University of Ottawa

Final Report on CookHelper Application

Brennan McDonald, Rock Liang, and Karim Alibhai
Introduction to Software Engineering
Professor El-Sawah
December 6, 2016

Table of Contents

Introduction

The application that this report pertains to is called ‘CookHelper’. It is tasked to be a simple recipe manager for cooks wishing to share their recipes as well as for people who wish to access recipes that meet their needs without doing much researching. Between these two users, we decided to maintain focus on the recipe viewers as opposed to recipe maintainers. As a result, our design decisions were more focused on the searching aspects of the application than the recipe creating aspects.

In this report are outlined the decisions we made towards workflow, testing, user interface principles, data storage, and searching as well as the challenges we faced during the implementation of each.

Workflow

The application was built using an agile workflow but as two separate application entities. The first of these entities is the user-interface of the application. More details on this entity is available under the section “User Interface.” The second part of the application is a library responsible for the underlying logic. This library was targeted to handle data storage, searching, and the fundamental manipulation functions over the dataset (create, read, update, and delete).

The repository of the library source code was integrated with Travis CI, a continuous integration platform. This platform spins up a new linux container when the code base is updated and runs unit tests on the latest revision of the code base. This helps us ensure that all dependencies are well defined and all compilation instructions are stated clearly. Unit tests were created at a per-class basis where each test was of a major class in the library.

Due to these design decisions, it was vital for us to build the library in a cross-platform manner so that it would be buildable and testable in a linux-based environment as well as function within the Android environment.

User Interface

When designing the user interface (UI), we decided to abide by Google's material design principles because it gave us two big advantages: (i) it is a beautiful and simplistic standard and (ii) it blends into the native Android environment. Due to the ease of integration we were able to create a simple and good looking UI.

On the home screen of the application, we wanted to ensure that users coming from little to no knowledge of the application would not have trouble understanding the purpose of the application (as seen in Figure 1). To further improve the user's experience in terms of understandability, we also added a help section which can be activated by visiting the 'Help' section of the navigational drawer.

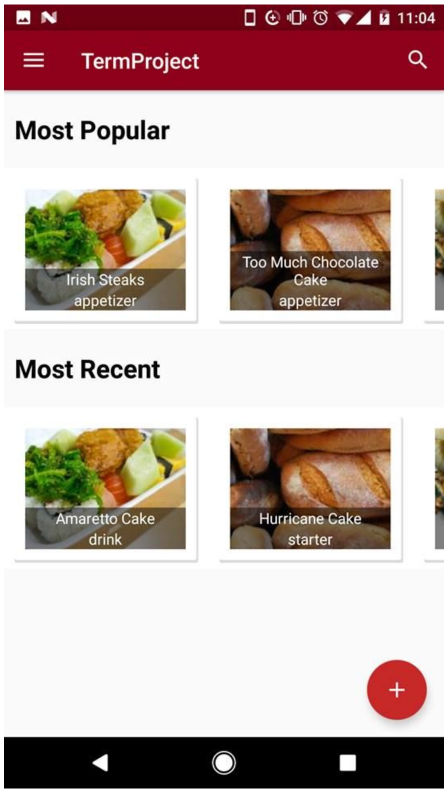

Figure 1: Home screen at first launch.

The navigational drawer is a menu-type component standardized by material design in terms of a sidebar that appears only when requested. It allows us to pack away certain configuration options such as the Help menu into a separate component so that it does not interfere with an average user's experience.

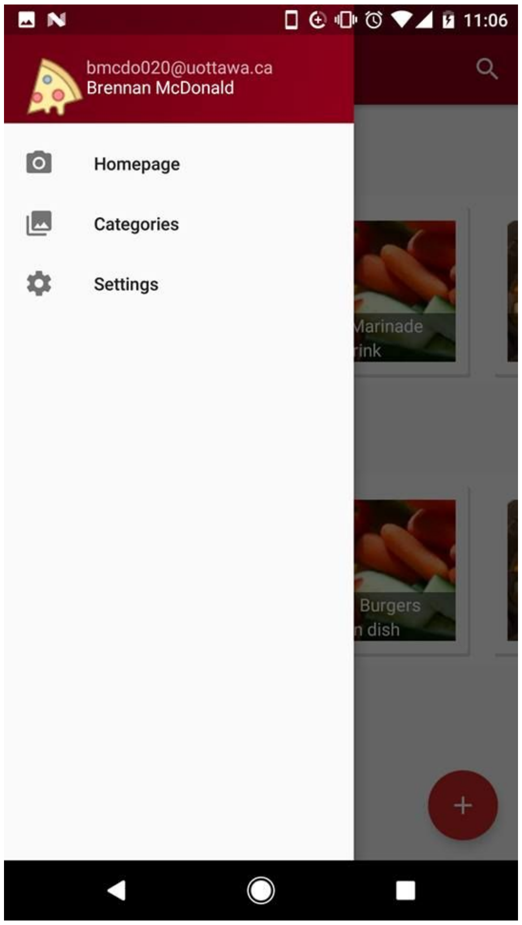

Figure 2: The navigation drawer opened on the home screen.

Many small adjustments were made to the UI implementation to further improve user experience. For instance, an overlay is laid on top of all components except the search box when the user activates the search. This is to ensure that all focus is drawn towards the component that the user is actively interacting with.

Data Storage

The problem of data storage within CookHelper had a few basic feature requirements: (i) it had to be permanent data storage so that storage was not dependent on device capabilities, (ii) it had to be cross-platform due to our workflow decisions, and (ii) it had to be easily searchable without requiring the application to download unnecessary data to the user's device.

As a non-functional requirement, all data models were built to support serialization and deserialization from JSON using Jackson (the exact models are available in Figures 3 and 4). Due to this, preference would be given to databases that support the storage of JSON objects directly (as opposed to saving them as strings).

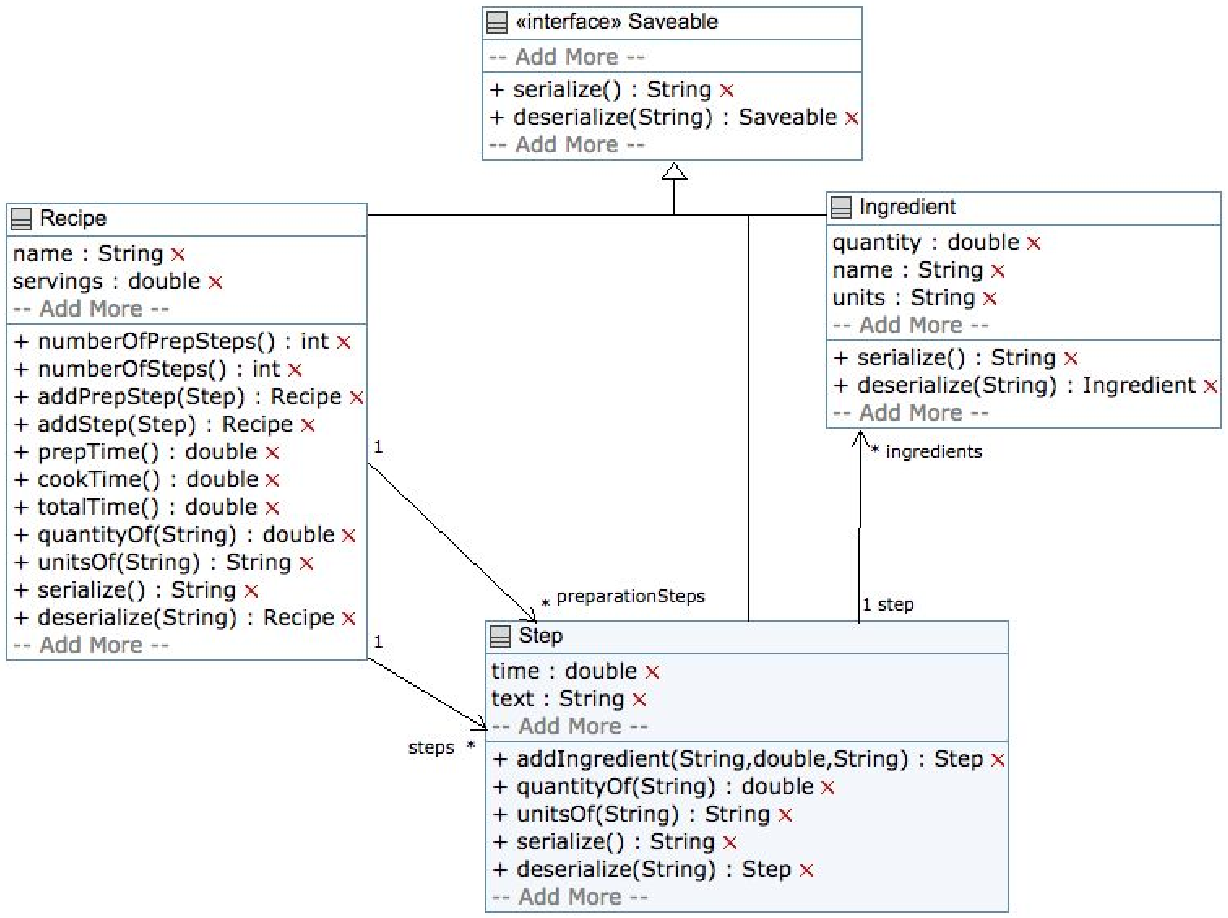

Figure 3: UML class diagrams to show the inner data storage.

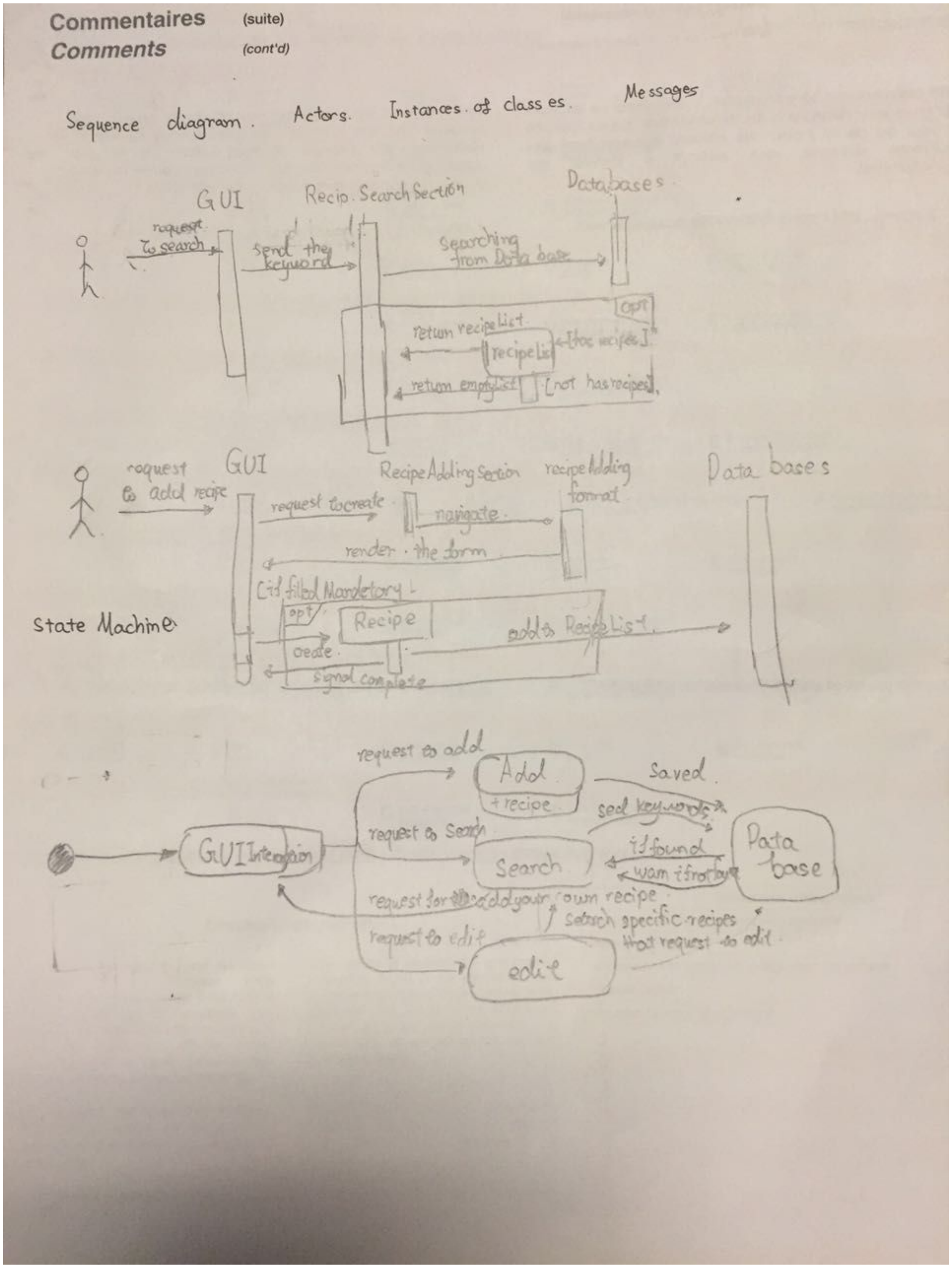

Figure 4: UML state machine.

With these requirements in mind, we decided to use Firebase due to its Database as a Service architecture which made integration very simple and easy to implement in a cross-platform nature. It's cross-platform architecture stems from the fact that Firebase allows communication to occur via a RESTful API. To tackle this, we wrapped the authentication and communication aspects of the Firebase RESTful API into a simple Java class that would handle database manipulation. This class would serialize a given recipe object into its JSON form and push that JSON object to Firebase. Upon retrieval, it would query Firebase for the specific JSON object and deserialize it back into its Java form.

The biggest issue that we faced with Firebase was that its data storage facilities were entirely key-value based. Due to this and due to the fact that Firebase provides a larger service than just data storage, Firebase does not have any ability to search stored data. To tackle this, we implemented our own search engine which would construct queries as Java objects and then match them against a given recipe (more on this algorithm is available under "Recipe Search Engine"). Although this solved the issue of searching, it was a very computationally expensive algorithm in terms of both space and time. This was partially due to the fact that Firebase stores all data under a map that maps from a Firebase ID to the data itself. This meant having to download the entire recipe database to run queries.

In order to improve the space issue, we designed the Firebase database schema to store two separate datasets: (i) a serialized Map object that would map from a recipe name to the recipe object's Firebase identifier and (ii) the recipe collection itself. This allowed us to speed up the search for a recipe because we simply had to do two lookups: the first lookup was inside the Map object that would convert the recipe's name to an ID and the second lookup would convert the ID to a JSON object. This saved a great deal of time when searching for specific recipes but was still very expensive when running more complex queries (more info on this is available in "Recipe Search Engine").

Recipe Search Engine

The search engine was the largest component of this project and was built to allow users to create complex queries to find recipes through any combination of smaller queries. The biggest challenges during the implementation of this engine were: (i) creating a parser that was recursively built to respect all forms of nested queries but in a generic fashion and (ii) wrapping the matching algorithm in an algorithm that would increase the speed of searches.

To accomplish the first of these requirements, we defined a new query syntax that would treat sets of brackets as well as their contents as expressions. We also wrote a parser that would treat expressions as a sum of conjunctions (which are boolean operators such as AND and OR) and other expressions. Primitive expressions would be a simple query with three components: (i) a property name, (ii) a query verb, and (iii) query text (a few examples of queries are present in Figure 3). All verbs were abstracted into separate instances and responsible for the key matching logic. For instance, the 'has' verb tests to see if the query text is present in the property value while the 'is' verb does an exact equals check. This allowed us to create query objects that could be tested against a single recipe object and would return true if the recipe matched the given query.

Query	**Query Explained**
(name has burger)	Display all recipes where the name contains 'burger'.
((category is lunch) or (category is dinner))	Display all recipes tagged as either lunch or dinner.
((name has cake) or ((category is dessert) and (time < 15)))	Display either recipes of cake or any other dessert that takes less than 15 mins to make.

Figure 5: Examples of simple and complex queries.

In order to implement the database-wide searching, we wrote an algorithm that would first convert a query string from the user into a query object and then create *n* threads where *n* is the number of cores on the user's device plus one (this number is generally believed to be an efficient number of parallel jobs by a few tools including make). Upon creation, each thread will be delegated a section of the database to search and will load a single recipe into memory, test the query against it, and add it to the results if there is a match. It will continue to do this until it has searched all the recipes that it was responsible for. Once all threads are done, the final results will be dispatched as the data to the onResults event. This improved the speed of the application drastically and also allowed us to do database-wide searching without worrying about the user's device specifications.