Alexander Vaughn

Food Tracker App Documentation

The purpose of this app is to allow a user to log foods that they've eaten throughout the day and add up nutritional facts automatically. This allows the user to see what they've eaten throughout the day at a glance. This app ultimately makes their lives easier when it comes to managing their health.

Problems and Solutions

View Pager (Fragments vs. Recycler):

Initially I wanted to try and use fragments to organize the UI into different pages. When I first started this project, I didn't quite know how fragments worked and I couldn't get them to work the way I wanted.

My solution was to opt for recycler views instead. Recycler views are more intuitive to use for people who are newer to Android studio and application programming. In hindsight, this created more problems down the road which is usually how things work when you try to do things lazily. Now that I know how fragments work, I will be using those in the future for similar designs.

Updating Pages in View Pager:

The major problem with trying to use recycler views for view pagers is that it can be difficult to update the data on a page if it isn't currently bound to the recycler view via the adapter. For example, the app starts out on the overview page. At this point none of the other pages have been created yet because they haven't been needed. So, it was difficult to fill the My Foods page with data from the database. Other problems arose when I was trying to add, remove, or edit items in recycler views.

My solution to this problem was to use a combination of the command pattern and coroutines. The command class I designed would specify the code to be executed when a specific page was brought into view. I created a command manager to maintain a list of commands and execute them at the appropriate times. Each execute function would be executed inside a coroutine.

The reason for the coroutines is that there would be a delay between when a page was brought into view and when its reference was reassigned by the page adapter, meaning that the reference could sometimes be null. The Coroutine would simply wait until that reference became non null and then would execute the code. I overrode the onPageChange method from the ViewPager class and whenever a new page was accessed, I would pass that page into the command manager which would then execute any pending updates associated with that page.

Software Design

Controller:

The primary object that controls the flow of information inside the activity.
To begin the application, this receives a reference to the main activity.
Contains two primary components:
1. Model which contains data.
2. MainView which controls the display of information in the UI.
The model is created first, then the view.

Model:

Maintains data. Contains two primary components.
1. FoodDatabase which is a SQLite database.
2. WebData which can access a web api and retrieve new food data.

todaysFoods - A list of foods for today that the user has logged. Presented in the overview.
myFoods - A list of food objects that mirror what is in the databse.
addFood - Add a new food to the database and myFoods.
removeMyFood - Remove a single food from the database, myFoods, and todaysFoods.
removeAll - Empty the database, myFoods, and todaysFoods.
populate - Tells the database to fill itself with dummy data. The database must be wiped before filling it with dummy data.

MainView:

Controls how information is presented in the UI. Makes use of drawables and layout XML's in the res folder.

viewPager - Information in the view is presented in pages. The list of pages can be found in the Pages enum.
selectingMyFood - This is true when the myFoods page is visible and the user is adding a new food to todays log.
proxyMyFoodsSearch - When a food is being searched for in the myFoods page, this list will contain the foods that match the search (tag or name). When getMyFoods is called (by the foodListAdapter or any other place) if this list is not null, the data within this list will be return instead of the actual list of myFoods retrieved from the model.
images - Cached bitmaps to be reused by FoodItem ViewHolders in recyclers.
loadImages - Initializes the cached bitmaps.
setOnPageChange - Set the callback for when a page is changed. Calls leavePage for the current page. Tells the command manager to make any updates to the new page. Then sets the current page.
getCurrentPageView - Returns the view inside of the current Page.
getMyFoods - Returns myFoods from the model, or the proxyMyFoods if its not null.

Food:

Used by Model to store information about a food. This is used to create items in the Recycler view.

foodFromData - Creates a new Food object from data pulled from the web.
tagsStringToTagList - Takes a string of tags formatted like 'tag1,tag2,tag3' and turns it into a list.
totalFoods - Receives a list of foods, creates a new food object with the sum total of all the fields.
tagsToString - Creates a string formatted like 'tag1,tag2,tag3' from the list of tags.
FoodData - Holds data for a food. Can be constructed from a json file. Used by WebData to retrieve data from the web. Since this is serializable, DO NOT change the field names.
FoodDataList - Holds a list of FoodData. Can be constructed from a json file. Used by WebData to retrieve data from the web. Since this is serializable, DO NOT change the field names.
ImageIds - Pairs an ID to an image.
imageNameToId - Converts a string into an ImageId.
imageIdToName - Converts an imageId as an int into a string.

FoodDatabase:

A sqLite database. Stores food information.

CREATE_TABLE - Sql statement to create the food table.
DUMMY_DATA - Sql statement to insert dummy data.
reset - Resets the database by dropping it and recreating it. The resulting database will be empty.
insertDummyData - Uses the dummy data sql statement to fill the database.
cursorToFoodList - Extracts the data from the received cursor and returns an array list of food.
generateNewId - Creates a new unique food ID.
all - Get a food arrayList of all the foods stored in the database.
add - Inserts a new food into the database.
edit - Edits a food in the database to match the data in the received food object. Finds the food to edit by food id.
remove - Removes the food in the database using the received id.
removeAll - Wipe all data in the databse.
queryNameOrTag - Returns a list of foods from the database where the received query string is contained in the name string or the tags string.

WebData:

Makes queries to calorieninjas.com to get nutritional information about a food. Networking code, such as data retrieval, should never be done on the main thread. Use withContext with dispatcher to execute code on the main thread, an I/O thread, or a default thread.

search - Query's the web API. Once a result has been returned, the doWithResult lambda function will be called, it will be passed the result.
getData - Query's the web API, returns the result.

CustomViews:

Various custom classes/views that can be used in a layout.xml. Having these custom views makes the code very modular and reusable. The custom classes are:

OverView, DayReport, SingleFood, CustomImage, FoodRecycler, Navigation, MyFoods, Selector, CreateNew, Editor

ViewCommands:

CommandManager - Manages the commands that should be executed when a new page is brought into view.
commands - A list of commands that should be executed when a specific page is brought into view.
newCommand - Queue a new command.
executeRelevantCommands - Goes through the stored list of commands, if the new page being brought into view is the page assigned to the command, that command will be exectued.
Command - A specific action is defined in the doExecute function. When execute is called, the doExecute code will be run in a coroutine. The received page specifies that the execute function should be called the next time that page is brought into view.
doExecute - Waits for the page to be fully loaded by the adapter and then executes the command.

Command variants are: UpdateMyFoods, ClearTodaysFoods, RemoveSingleTodaysFoods, EditSingleOverview, EditSingleMyFoods

PageAdapter:

Adapter for the ViewPager in MainView. The user may swipe left or right or use the navigation bar to cycle through the pages. 1 page takes up the whole screen, except for the navigation bar. The ViewHolder items for this recyclerView is a Page class. There are 4 pages. Overview, MyFoods, and CreateNew are accessible by swiping or via the navigation bar. The 4th page, the editor, is locked by default, it is only accessible through the CreateNew page or by clicking edit on a FoodItem.

Pages - This enum is used for matching a page layout to an index.
pages - Holds references to each page. Initialized to null. When a page is created or bound, the reference will be updated in this list.
enableEditor - Unlocks the editor page.
disableEditor - Locks the editor page.
leavePage - Called by MainView when a new page is being moved into view. This function is typically used to cleanup the page that is being left (such as emptying fields in the editor).
Page - ViewHolder used in the ViewPager recycler/adapter.

FoodListAdapter:

Adapter for a FoodRecycler. Manages displaying food items within.

FoodListType - The FoodRecycler is used in two places. This enum specifies which recycler the adapter is adapted to and adjusts what is visible accordingly.
FoodItem - The ViewHolder that will be used inside of the foodRecycler.
onBind - Called when the adapter for the foodRecycler binds a food with a FoodItem.
updateFood - When a food has been edited, this will be called to update the view.
onClickRemove - Remove the item from today's Overview. This button will only be visible in the today's Overview.
onClickEdit - Edit the selected food in the database. This button will only be visible in MyFoods.
onClickDelete - Remove the item from MyFoods and the database. This button will only be visible in MyFoods.
collapse - Hides various elements of this view and plays an animation.
expand - Shows various elements of this view and plays an animation.

Project Requirements

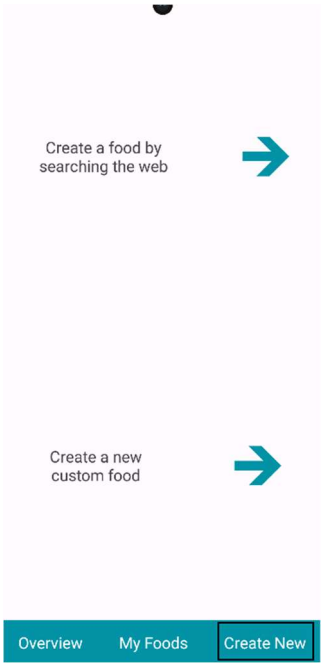
SQLite Database:

The SQLite database is managed by the FoodDatabase class. The structure of the database is depicted below.

```
private const val CREATE_TABLE = """
        CREATE TABLE IF NOT EXISTS "foods" (
        "id"            INTEGER NOT NULL,
        "name"          TEXT,
        "calories"      INTEGER,
        "servingSize"   REAL,
        "totalFat"      REAL,
        "saturatedFat"  REAL,
        "protein"       REAL,
        "sodium"        REAL,
        "potassium"     REAL,
        "cholesterol"   REAL,
        "carbs"         REAL,
        "fiber"         REAL,
        "sugar"         REAL,
        "tags"          TEXT,
        "imageId"       INTEGER,
        PRIMARY KEY("id")
        );
    """
```

Add an Item:

To add an item, navigate to the create new page. Here you can create a food from scratch, or pull in data from the web.



Delete or Edit an Item:

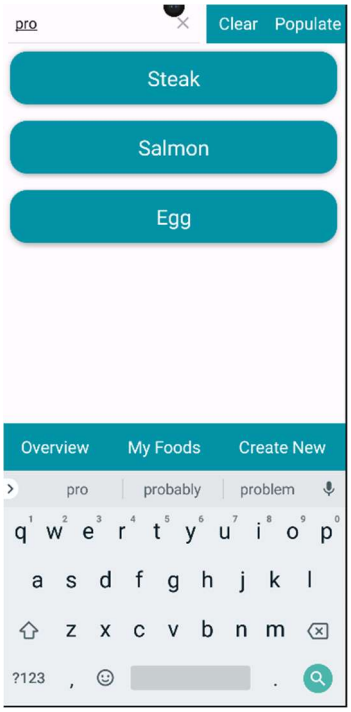
To delete or edit an item, navigate to the my foods page. Select the food to reveal its options.

Clear or Populate Database:

Navigate to the My Foods page, here you may clear or populate the database.

SQL Query:

Navigate to the My Foods page. When text is entered into the search bar, the database will be queried for foods that contain the query either in their name or in their list of tags. In this example, you can see the user is entering in the tag “protein”.

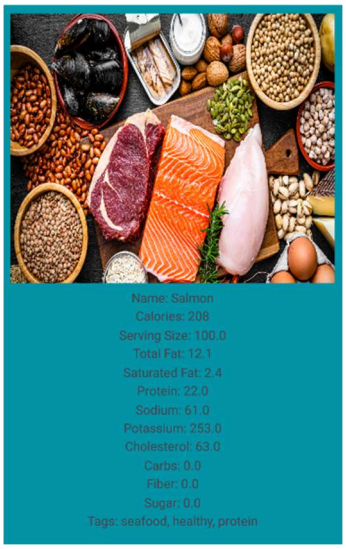


Web API:

The web API this app uses belongs to calorieninjas.com. The data retrieved from it comes in formatted as JSON text. @Serializable is used to fill the data classes FoodData and FoodDataList.

Custom View:

The custom view draws an image using the onDraw method. This is used in the FoodItem ViewHolders in the RecyclerViews.



Additional Functionality:

1. Maintain Personalized Food Items (can be edited from the web or created from user input)
2. Maintain A Log of Today's Foods

   Add or Remove an Item from Today's Log:

   To remove an item from today's log, navigate to the overview, select the item and press remove. To add a food, you may press the "log another food" button.