Recipe App: Remove Favorites and UX Design

Today we're going to give the user to remove recipes from their favorites. We'll also be doing a variety of smaller tasks to make our app look and work better for the user.

Demo

I'll be working from the 05-style-START branch throughout this demo so be sure to check that out before you start to follow along. There aren't any major changes between where we left off after the last demo and where we're starting from today.

Check for Favorites

Before we can really start to remove favorites, we need to add the ability to check if the current recipe is already a favorite in the RecipeDetails Fragment. In order to do this, we'll add a LiveData object in our repository that checks if a recipe is a favorite based on the id. Then, we'll add an observer in RecipeDetailsFragment when we create the options menu.

Changes to Repository Class

Make the following additions to the *FavoriteRepository* class:

```
val recipeIsFavorite = MutableLiveData<Boolean>()

fun isRecipeFavorited(id: Int) {
    CoroutineScope(Dispatchers.IO).launch {
        val favorite = favoriteDAO.getFavorite(id)
        recipeIsFavorite.postValue( favorite != null)
    }
}
```

Changes to ViewModel

We need to get references to these new repository members in the *SharedFavoritesViewModel* class so we can access in the *RecipeDetailsFragment*

Add the following function and property to *SharedFavoritesViewModel*

```
val isFavorite = favRepo.recipeIsFavorite

fun isRecipeFavorited(id: Int) {
    favRepo.isRecipeFavorited(id)
}
```

Changes to RecipeDetailFragment

Finally, we can implement the required functionality in *RecipeDetailsFragment*. We'll check if the recipe is a favorite in the *updateViewWithDetails* Observer function, and then we'll add an Observer to *isFavorite* LiveData that updates the menu item's title and icon. I've also added a helper function to help with toggling the state of the menu item since we need to do it in a couple different places.

Add the following line near the top of *updateViewWithDetail* Observer function:

```
favoritesVM.isRecipeFavorited(it.id)
```

Next, add the helper method:

```
private fun setFavoriteMenuItemState(menuItem: MenuItem, title: String) {
    menuItem.title = title
    if(title == getString(R.string.save)) {
        menuItem.icon = ResourcesCompat.getDrawable(
            resources,
            android.R.drawable.btn_star_big_off,
            null
        )
    } else if (title == getString(R.string.remove)) {
        menuItem.icon = ResourcesCompat.getDrawable(
            resources,
            android.R.drawable.btn_star_big_on,
            null
        )
    }
}
```

Next, add the Observer in *onCreateOptionsMenu*:

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

    favoritesVM.isFavorite.observe(viewLifecycleOwner, Observer {
        val item = menu.findItem(R.id.favoriteRecipe)
        if(it) {
            setFavoriteMenuItemState(item, getString(R.string.remove))
        }
        else {
            setFavoriteMenuItemState(item, getString(R.string.save))
        }
```

```
    })

    super.onCreateOptionsMenu(menu, inflater)
}
```

Lastly, make changes to utilize our new helper method in *onOptionsItemsSelected*:

```
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)) {
            setFavoriteMenuItemState(item, getString(R.string.remove))
            //pass new favorite to view model for persistence
            favoritesVM.addFavorite(currentRecipe)
        } else {
            setFavoriteMenuItemState(item, getString(R.string.save))
            //pass removed favorite to view model for deletion
        }
    }
    return super.onOptionsItemSelected(item)
}
```

We can **run the app** at this point if we want to verify that the action item button state is toggling correctly.

Remove Favorites in RecipeDetails

Now that we know whether a recipe is a favorite or not in RecipeDetailsFragment, we can start to think about deletion. There are different ways to delete data from the database using Room: we can use the @Delete annotation on a DAO and pass the entity to be deleted, or we can use a SQL Query to do it. I decided to go with the SQL query since it gives us more control on how the deletion happens. This is useful since we have some foriegn key constraints that could give us problems if the favorite is removed before its associated instructions and ingredients.

Delete Methods in DAOs

The first step is to create the necessary methods for deletion in all of our DAO interfaces.

Add the following method to *FavoritesDAO* that removes a favorite based on the id:

```
@Query("DELETE FROM favorites_table WHERE recipe_id = :id")
fun removeFavorite(id: Int)
```

Add the following method to *IngredientsDAO* that removes all the ingredients associated with a recipe's id:

```
@Query("DELETE FROM ingredients_table WHERE recipe_id = :id")
fun deleteIngredients(id: Int)
```

Add the following method to *InstructionsDAO* that removes all the instructions associated with a recipe's id:

```
@Query("DELETE FROM instructions_table WHERE recipe_id = :id")
fun deleteInstructions(id: Int)
```

Method in Repository Class

Next, we'll **add a function to utilize the new DAO methods** in the *FavoritesRepository*:

```
fun removeRecipeFromFavorites(id: Int) {
    CoroutineScope(Dispatchers.IO).async {
        ingredientDAO.deleteIngredients(id)
        instructionDAO.deleteInstructions(id)
        favoriteDAO.removeFavorite(id)
    }
}
```

Method in ViewModel

Create a reference to this new function in the SharedFavoritesViewModel:

```
fun removeRecipeFromFavorites(id: Int) =
favRepo.removeRecipeFromFavorites(id)
```

Add to RecipeDetailsFragment

Lastly, we just need to **call the deletion method** from *RecipeDetailsFragment* in the *else* block of the *onOptionsItemSelected* method

```
favoritesVM.removeRecipeFromFavorites(currentRecipe.id)
```

Now we should be able to **run the app** and add/remove recipes to our favorites at will!

Loading Indicators

The next thing we'll do is add some loading indicators when we are searching the API or showing the details of a recipe. These are async tasks that involve loading data and we should probably wait until all the data has been loaded before we show the results. While this is happening, we should let the user know that we're working on it.

RecipeDetailsFragment

To start, let's **add some properties** for the TextViews and the ConstraintLayout because we'll need to hide/show those appropriately. We'll also create a property that will let us know when our fragment has been resumed.

```
private lateinit var ingredientTitleTextView: TextView
private lateinit var instructionTitleTextView: TextView
private lateinit var constraintLayout: ConstraintLayout
private var resumed = false
```

Make sure we initialize these in *onCreateView*

```
instructionTitleTextView =
root.findViewById(R.id.instructionsTitleTextView)
ingredientTitleTextView = root.findViewById(R.id.ingredientsTextView)
constraintLayout = root.findViewById(R.id.detailConstraintLayout)
```

Next, **we'll programatically create** a *ProgressBar* widget which is basically just a spinner. We need to also set constraints after we add it to the view and then apply the new constraints to the ConstraintLayout. Add the following code to the *onCreateView()* method:

```
loadingBar = ProgressBar(requireContext())
loadingBar.id = 1
constraintLayout.addView(loadingBar)

toggleLoading(false)

var constraints = ConstraintSet()
constraints.clone(constraintLayout)
constraints.connect(loadingBar.id, ConstraintSet.RIGHT,
constraintLayout.id, ConstraintSet.RIGHT, 8)
constraints.connect(loadingBar.id, ConstraintSet.LEFT, constraintLayout.id,
ConstraintSet.LEFT, 8)
constraints.connect(loadingBar.id, ConstraintSet.TOP, constraintLayout.id,
ConstraintSet.TOP, 32)

constraints.applyTo(constraintLayout)
```

We haven't done any programmatic widget creation yet this semester, but there's tons of documentation available for how to create them and apply constraints so feel free to poke around in the documentation if you'd like to know more.

Next, we need to apply some logic for hiding the content widgets (*ImageView*, *TextViews*, *RecyclerViews*) while we load the data, and then hide the *ProgressBar* and show the content widgets once we actually get the data.

To start, let's **create a helper function** that we can use to toggle the visibility of these elements:

```
private fun toggleLoading(loading: Boolean) {
    if(loading) {
        ingredientListView.alpha = 0.0f
        recipeTitleTextView.alpha = 0.0f
        instructionsRecyclerView.alpha = 0.0f
        imageView.alpha = 0.0f
        instructionTitleTextView.alpha = 0.0f
        ingredientTitleTextView.alpha = 0.0f
        loadingBar.visibility = View.VISIBLE

    } else if(!loading && resumed){
        ingredientListView.animate().alpha(1.0f).duration = 200
        recipeTitleTextView.animate().alpha(1.0f).duration = 200
        instructionsRecyclerView.animate().alpha(1.0f).duration = 200
        imageView.animate().alpha(1.0f).duration = 200
        instructionTitleTextView.animate().alpha(1.0f).duration = 200
        ingredientTitleTextView.animate().alpha(1.0f).duration = 200
        loadingBar.visibility = View.GONE

        resumed = false
    }
}
```

Now, we'll override the *onResume()* method of the Fragment to initially hide the content widgets but show the loading indicator:

```
override fun onResume() {
    //hide content widgets and show progress bar
    resumed = true
    toggleLoading(true)
    super.onResume()
}
```

This is a lifecycle method that is called anytime the Fragment appears to the user. The reason we can't do this in onCreateView() is because that lifecycle method is only called once when the Fragment is first shown.

Finally, we'll show the content widgets and hide the content widgets at the very end of the *updateViewWithDetails* observer by **adding the following call to our helper function**:

```
toggleLoading(false)
```

We can now **run the application** and see the fruits of our labor.

Search Result Fragments

The process of showing a loading indicator when we're searching for recipes is similar to what we just did for showing recipes but with an added twist. We can navigate to the *SearchResultsFragment* from two places: *SearchFragment* and up nav from *RecipeDetails* Fragment. We only want to show the loading indicator when we are navigating from *SearchFragment* since we already have the data when we up navigate from the *DetailsFragment.*

First, let's **add some properties** to get our *ConstraintLayout* and store our *ProgressBar*:

```
private lateinit var loadingBar: ProgressBar
private lateinit var constraintLayout: ConstraintLayout
```

Next, in *onCreateView()*, we'll **initialize these properties and setup the constraints just like before**:

```
constraintLayout = root.findViewById(R.id.resultsConstraintLayout)

//Loading bar with constraints
loadingBar = ProgressBar(requireContext())
loadingBar.id = 1
constraintLayout.addView(loadingBar)

var constraints = ConstraintSet()
constraints.clone(constraintLayout)
constraints.connect(loadingBar.id, ConstraintSet.RIGHT,
constraintLayout.id, ConstraintSet.RIGHT, 8)
constraints.connect(loadingBar.id, ConstraintSet.LEFT, constraintLayout.id,
ConstraintSet.LEFT, 8)
constraints.connect(loadingBar.id, ConstraintSet.TOP, constraintLayout.id,
ConstraintSet.TOP, 32)

constraints.applyTo(constraintLayout)
```

Before we think about hiding/showing the widgets, let's add a property to the ViewModel that we'll use to determine whether the loading indicator should be shown at all. We'll only show it when we navigate down from SearchFragment and not up from RecipeDetailsFragment.

Add the following property to the *SharedSearchViewModel*:

```
var searchLoading = true
```

Now we can **create a helper function to hide/show widgets appropriately**:

```
private fun toggleLoading(loading: Boolean) {
    if(loading && sharedSearchViewModel.searchLoading) {
        recyclerView.visibility = View.INVISIBLE
        loadingBar.visibility = View.VISIBLE
        //we don't want to show the loading indicator unless we perform a
new search

    } else if(!loading && sharedSearchViewModel.searchLoading){
        recyclerView.scaleY = 0.0f
        recyclerView.pivotY = 0.0f
        recyclerView.visibility = View.VISIBLE
        recyclerView.animate().scaleY(1.0f).duration = 300
        loadingBar.visibility = View.GONE

        sharedSearchViewModel.searchLoading = false
    } else {
        loadingBar.visibility = View.GONE
    }
}
```

The last thing in *RecipeDetails* is to **call the helper function** from *onResume()*:

```
override fun onResume() {
    toggleLoading(true)
    super.onResume()
}
```

And also in our *recipeData* Observer that lives inside *onCreateView()*. **Add the following line to the very end of that Observer**:

```
toggleLoading(false)
```

Finally, go to the *SearchFragment* and **add the following line** to the *searchRecipes()* function **before we perform the navigation**:

```
sharedSearchViewModel.searchLoading = true
```

This ensures that we'll only show the loading indicator when we navigate down from Search Fragment. **Run the app** and verify this works

Transitions

Next, we'll look at applying some basic transitions between our fragments as we navigate around. Using preset transitions is pretty easy and there's the option to add completely custom transitions or customize the defaults as well. For now, we'll just add some default animations.

Go to *mobile_navigation.xml*
Select an action from the Graph
Set *enterAnim* to `@anim/fragment_open_enter`
Set *exitAnim* to `@anim/fragment_open_exit`
Set *popEnterAnim* to `@anim/fragment_open_enter`
Set *popExitAnim* to `@anim/fragment_open_exit`

Repeat that process for the remaining two actions

The *enterAnim* and *exitAnim* properties are used when we navigate using NavController.navigate() the *popXAnim* properties are used when we perform up navigation.

Feel free to test out some of the other animation styles to see what you like.

Toasts and Confirmation

The last thing we'll do today is provide some toasts and a confirmation dialog when the user is adding/removing a recipe to/from their Favorites in *RecipeDetailsFragment*. We'll use a couple helper functions to accomplish this.

Go to **RecipeDetailFragment**

Add the following methods and make the changes to onOptionsItemSelect:

```
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)) {
            setFavoriteMenuItemState(item, getString(R.string.remove))
            //pass new favorite to view model for persistence
            favoritesVM.addFavorite(currentRecipe)
            showFavToast("ADD")
```

```
        } else {
            confirmFavRemove(item)
        }
    }
    return super.onOptionsItemSelected(item)
}

private fun showFavToast(type: String) {
    if(type == "ADD") {
        Toast.makeText(requireContext(),"Recipe added to Favorites!",
Toast.LENGTH_LONG).show()
    } else {
        Toast.makeText(requireContext(),"Recipe removed from Favorites!",
Toast.LENGTH_SHORT).show()
    }
}

private fun confirmFavRemove(item: MenuItem) {
    val dialogBuilder = AlertDialog.Builder(requireContext())
    dialogBuilder.setMessage("Do you want to remove this recipe from you
favorites?")
        .setCancelable(false)
        // positive button text and action
        .setPositiveButton("YES") { dialog, _ ->
            favoritesVM.removeRecipeFromFavorites(currentRecipe.id)
            setFavoriteMenuItemState(item, getString(R.string.save))
            showFavToast("REMOVE")
            dialog.dismiss()
        }
        // negative button text and action
        .setNegativeButton("NO") {
                dialog, _ -> dialog.cancel()
        }

    val alert = dialogBuilder.create()
    alert.setTitle("Remove Favorite")
    alert.show()
}
```

Run the application and verify that everything works properly.
We made some good progress on our app's user experience in today's demo. Next week, we'll spend some time with styles, themes, and layout to make the UI look and feel better.