView: RecyclerView
private lateinit var adapter: RecipeRecyclerAdapter
```

In the *onCreateView()* method, we'll **instantiate these properties** and then **add an Observer** that will notify the adapter when the LiveData object is updated:

```
val root = inflater.inflate(R.layout.fragment_favorites, container, false)
favRecyclerView = root.findViewById(R.id.favoritesRecyclerView)

adapter = RecipeRecyclerAdapter(requireContext(), emptyList<Recipe>(),
this)
favRecyclerView.adapter = adapter
```

```
//add observer for new favorites in database
favVM.favoriteRecipeList.observe(viewLifecycleOwner, Observer {
    adapter.recipeList = it
    adapter.notifyDataSetChanged()
})
```

We should get an error at this point because we haven't implemented RecipeItemListener in FavoritesFragment. Add it to the class declaration and use the error to create the required onRecipeItemClick method:

```
class FavoritesFragment : Fragment(),
RecipeRecyclerAdapter.RecipeItemListener {
```

Run the app! We can finally see our Favorites getting populated into the RecyclerView in the Favorites tab. Nothing happens when we click yet, but that won't be too difficult since we can easily pass data around with our SharedFavoritesViewModel.

Display Details for Selected Favorite

The last thing we'll do today is show the details for a favorite when it's selected from the list in the Favorites tab.

Update FavoriteDAO

We need a way to look up a specific Favorite from our Favorite Entity by it's ID (you'll see why in a second), so let's **add a new method** to that Interface:

```
@Query("SELECT * FROM favorites_table WHERE recipe_id = :id")
fun getFavorite(id: Int): Favorite
```

Update Repository

First, we'll need to fetch the details from the database for the favorite that the user selects from the list. We'll do this in the repository and make use of LiveData to notify that we've found the details:

Add the following property and method to *FavoritesRepository*:

```
val favoriteDetails: MutableLiveData<RecipeDetails> = MutableLiveData()

fun getDetailsForRecipe(recipe: Recipe) {
    CoroutineScope((Dispatchers.IO)).launch {
        val fav = favoriteDAO.getFavorite(recipe.id)
        val instructions =
instructionDAO.getInstructionsForRecipe(fav.recipe_id)
        val ingredients =
ingredientDAO.getIngredientsForRecipe(fav.recipe_id)

        favoriteDetails.postValue(RecipeDetails.fromRoomTypes(fav,
instructions, ingredients))
    }
}
```

View Model

Next, we'll **create a method** to use *getDetailForRecipe* and **create a reference** to *favoriteDetails* in the *SharedFavoritesViewModel*:

```
val favDetails: MutableLiveData<RecipeDetails> = favRepo.favoriteDetails
//get selected favorites details and update live data
fun favSelected(recipe: Recipe) {
    favRepo.getDetailsForRecipe(recipe)
}
```

Update RecipeDetailsFrag

Next, we'll update RecipeDetailsFragment to show the details when a selected. We'll need to do a bit of refactoring in to do this efficiently.

First, move references to the ui elements from inside onCreateView() to class properties:

```
private lateinit var ingredientListView: RecyclerView
private lateinit var recipeTitleTextView: TextView
private lateinit var instructionsRecyclerView: RecyclerView
private lateinit var imageView: ImageView
```

Next, we'll **create a common observer** that will handle updating the content of the fragment with the details for a recipe:

```
private val updateViewWithDetails = Observer<RecipeDetails> {
    //set the current recipe
    currentRecipe = it

    //set the title in the action bar
    (activity as AppCompatActivity?)?.supportActionBar?.title = it.title
    //set the title textview
    recipeTitleTextView.text = it.title

    //images can be fetched using the following pattern:
```

```
https://spoonacular.com/recipeImages/{ID}-{SIZE}.{TYPE}
    //runs on background thread
    Glide.with(this)
        .load("${IMAGE_BASE_URL}/${it.id}-556x370.jpg")
        .into(imageView)

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

    //setup recyclerview for instructions
    val instructionList = mutableListOf<String>()
    for(instruction in it.analyzedInstructions[0].steps) {
        instructionList.add("${instruction.number}. ${instruction.step}")
    }
    val instructionAdapter = DetailsRecyclerAdapter(requireContext(),
instructionList)
    instructionsRecyclerView.adapter = instructionAdapter
}
```

Now we just need to **add this Observer** to both *recipeDetails* and *favDetails* LiveData objects in the onCreateView() method:

```
sharedSearchViewModel.recipeDetails.observe(viewLifecycleOwner,
updateViewWithDetails)
favoritesVM.favDetails.observe(viewLifecycleOwner, updateViewWithDetails)
```

Navigation Action

All that’s left to do is navigate to the RecipeDetailFragment when the user clicks on a recipe in the Favorites RecyclerView. To do this, we need to **create a new action in our Navigation Graph** that connects the two fragments.

Open *res/navigation/mobile_navigation.xml* and **create the action by dragging from the handle** on the side of *navigation_favorites* to the *recipeDetailFragment*

Make note of the action's ID, but you can leave it as the default.

Update FavoritesFragment

The last thing is to use the app's nav controller to fire the action in the FavoritesFragment
Add a class property to store the reference:

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

Initialize in onCreateView():

```
navController = Navigation.findNavController(requireActivity(),
R.id.nav_host_fragment)
```

And implement onRecipeItemClick():

```
override fun onRecipeItemClick(recipe: Recipe) {
    favVM.favSelected(recipe)

navController.navigate(R.id.action_navigation_favorites_to_recipeDetailFrag
ment)
}
```

Run the app! We should be able to navigate to the details for a favorite.

Next time, we'll take a look at removing recipes from favorites, and start to look at improving the UI/UX design of our app since we have most of the technical side working.