
GitHub Username: Munifrog (https://github.com/munifrog)

Recipe-Q (where Q represents queue or list)

Description

This Android app has two main jobs: (1) search for recipes and (2) maintain a list (presumably for shopping, but not necessarily).

Additionally, users can save links to favorite recipe discoveries. Recipe ingredients can be ported to the list with custom entries encouraged. As list items are swiped, they are hidden from the user's view, with an "UNDO" action always available.

Ads will be inserted between recipe queries for a free version.

Finally, as a stretch goal, the List Activity could accept items sent from external Apps.

Intended User

The list portion of the App would be useful for anyone who uses a list (anyone, really). The recipe portion of the App is useful for people who cook or acquire ingredients.

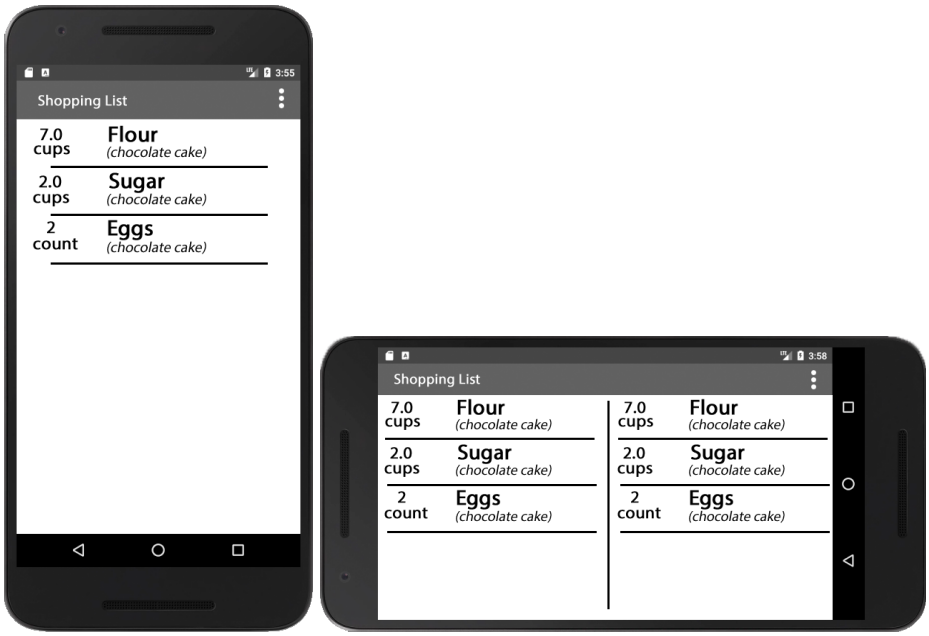
Features

- Create using Android Studio 3.5.2
- Write code (non-XML) solely in the Java programming language
- Maintains (Shopping) List
    - Remembers item state
        - Found / Purchased or not
        - History updated for undo actions
    - Receives custom list items (additions)
    - Receives requests to remove recipe contents (subtractions)
        - Check if recipe exists, before subtracting amounts
    - Show where each item came from (i.e., what recipe(s), app, or custom / blank)
    - Combine quantities for same ingredients from different recipes
        - Show comma-separated list of recipes as sources
- Provide search query options
    - Fragment for standard search terms
    - Fragment for ingredient searches
    - Fragment for nutrition searches
    - Allow user to specify which query fragments to enable
    - Allow user to specify number of recipes to retrieve
    - All queries through complex search engine
- Display recipe search results
    - Display as either list or grid, depending on screen width
    - Provide images, if possible
    - Clicking on the recipe opens the recipe details activity
    - Same view as popular or random and favorite
- Displays popular or random recipes
    - Include food joke of the day
        - Retrieved once at startup, and cached
    - The API endpoints for both (1) recipe of the day and (2) popular recipes is not clearly advertised in the API documentation, but the main website displays them automatically, suggesting it could be possible
    - If API calls for recipe of the day and popular recipes cannot be found, replace this page with random recipes, which there is a clear API endpoint for
    - Clicking on the recipe opens the recipe details activity
    - Same view as favorite and search results
- Remember and display links to favorite recipes
    - Use links for permanent saves
    - Use cache for each search performed
        - Define a garbage-collecting service to purge results after an hour

        - Provide a way to refresh the recipe whose details were or are viewed ...
            - … most recently
            - … currently / actively
    - API end user license agreement specifies that recipe results can only be cached for an hour, (with some exceptions, such as the list of ingredients)
    - API also limits the number of API calls that are made with the key, decrementing the available point balance by a certain number of points for the call, and a smaller amount for each result returned
        - For this project, it will be sufficient to use a single API key
        - I could ask each user to register for their own API key, if this were a production system
        - If someone clones this GitHub project, they can obtain their own API key
- Display recipe details
    - Scrollable
        - Two columns when horizontal
    - Title
    - Action to query for similar recipes
    - List of ingredients
        - Send to shopping list
        - Recall from shopping list
    - List of directions
    - (Stretch Goal) Serving multiplier
        - Disallow negative entries
        - ¼, ½, 1x, 2x, 3x, 4x, 5x, 6x
        - Input field(s)
            - Possibly a spinner
            - Could be single float
            - Could be double integer for a quotient
                - Numerator
                - Denominator
                - Makes numeric reference updates tricky
        - Ingredients multiply numeric references
        - Directions multiply numeric references
    - Miscellaneous
        - Calorie estimate
        - Carbohydrate estimate
        - Fat estimate
        - (Optional) Nutritional information
        - (Optional) Cost estimate
- Show Ads between searches or recipe views (i.e., Free and Paid Versions available)

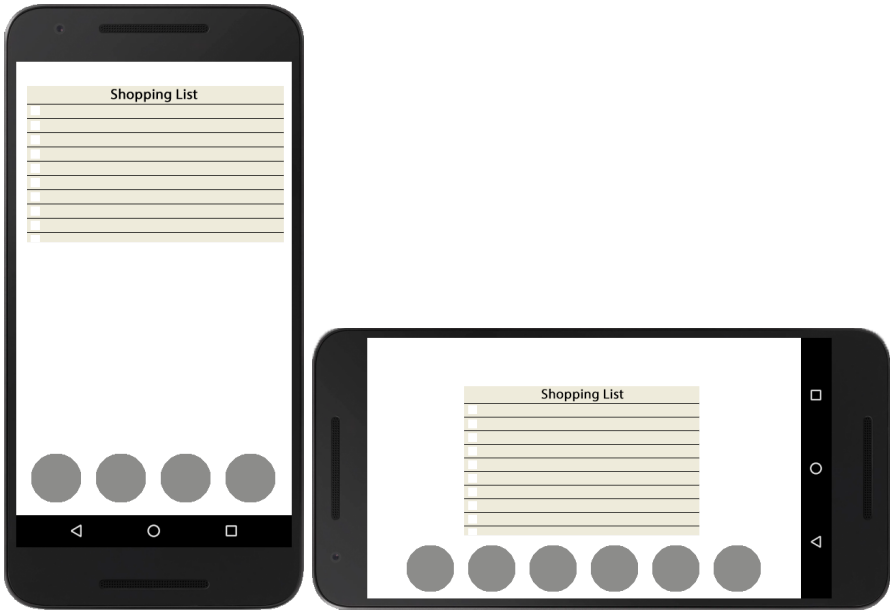
User Interface Mocks

Shopping List Mock



The shopping list is fairly simple: a scrollview with “to purchase” (or “to find”) at the top, and “purchased” (or “found”) at the bottom (or possibly right on horizontal screens). Swiping an item left or right will move the item between lists. Since this action should minimize mistakes (as opposed to just clicking the item), the “Undo” action can appear as an item in the collapsed menu. Menu will have options for Favorite Recipes, Popular or Random Recipes, Undo, and possibly Redo, if there is time to implement it. As an alternative to two lists in horizontal mode, this would be a good place to implement collapsing toolbar techniques for material design.

Widget



Note that the widget will similarly perform the shopping list. But it will merely display the list items, rather than allow movement of or even marking with checkboxes (unless widgets allow for these). Similar care will be performed to combine amounts for same-named ingredients: e.g., 2 cups of chocolate chips plus 1 cup of chocolate chips would become 3 cups of chocolate chips.

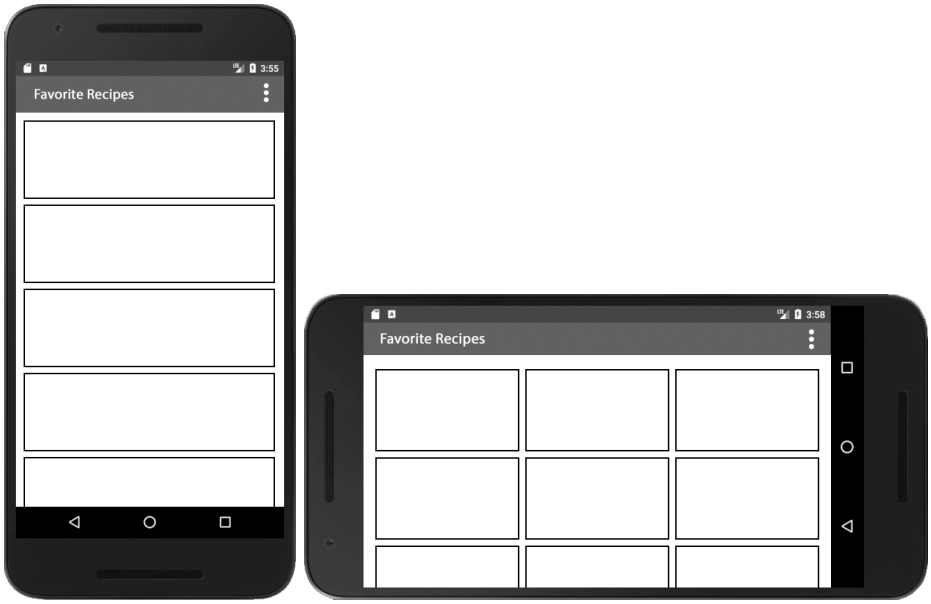
Recipe Details

The Recipe Details will display, at a minimum, the recipe name, ingredients, directions, and button for marking as a favorite, and allow users to send the ingredient list to the shopping list. As stretch goals, users could retract adding the ingredients to the shopping list, display various metrics regarding the recipe, and even multiply the total servings. The menu will have options for launching the Shopping list, Popular or Random Recipes, and Favorite Recipes.

New Search

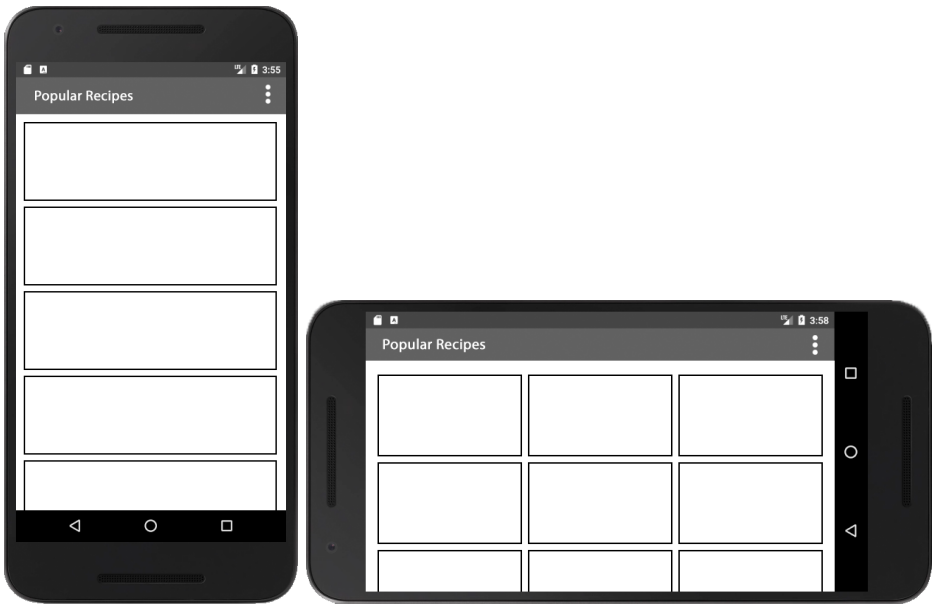
The New Search page will generate the complex-search API call and make the request. When the results return, the Search Results page will launch or display automatically. This search definition page will be a scrollview composed of a launch button at the bottom and three separate fragments, each covering a specific type of search: standard (including cuisine type, diet and intolerances like nuts), ingredient-based, and nutrient-based. The user will enable/disable each of the fragments, but all API calls will be passed through the complex search endpoint, unless it is discovered that the simpler searches are noticeably or significantly faster.

Favorite Recipes



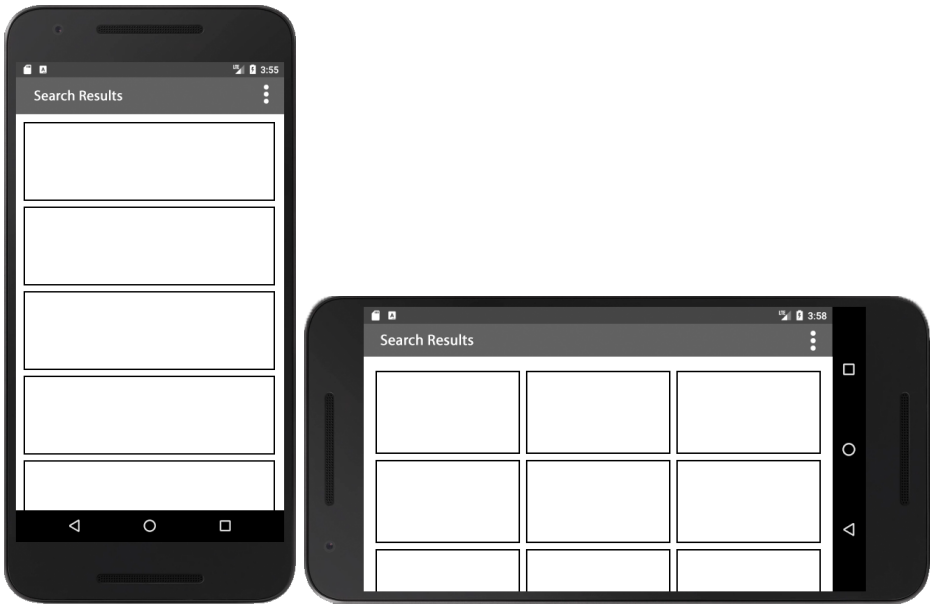
The favorite recipes will display as a list or grid, depending on the device orientation. Most recently added recipes will appear first or at the top. Clicking on a recipe will launch the details for that recipe. The menu will have options for launching a new search, opening the shopping list, viewing the popular/random recipes page.

Popular or Random Recipes



The Popular or Random Recipes page will display as a list or grid, depending on the device orientation. Clicking on a recipe will launch the details for that recipe. The menu will have options for launching a new search, opening the shopping list, viewing the favorites page.

Search Results



The Search Results page will display as a list or grid, depending on the device orientation. It will normally display the search results from the most recent search, but we could display all within-hour cached results under certain conditions (i.e., explicitly from the menu). Clicking on a recipe will launch the details for that recipe. The menu will have options for launching a new search, opening the shopping list, viewing the popular/random, favorites, or recent search results pages.

Key Considerations

How you will support accessibility?

Most of the content on the page will exist as key (or label) and value pairs, laid out in mostly vertical fashion. As appropriate, each button or content field will have both a description and values specified for “android:nextFocusUp”, “android:nextFocusDown”, “android:nextFocusLeft”, and “android:nextFocusRight”. It seems appropriate that the left and right focus-next, switches between the label and value, and up and down focus-next switches between just labels or just values. Alternatively, the focus could simply navigate from label to value to label to value, etc., in one big circuit. However the focus is assigned, each field or label should be focusable, and the bottom and top fields will be connected.

Each fragment used for the complex search functionality should navigate through its own fields, and include the checkbox (or whatever else gets used) which enables or disables the fragment. If the fragment is disabled, navigation should proceed to the next fragment, skipping the contents of the disabled fragment. If all fragments were disabled, navigation would go:

- Enable fragment A option
- Enable fragment B option
- Enable fragment C option
- (Potentially disabled) launch search button
- Enable fragment A option
- etc.

Continuing the analogy with fragment B enabled, and assuming two options for it, navigation would proceed as follows:

- Enable fragment A option
- Disable fragment B option
- Field B1
- Field B2
- Enable fragment C option
- Launch search button
- Enable fragment A option
- etc.

How resources will be stored in the project including colors, strings, and themes?

Most resources, such as layouts, strings, dimensions, and various integer, boolean, or other default primitive values, will be stored as XML files. Some constant values, e.g., JSON tags, may exist only in the code that uses them.

Layouts will define *Start and *End positional attributes rather than, or in addition to, *Left and *Right positional attributes so that RTL can be enabled.

How will your app handle data persistence?

Data persistence will be handled using ViewModel, Rooms, and separate databases for (1) favorite links, (2) grocery list and short-term history, (3) short-term popular/random recipe cache, (4) short-term search results cache.

A database-querying and garbage-collecting IntentService will ensure recipes are not cached longer than one hour, and prompt a refresh for the current/active recipe (requiring an additional API call, as specified by Spoonacular). This service will essentially send a time-delayed event to itself to query the database after an hour has passed. Once the database is empty, it stops sending itself this time-delayed event to query the database and clean up.

How will your app handle backend communication? Specifically, how will it implement one or more of the following: SyncAdapter/JobDispacter or IntentService or AsyncTask?

Each database will be queried using an AsyncTask that defines a listening interface. When the query has returned, the listener will be notified. This listening interface may also include a method for catching internet failures.

The List activity database will have items added to it either directly or using an IntentService that then adds to the database and perform the insertions. A separate intent will be called to display the list contents. At display-time similar ingredients would be combined for the view.

The widget will update at the same time the list does, either by having the list immediately call the widget with its updated contents, or by sending intents for both the list and the widget at the same time. The first might be easier since it would contain all the ingredients to display, and not require the widget to remember, or recalculate additions or subtractions when that work may have already been done.

Describe any edge or corner cases in the UX.

Most of the navigation between activities will be initiated by selecting an action from the current activity menu. Pressing back should return to the previous activity.

Users can open the recipe they are purchasing ingredients for, if they long-click on the ingredient in the shopping list. If ingredients have been combined, a dialog must prompt the user to specify which recipe to view.

When hiding search options (standard, ingredient, or nutrition fragments), the respective search criteria will not be included in the search query.

When recalling a recipe from the list, the list manager must determine if the recipe has been added, and only then remove all of the ingredients for the recipe. This would involve sending the entire list of ingredients. It may help to include the API ID for the recipe since that will be unique.

Describe any libraries you'll be using and share your reasoning for including them.

Plan on using Picasso (**com.squareup.picasso:picasso::2.71828**) for image loading, since it handles the background edge cases automatically.

(Optional) Past projects have disallowed the use of third-party libraries, such as RetroFit (**com.squareup.retrofit2:retrofit:2.6.2**), for parsing JSON. Since RetroFit is allowed for this project, see how it performs relative to writing custom JSON parsing code.

Describe how you will implement Google Play Services or other external services.

Use Firebase Console (**com.google.firebase:firebase-messaging:20.0.1**) to push suggested recipes to the devices, possibly only those that are subscribed for these pushes.

Use Google AdMobs (probably **com.google.firebase:firebase-ads:18.3.0** since Firebase Console will already be in use; could instead use a standalone AdMob provider with **com.google.android.gms:play-services-ads:18.3.0**) for creating a free version that displays interstitial ads (after menu clicks or between searches) and a paid version that has no ads.

Next Steps: Required Tasks

Here is a list of tasks that will help break the project down into manageable portions.

Task 1: Project Setup: API Calls to Database Storage

Create the methods that call the API endpoints

- Complex search
- Joke retriever
- Random recipes
- Similar recipes
- (If found) Popular Recipes
- (If found) Recipe of the Day

The Spoonacular website has a description of their various API calls. Initially, perform hard-coded calls for random recipe retrieval, joke retrieval, similar recipes, and a complex

search: perhaps for the Ketogenetic diet minimizing Potassium and using ingredients in my fridge.

Create the method that parses the JSON response

- Hard-coded JSON tags expected
- Or, implement RetroFit (possibly later)

After making API calls, it is necessary to parse the responses into something that can be displayed in a simple activity. This might be a good time to implement RetroFit, but that is optional, for now.

Create database objects

- List objects
- Recipe search-result objects
- Recipe link objects

With the recipe results parsed, this is a good time to create database objects for storing the results. There will be shopping list objects; recipe search result objects for keeping a short-term cache; and recipe link objects for long-term memory of favorites. We also want these objects parcell-able, but that is easier to do after we know these database objects work with their respective database (see later).

Create the Rooms databases

- Search-results cache
    - All
    - Most recent
- Favorite links cache
    - Image
    - Title
    - URL
- Retrieve hour-old recipes (search results only)
- Purge search result items (set)
- Purge recipe link items (individual)
- Purge list items (as a set or individually)
    - Use swipe timestamp for history
    - Undo restores the most recent ingredient swiped
- Create service for automatically removing search results database objects
    - When recipe is retrieved, launch a timer for an hour later
    - Initially, just remove the object
    - Later, we will prompt for refresh on an open item

Create the databases for the list, search results, and links. Also provide a way to retrieve items that are older than an hour (just for the search results) and correspondingly purge a set of

database objects (in all three databases). Until we can purge based on time, we will have to remove all existing database objects. Since the hour time-limit only applies to the recipe results, we will start a timer that will call the service an hour after receiving the recipe search results from the API endpoint. Multiple recipes are returned with the call, so expect to remove several at once. Might be good to perform the purge at startup as well.

Once the databases are ready, we can upgrade our database objects into parcell-able objects for easier transfer between activities, or upon screen rotation.

Create the ViewModel that manages the databases

- Returns appropriate database for the current state/view
- Search results
- Random recipes
- Popular
- Favorites

With the databases ready, we need them to persist within a ViewModel. The ViewModel will return the database objects appropriate for the current activity. For example, if the favorite links are currently viewed, the favorites-database will be returned by the ViewModel.

Task 2: Implement UI for Each Activity and Fragment

Create recipe results item view

- Grid adapters for RecyclerViews
- Use hard-coded API call for obtaining results

In preparation for displaying all the recipes, we first need to define how a single recipe (item view) will appear in the list or grid. (Note that a list could be managed as a single-column grid.) This step involves defining the grid adapter that will be used with the RecyclerView.

Create search results activity

- Search results
- Random / Popular
- Favorites

With the item view in place, we can define the grid that displays the recipes. Search results, popular or random recipes, and favorites, can all use the same grid to display their separate contents. And we can use a hard-coded API call for obtaining results to display. (i.e., varying searches will be performed later.)

Create List Activity

- Create service to add items to list database
- Create list item view

- Create linear adapter for RecyclerView
- Send array from search results activity to list activity
- Populate list with received list objects
- Implement undo using swipe timestamps
- Implement non-interactive widget (unless it can handle checkboxes)

With results available to send to the list activity, and a list database already in place, this is a good time to prepare the list activity for displaying and storing received items. Create a service that adds received items to the list database without launching the activity. Then create an item view and linear adapter for displaying the database contents. Provide a way (e.g., swiping) to switch item state between searching and found (or purchased). Also maintain a history of swipes for easy undo. This can be done by setting the item swipe timestamp, and clearing the same on “UNDO” action. This is also an ideal activity to implement some collapsing toolbar techniques.

This would be a good time to implement the non-interactive widget that will display the list items.

Create the Complex Search activity

- Fragment for standard searches
- Fragment for ingredient searches
- Fragment for nutrition searches
- Provide a way to enable/disable each fragment in search composition activity
- Perform a search, (latter show an interstitial ad), and automatically launch search results activity

With the application currently showing hard-coded or favorite results, now provide a way for the user to perform their own recipe searches. The Spoonacular site has four kinds of searches: (1) standard or unnamed (2) by ingredients (3) by nutrition, and (4) complex which includes the three previous search options. This naturally lends to defining (individual) fragments for the first three, combined into a single complex search. The easiest way to enable or disable the fragments would be with checkboxes, and while that may be done initially, it would be desired to find a more aesthetically pleasing option.

Task 3: Provide Free and Paid Versions

Using Gradle (**com.android.tools.build:gradle:3.5.2**), provide free and paid versions of the app

- Interstitial ads between searches and activities
- (Potentially) embedded ads within list activity

With the activity functioning as a recipe finder and shopping list, this is a good time to provide some ads for a free version. This will involve defining alternate activities where ads will be inserted. Interstitial ads would go well before or after searches, and is the minimum that will be

accomplished here. Embedded ads would be better suited for the list since swipes probably will occur too frequently to warrant an interstitial ad.
