

ECSE 428 - Software Engineering Practice

Winter 2019



FoodMe

(previously Team Lunch)

Iliana Katinakis	260740917
Marine Huynh Huu	260743073
Anthony Laye	260729851
Ammar Rudani	260804420
Rayan Osseiran	260803963
Lara Kollokian	260806317
Steve Lee	260568977
Anne-Julie Côté	260803768
Jeffrey Leung	260747885
Ryan Khan Logan	260378474
Massiva Mahamli	260806869
Ajay Patel	260791038
Qingzhou Yang	260687570
Erion Hysa	260746699
Nicolas Barreyro	260730549

Final Report

Prof. Robert Sabourin

Faculty of Engineering
McGill University

March 31st, 2019

Table of Contents

1. Our Project	3
1.1. Motivation	3
1.2. Software Architecture	3
Backend	3
Android	4
Repositories	5
1.3. Running our App	5
Backend	5
Android	5
1.4. Testing our App	8
Backend	8
Android	9
1.5. Scenarios the Project is Not Suitable For	10
2. Scrum Rituals	11
2.1. Backlog Grooming	11
2.2. Sprint Planning	11
2.3. Story Estimation	11
2.4. Daily Stand Up Meetings	12
2.5. Sprint Demo	12
2.6. Sprint Retrospective	12
2.7. Collaboration Between Team Members	12
3. Scrum Objects	13
3.1. Done Checklist	13
3.2. Product Backlog	13
3.3. Sprint Backlog	13
3.4. Task List	13
3.5. Story Point Burndown Chart	14

4. Agile Manifesto	14
4.1. Individuals & Interactions Over Processes & Tools	14
4.2. Working Software Over Comprehensive Documentation	15
4.3. Customer Collaboration Over Contract Negotiation	15
4.4. Responding to Change Over Following a Plan	15
5. Agile Principles	16
5.1. Prioritizing Customer Satisfaction Through Early and Continuous Delivery	16
5.2. Agile Processes Harness Change for the Customer’s Competitive Advantage	16
5.3. Frequent Delivery	16
5.4. Business People and Developers Must Work Together Daily Throughout the Project	16
5.5. Build Projects Around Motivated Individuals.	17
5.6. Emphasis on Face-to-Face Communication	17
5.7. Working Software is the Primary Measure of Progress	17
5.8. Sustainable Development	18
5.9. Continuous Attention to Technical Excellence and Good Design Enhances Agility	18
5.10. Simplicity—the Art of Maximizing the Amount of Work Done	18
5.11. Self-Organizing Teams for Better Architecture, Requirements and Designs	18
5.12. Regular Reflection and Adjustment	19

1. Our Project

FoodMe ReadMe: <https://github.com/larakollokian/foodme#foodme->

See Also Attached File: 'Log of Work done'

1.1. Motivation

The motivation for this application was to address a simple yet ever-present issue of indecisiveness that comes with picking where one should go to eat. More often than not when you are out with your friends you will often ask them where they would like to go eat and the most likely answer you will receive is "I don't know/I don't mind" this inability to choose a restaurant makes the process a lot more inconvenient for everyone. Foodme offers a solution, Foodme is an app that recommends users where they should go eat based on their preferences. The app is designed to make the entire process more convenient and efficient.

1.2. Software Architecture

Backend

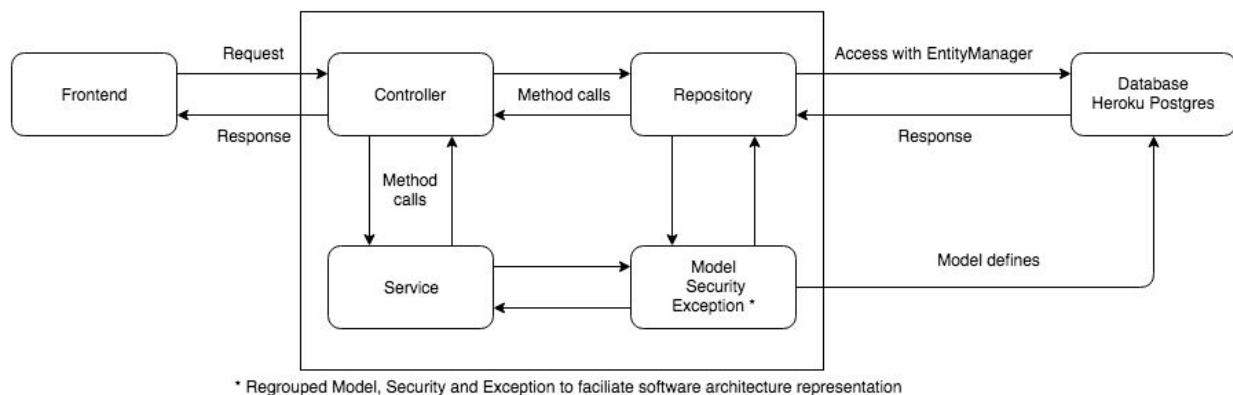


Figure 1. A high-level representation of our software architecture.

The backend main files implementing the business logic can be found under `src/main/java/ca/mcgill/ecse428/foodme`. The backend architecture is divided into 4 main layers: model, repository, service and controller. To start with the model layer, it contains model classes (AppUser, Preference, Restaurant) that are primarily used to define the database (we used Heroku Postgres). An additional class named Response can be found in this layer and is used to define the body of return values of type ResponseEntity of controller methods. Every model class has a repository class that can be found in the repository layer. Repository classes use an entity manager to execute operations on the database known as CRUD (create, read, update, delete). Regarding the service layer, it contains the authenticationService class that serves to login and logout. The last layer is the controller layer. It is composed of 4 controller classes (AppUserController, PreferenceController, Restaurant Controller and SearchController). Controller classes' primary goal is to perform the tasks requested by the frontend. SearchController uses Yelp API to search businesses as specified by the frontend with different attributes(cuisine, price, distance, sortby). The other controller classes make method calls on

their corresponding repository class and on the service class (AppUserController). One thing to note is that all controller methods return a ResponseEntity object to the frontend to facilitate response processing. Aside from the 4 main layers, two additional packages can be observed: security and exception. The exception folder contains the definition of several exceptions that are used throughout the code (NullPointerException, InvalidInputException, AuthenticationException). For consistency purposes, exceptions are all thrown in either the repository layer or in the service layer and all caught in the controller layer. Concerning the security package, it contains the password class that offers random password generation (a feature available for users) and password hashing. To give a brief idea of what password hashing is, it is a one-way function that transforms a password into a fixed-length “key” that can’t be reversed. It is a good way to protect passwords and to assure that they are securely stored in the database. The hashing technique used is salt hashing. Passwords are hashed using the salt generated with java secure random API using PBKDF2 (Password-based Key Derivation Function 2). The figure below shows that the database actually stores the hashed password and not the actual password.

Data Output		Explain	Messages	Notifications		
	username character varying (255)	default_preferenceld integer	email character varying (255)	first_name character varying (255)	last_name character varying (255)	password character varying (255)
1	johndoe	0	john.doe@mcgill.ca	John	Doe	fpy2m78FAiQ=\$59WujxBje/...
2	yeffo	0	yeffo@mcgill.ca	yeffo	yeffo	V0MIMhOmBf4=\$QN91PFr...
3	raylabs	0	raylabs@mcgill.ca	raylabs	raylabs	8PNDbeyJTs0=\$K25T2l0I6...

Figure 2. AppUser relation in the cloud database

Android

The app architecture and package structure for the app are organized in a traditional “package by layer” sense. This is an older method of organizing app architecture that has been superseded by “package by feature” in recent times. However, due to the smaller size of this project, the package by layer architecture was deemed to be adequate. For larger projects, that need to scale, a different architecture would need to be explored. Our own implementation of the package by layer principle is organized into the following packages: activity, adapters, fragment, model and utils. The activity package contains the login activity which handles both login and registration, as well as the main activity, which is the heart of the app and makes use of bottom navigation. It also contains other activities which handle other app features such as editing user information and modifying preferences. The fragment package consists of the three main view elements (the restaurant browser, the restaurant history page, and the settings page) as well as fragments for browsing individual restaurants and navigating to them. Meanwhile, the model and adapter packages go hand in hand to provide functionality and data to the recyclerviews implemented in the app. The classes within the model package essentially define the objects used in the recyclerview and implement their getters and setters. The adapters, on the other hand, handle binding data to the recyclerviews and provide helper methods such as getting the number of elements within the recycler. The utils package consists of a sole universal class that is used to handle touch clicks on the recyclerview.

The Android application also makes use of Jetpack and thus Android X which brings together support libraries and architecture components and arranges them into four components which includes

the architecture, UI, foundation and behaviour. It is worth noting that the Jetpack components are bundled. This means each component can be adopted as we need it enabling us to create a higher quality app with more code reuse and less overall code.

Repositories

To facilitate software development and version control the backend and the frontend (Android) were put in separate repositories on Github to reduced the number of merges.

To start with, in the backend repository, there is the file '*pom.xml*' that defines the Maven project's dependencies and configurations. There is also the '*travis.yml*' file that assures the build of the project using Travis CI on Github. '*Database_tables*' and '*test_coverage*' are two folders that contain images of respective items for visualization purposes. The doc folder contains the generated JavaDoc for the entire codebase. '*README.md*' offers a description of the FoodMe application. The src folder contains the source and the test files. The remaining files are supportive files to the project.

The Android repository consists of the Android source code as well as the '*travis.yml*' file which tells Travis CI how to build the app and run the tests. The core app is stored in the app folder which consists of the source files. This includes a package for testing called '*androidTest*' as well as the main package which consists of the Java source code and the XML resource files, drawables, and other UI related data. The main package also consists of the Android manifest (AndroidManifest.xml) which describes essential app information to the build system, the OS and Google Play. The Android app is built with gradle instead of maven in the backend as it is the standard for Android. The gradle dependencies are listed in the "*build.gradle*" file within the app folder. No action is required to set up these dependencies as they are automatically downloaded.

The repositories can be found under the link below:

Backend: <https://github.com/larakollokian/foodme>

Android: <https://github.com/rayano24/FoodMe-Mobile>

1.3. Running our App

Backend

For the backend, Java JDK 1.8 is required. To download the source code, go to the project repository on GitHub and clone the project to the local machine. All the dependencies needed to run the project are written in '*pom.xml*' so, all one has to do is to import the project as a maven project into an IDE (e.g. IntelliJ, Eclipse) and run it as spring boot app. It may take several seconds to build the spring app and to connect to our cloud database (Heroku Postgres). Once the connection is created and no errors are found, the app should be running properly.

Android

In order to run the Android application, there are two options, however, a physical device is recommended. In the case that you do use a physical device, you will need to enable USB debugging via

developer options on your device. This can vary based on device, but the general instruction is to go to the 'settings menu → about phone → build number'. Once you are there, tap build number 7 times in order to enable developer options. Now, go back to the settings menu and visit the System page and tap the advanced button. From there you will see 'Developer Options', and should find the 'USB Debugging' option and enable it.

Note that the target API for the application is Android 9 (Pie), however, the app will run on all devices Android 5 and up.

You may simply sideload the APK provided into your physical Android device. Once it is on your phone, you simply open and install the APK. Note that if you are running Android Nougat (7.1.2) or lower, this may require turning off permissions within your phone in order to download applications from sources outside the Google Play Store. If you are running on Android Oreo (8.0) or higher, an individual permission pop up will show when you attempt to install the app.

For this option, you will need to use Android Studio (preferably the latest version). Clone the repository that is linked at the end of section 1.2 of this report. Open the project via Android Studio. If you are not testing on a physical device, you will need to create a virtual device which will include a large download of an Android image. The FoodMe app requires that the emulator has access to the Google Play Store in order to make use of the Google Maps API, and may not function as intended on all emulators. We recommend the Pixel 2 emulator but have also tested and confirmed the Pixel and Nexus 5X emulators. Results may vary with other emulators.

Name ▼	Play Store	Size	Resolution	Density
Pixel XL		5.5"	1440x2...	560dpi
Pixel 2 XL		5.99"	1440x2...	560dpi
Pixel 2	▶	5.0"	1080x1...	420dpi
Pixel	▶	5.0"	1080x1...	420dpi
Nexus S		4.0"	480x800	hdpi
Nexus One		3.7"	480x800	hdpi
Nexus 6P		5.7"	1440x2...	560dpi
Nexus 6		5.96"	1440x2...	560dpi
Nexus 5X	▶	5.2"	1080x1...	420dpi

Figure 3. An illustration of AVD options. Recommended options are circled in black.

Once you have set-up an Android virtual device, simply press the run button which will display a pop-up with a list of your emulator devices. Select the one you just created and the app will launch.

Note - When testing via an emulator, you must configure your location in the emulator settings. To do this, press the overflow button on the right side of the emulator (illustrated in Figure 4). Once there, enter a longitude and latitude on the locations tab. We recommend setting your location to Montreal. The coordinates for Montreal are latitude: 45.5017 and longitude: -73.5673. **Not setting valid coordinates may result in loss of functionality.** Also, while enabling location services is not required for the app to run, it is recommended for a better experience.



Figure 4. An illustration of the overflow button in the emulator.

Should you run into any trouble with the emulator, we have also provided a list of the SDK tools we have installed on our own version of Android Studio. This is illustrated in figure 5 and can be found by opening the tools menu and selecting 'SDK Manager'. If you cannot get the map to display on the emulator, it is recommended that you test on a physical device.

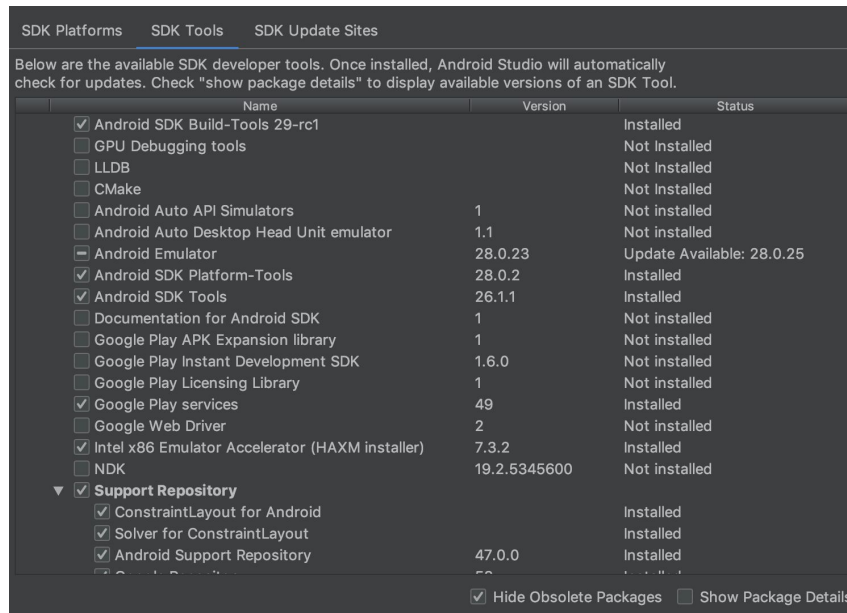


Figure 5. Illustration of Android Studio SDK tools.

1.4. Testing our App


Element	Class, %	Method, %	Line, %
controller	100% (7/7)	90% (50/55)	83% (268/322)
exception	100% (3/3)	100% (3/3)	100% (6/6)
model	100% (4/4)	100% (58/58)	100% (110/110)
repository	100% (3/3)	92% (35/38)	82% (244/297)
security	100% (1/1)	100% (5/5)	95% (22/23)
service	100% (1/1)	100% (6/6)	100% (34/34)
FoodmeApplication	100% (1/1)	0% (0/1)	25% (2/8)

Figure 6: Test coverage generated using the IntelliJ IDE

Backend

Backend is tested through 5 test classes namely ModelTests (unit testing), FoodMeApplicationTests (unit testing), AuthenticationServiceTests (unit testing), ControllerTests (integration testing), SearchControllerTests (integration testing). These test classes can be found in the repository under `src/test/java/ca/mcgill/ecse428/foodme`. As the name suggests, the ModelTests class tests the classes of the model layer (AppUser, Preference, Restaurant, Response). FoodMeApplicationTests class tests all methods that are found in the repository layer while AuthenticationServiceTests verifies the validity of the methods in the service layer. Both are using the Mockito framework to test the methods. The ControllerTests class tests all methods found in the Controller classes related to the model (appUserController, preferenceController, restaurantController). More importantly, the order at which tests are run is actually very important as some depend on others to run successfully. In order to assure that dependencies are respected, alphabetic prefixes were added in front of test methods' name and tests were run in ascending order. Finally, the SearchControllerTests class assures that the methods in SearchController produce consistent output. In brief, these 5 test

classes all run with the SpringRunner class and cover 85% of line codes. The CoverageTest class is a test suite of all test classes that was used to facilitate the output of test coverage.

To test the 5 test classes, one needs to run them as JUnit test and to see the test coverage, simply run them with coverage code measurement. On IntelliJ IDE, this can be achieved by pressing on this button .

Android

The frontend (Android) is tested through a suite of acceptance tests. Everything concerning these tests is found under src/androidTest in the Android repository. The tests are organized into three main components: feature files, step definitions and action steps.

First, the feature files are written using the Gherkin language and describe exactly one high priority user story. Every feature contains one or more scenarios, and scenarios contain several tests. These files are found under src/androidTest/assets/features.

Second, the Cucumber tool allows to match every Gherkin step (given, when, then) in a feature file to its definition in the CucumberSteps class found under app/src/androidTest/java/com/mcgill/ecse428/foodme/test. From this class, calls are made to step definitions for every particular feature found under the StepDefs package.

Third, the action steps are navigation actions that simulate user interaction with the UI. These interactions are made using the Robotium library. This library allows us to click, send data, receive data from/to UI elements such as buttons, editable text fields, etc. Given that the Gherkin features must not describe any sort of navigation, these navigation actions are defined separately. They are found in the CucumberActionSteps class in the same package as CucumberSteps.

To run tests on a physical device only:

1. Having a physical Android device plugged in and having allowed USB debugging, open a command line, navigate to the root of the Android project and type the command: `“./gradlew connectedAndroidTest”` or `“./gradlew cAT”`. This command builds and runs the tests locally using the gradle wrapper.

To run tests on a physical device or emulator:

2. Navigate to the tests folder using Android Studio and run every test under the features package as shown in Figure 7.

The tests are also built and run continuously using Travis CI. However, given the large amount of time it takes to set up the emulator on the Travis side, the master branch of the project does not run these tests. The tests are continuously built and run on the “gherkin” branch in the Android repository.

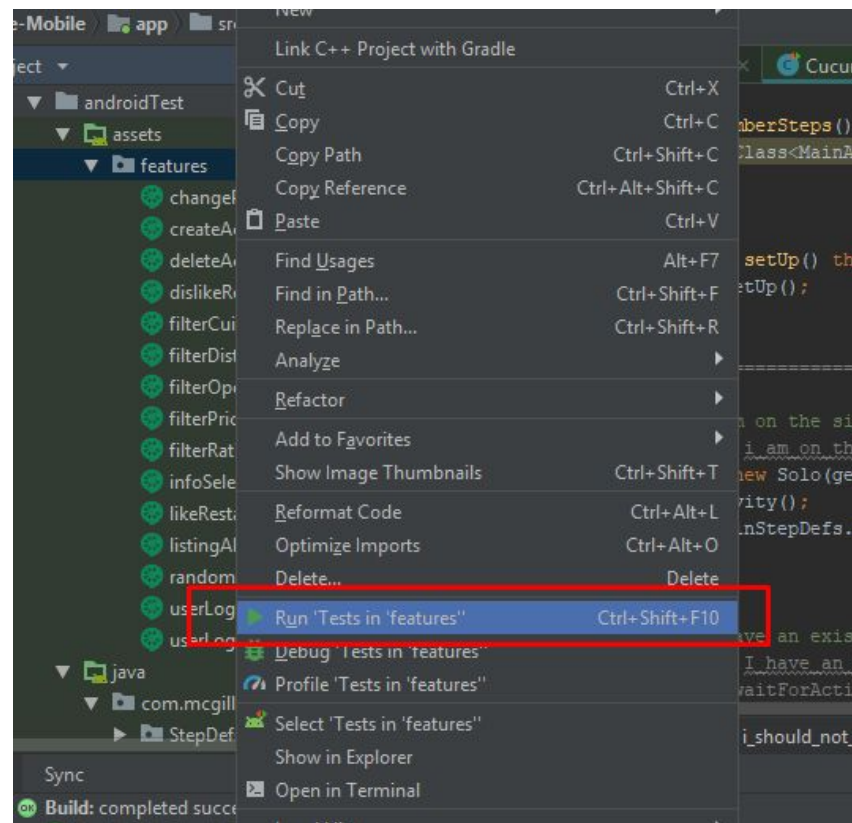


Figure 7. How to run tests using Android Studio

1.5. Scenarios the Project is Not Suitable For

There was an attempt to implement a medium priority user story aiming to display the itinerary to the chosen restaurant from the current location of the user using the Google maps API. This feature was not perfected and wasn't pushed to the master repository. There was a lot of trial and error in order to implement this feature, the developer started by trying to implement the feature in the backend and testing it before making an http call in the Android application. However, after doing some research, this feature could've been done in Android right from the start and the implementation did not use the backend anymore. While a certain amount of code was written to implement this feature, the developer had difficulty running the map that displayed while using the app, it was realized later that an updated version of the Android Studio emulator needed to be installed. All in all, the itinerary feature was not functioning on time and was not integrated into the master repository as it was decided that it would be more efficient to focus efforts on the high priority stories. With a time extension, this feature could certainly be integrated.

2. Scrum Rituals

2.1. Backlog Grooming

During both sprint 1 and 2, every time the team would meet collectively to do backlog grooming. As tasks were completed, they were closed on Github. However, often times many user stories were deemed irrelevant and removed from the board. Additionally, the team would collectively choose which user stories were high priority depending on the essential features of our application. All these decisions were updated in the backlog. During the first sprint, backlog grooming was done quite a bit as the project was still in its early stages. This required many changes to user story priorities as well as the addition of new stories that were not considered in the tasklist originally. Over time, fewer changes were made. Throughout the second sprint, backlog grooming did not have to be done nearly as much as the direction of the project was decided and most user stories were given the appropriate priorities.

2.2. Sprint Planning

Sprint planning would occur during the first team meetings for the sprints. The meetings were usually held in a large enough room to fit the team. The goal of sprint planning was always to create a tasklist of all the features to be implemented and tests to be done during the sprint. The first sprint planning session was convoluted and many of us came out of the meeting unclear on the meaning of several aspects of the tasklist. This is due to the fact that sprint planning was relatively new to all group members. The solution was to meet again the following week and clear up the confusion as well as create a concrete tasklist that everyone was clear on. In sprint 2, the sprint planning session went much better as every team member came prepared with tasks and knew what to expect. The key difference between the two sessions was the preparation and planning of the meeting. In the second meeting, the time was split into the scoping and the planning. In the scoping part of the meeting, we discussed the goal of completing the project and discussed how many aspects of the project that we believed we could manage before the deadline. In the planning part, tasks for the sprint were added to the backlog according to the previous discussion. The assignments were distributed according to the team members familiarity with the technology to be used to complete the given task. Proper sprint planning allowed the second sprint to be more successful than the first with respect to tasks being completed.

2.3. Story Estimation

Instead of creating tasks first and then trying to gauge their estimated completed time, tasks were allocated so that they would take an approximate 1.5 hours to complete. The main factors considered in this estimation were: the technical difficulty of the feature (complexity), the team member's experience with similar tasks (risk), and whether or not progress could be blocked waiting on other dependent features (interdependencies). Tasks for the first sprint often took close to 3 or 4 hours to complete. Lack of previous experience and unexpected problems may have contributed greatly to this underestimation. The process evolved after sprint 1, as members were able to get a better idea of how

long certain tasks would actually take them. The time it took to complete different jobs was very close to their estimated time for the second sprint, which shows much improvement.

2.4. Daily Stand Up Meetings

The way our team organized the daily standup meetings was through slack where the ScrumMaster would gather updates from everyone. From then on the progress could be kept track of and at the beginning of the day objectives on what has to be accomplished for the project could be identified. Through this information, all of the teammates knew where the project was standing and know which parts they should focus on. Sometimes the meetings were held in person by the team members, but because of the time conflicts, not everyone could participate, however they could gather the information by being updated by those who attended the meetings.

2.5. Sprint Demo

At the end of each sprint, the scrum master for the respective sprint had a meeting with the TA to discuss the progress over the course of the sprint. During sprint 1, as we explained before, there was some confusion between the team members on the direction we wanted to take with the app in the first week. Therefore, adjustments had to be made and a revised tasklist had to be created. Since it was our first try, we had some trouble with estimating the tasks properly. Therefore, at the sprint meeting, we discussed which tasks were completed and what was leftover, and the TA gave some suggestions on how to organize ourselves better for the next sprint. The second sprint went much more smoothly, and we completed the tasks on time and in a more organized fashion. When it came the time to demo, we had a functioning final product. During the meeting, we demonstrated the main functionalities of our app related to the high priority stories as well as their associated tests.

2.6. Sprint Retrospective

A Sprint Retrospective was done in tandem with the planning meeting for sprint 2. During the meeting, we discussed our problems with estimation and updating the way we handled our sprint backlog. It was decided that we needed to ensure our tasks were divided into smaller chunks; chunks that would be achievable in less than ninety minutes. We also noted that, as we now had more experience, our estimates would be more accurate. With regards to our sprint backlog, we decided to begin maintaining a known issue board on GitHub with all the tasks that needed to be completed, so that our Scrum Master could better facilitate the work that needed to be done.

2.7. Collaboration Between Team Members

The way we organized our team was that we made 2 small subgroups on our slack channel, one specific to the frontend, and one for the backend. We used these channels to communicate issues/concerns related to the specific development end. Slack was a very useful tool to help us collaborate since it was a fast and effective way to ask questions and get feedback from team members on issues that we were facing without necessarily having to meet constantly. Another form of collaboration came from team members with similar tasks which involved implementing similar

methods/tests. If this was the case, those members would meet up to work on those tasks together, putting a combined effort into tasks with similar end goals so that they may be achieved.

3. Scrum Objects

3.1. Done Checklist

The original done checklist was included in the first sprint preparation document. Throughout the sprint, whenever a task was completed by a team member, they went over the done checklist to ensure that the standards for the project were met. Since the sprint preparation submission, the checklist had to slightly evolve to account for changes made in the work process. When a task was completed the member now had to also close the issue on the GitHub issue tracker, as well as update the task list. If the task was done in the Android app repository, they also had to ensure that all of the gherkin tests built properly.

3.2. Product Backlog

Our Product Backlog was produced during the project planning phase and remained mostly unchanged throughout the two sprints. From the beginning, we had identified 25 high-priority user stories. The Product Backlog is included at the beginning of the attached document, 'Log of Work Done'. On the backend, 23 high-priority stories were implemented, one was removed and the remaining story only applied to Android development. For Android development, 20 high-priority stories were implemented, one was removed, and five would remain for the next sprint.

3.3. Sprint Backlog

As mentioned in 2.3, over the course of Sprint 1, when breaking down a story into its constituent task the time estimates for completion were often quite inaccurate. The central difference in Sprint 2 came not only from greater experience in estimation but also be more experienced in breaking down stories into more, quick to develop parts.

3.4. Task List

A master task list was made at the beginning of every sprint. This means that, at the beginning of every sprint, all tasks that need to be completed would be written down, along with their start date and an estimated time of completion. Then, individual members would assign themselves the tasks that they were confident they would be able to complete. Throughout the sprint, the completed tasks would be updated to indicate the date of their completion, as well as the actual time that was taken to complete the task in question. Doing so allowed us to visually track our progress by counting the number of completed tasks and the number of hours that each team member had put into the project. Our team also coupled the task list with GitHub's issue tracker and project board, whereby we can also open issues as bugs were discovered, close them as they were fixed, and have an online project board tracking the tasks to do, complete, and in progress.

3.5. Story Point Burndown Chart

For our story point burndown chart, all tasks were assigned a half point for every hour of work estimated. All tasks that were related to a high priority story were counted twice. For instance, a task estimated as being two hours of work would be worth 2 points if it was related to a high priority story, and only one otherwise. This remained unchanged throughout the project. 157 points were completed during Sprint 1 and 149 during Sprint 2. See Figure 8, below.

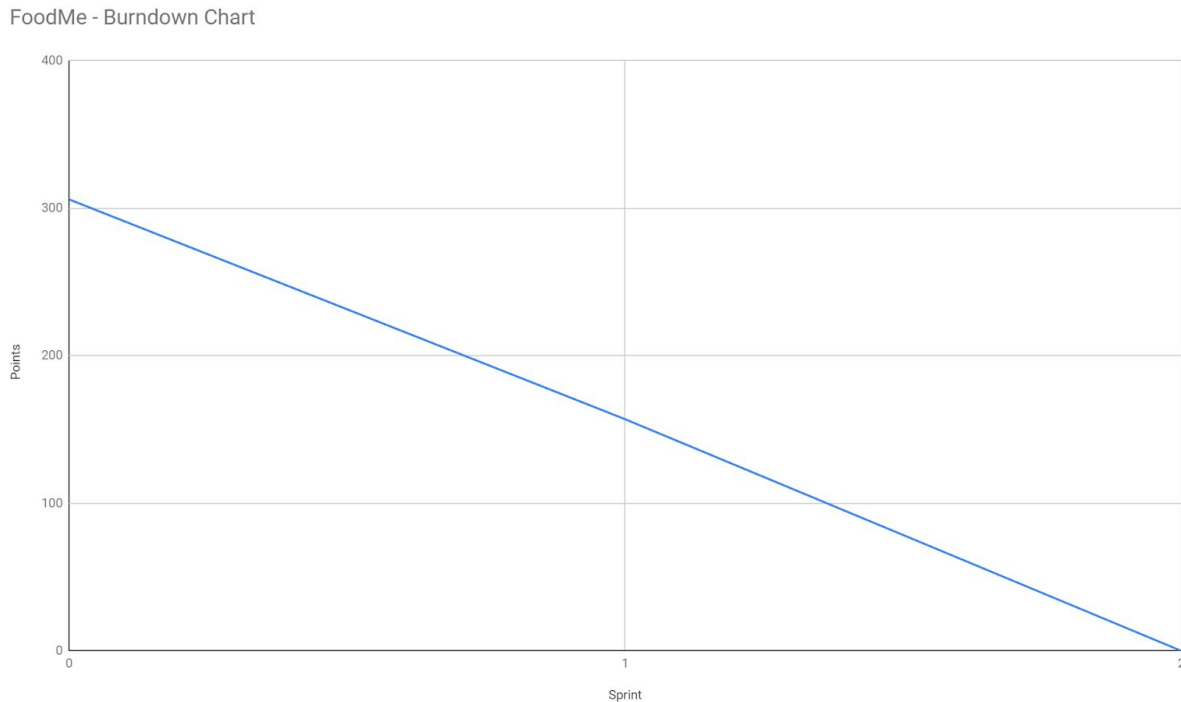


Figure 8. Project Burndown Chart

4. Agile Manifesto

4.1. Individuals & Interactions Over Processes & Tools

This Agile value focuses on the idea that, while predetermined processes of work and available tools are important in contributing to a well-organized project, they are not the most efficient since everyone has their own strengths and weaknesses that they bring to a project—not every team will yield the same output by using the same processes and tools. Therefore, it is important to be mindful of your team composition and the resources available to the project. The way this was implemented in our team was through the design decisions we used while doing this project. For example, when it came to setting up our database, we decided to use the tool most familiar with the majority of our team, Heroku. Similarly, we decided to implement an Android application as it was the development environment most people were familiar working in. When it came to situations where certain members were unfamiliar

with processes of developing with the tools we were using, we placed an emphasis on collaboration; where more experienced members helped the less experienced so that they became proficient with the new work environment.

4.2. Working Software Over Comprehensive Documentation

Creating the software was the priority of the team, instead of focusing most of the team's resources on the documentation aspect. However, this does not mean the documentation was nonexistent over the course of the project. Every team member added Javadoc comments to their code and what the functions would do as well as provide a skeleton of documentation for their parts. This ensured that the code could be prioritized. The documentation was partially generated alongside the code and the organization could be understood by everyone on the team. Then the documentation could be completed in thorough detail for all the parts. Another reason why documentation was not focused on in the beginning is that there could be some methods written that would have been documented, but these methods ended up being deprecated quickly, wasting project resources. Since the unit tests validate the behaviour of individual routines within the system and integration tests verify that all the parts of the system work together as they should, they provide living documentation of the system.

4.3. Customer Collaboration Over Contract Negotiation

Throughout the project, we were simultaneously acting as our own customer and developer. Our contract was 3 person-hours/week, and, because we were acting in both roles, we sought to both stay on budget, on time, and adapt the project to best suit the business needs.

4.4. Responding to Change Over Following a Plan

The fourth core value of Agile project development address a team's ability to react to change. A change may include a change in ideas, or stories, though it also involves the ability to respond to unexpected changes, which would not be possible by following a rigid plan. Working on a predefined task list meant that all high priority stories were established prior to the beginning of the sprint. In the first sprint, we often found ourselves coming across difficulties with tools, such as a limited number of connections allowed at any time to the Heroku server. This often lead to a delay in the completion of some tasks, as the tasks were dependent on the proper functioning of these tools. Once these issues had been identified, it was possible to incorporate them into our development process. As such, running into issues did not hinder the development of our application, but rather helped us gain a better understanding of the tools we were working with. Consequently, when preparing for the second sprint, it was easier to establish a better time estimate for tasks involving these tools, since we had already come across issues and our team has learned to adapt to sudden changes. Following a rigid plan, such as the waterfall model, the errors or changes we would have run into at a later stage in the project would have caused us to backtrack and rework all affected tasks.

5. Agile Principles

Within the Agile Manifesto, twelve principles have been described for the Agile approach. The principles keep in mind that working software and client satisfaction are the goals to a successful software project. The twelve principles have been stated below, along with a description of how each principle exists in our project.

5.1. Prioritizing Customer Satisfaction Through Early and Continuous Delivery

At the beginning of each sprint, our team has described a list of tasks related to each user stories. The high priority stories were specified first during the first sprint to make sure that they were done as soon as possible to be able to make adjustments, if necessary, during the second sprint. These tasks were mandatory to assure the satisfaction of the client. All high priority tasks were implemented and tested to assure the reliability of the features. In order to maintain consistency, a nightly build was triggered through Travis-CI to make sure that the build was successful. In the case where the build failed, it was easy to view where the issue was, and features and tests were modified to comply to it.

5.2. Agile Processes Harness Change for the Customer's Competitive Advantage

Being able to respond to changing requirements was the core strength of our development team. As has been mentioned, during Sprint 1 we had to respond to several changes that came up as we were about to transition to Sprint 2 in order to deliver the best product possible: Addressing issues with a limited number of connections; readjusting many stories through grooming; etc. Addressing and adapting to these changes allowed the team to deliver better software in the shortest amount of time possible.

5.3. Frequent Delivery

As Git handles version control, team members were encouraged to commit frequently, as the master branch could always be rolled back. This ensured that a working product was always available and frequently being updated. In the beginning, fewer commits were made to the master as people wanted to keep the code separate to ensure the full functioning of their feature before merging. This mentality evolved over time since it became more difficult to merge automatically when there were more changes to the code. Once the code was pushed, a build was automatically triggered. Using Travis-CI for continuous integration, as soon as someone pushed code to a branch, a build started. It also assured maintenance of the code; the failure or success of a build helped us with the surveillance.

5.4. Business People and Developers Must Work Together Daily Throughout the Project

In the context of this project, our team was simultaneously the customer and development team, so we made sure to put ourselves in the shoes of business people as well as developers. When finding the idea for our app and establishing the requirements for it, we considered the business side of things because we wanted to develop a realistic app that could be profitable. We also thought like developers to make sure that what we were planning on doing was doable in terms of our programming

skills as a team. Then, during sprints, even though most of our time was devoted to developing the app, we kept in mind that the deliverable should be an app that we could see being used on the market. We also received feedback from the TA throughout the project, which was similar to the product owner telling us if we were on the right track or not. Every change we encountered had to respect considerations issued by both the developers and those in charge of the business.

5.5. Build Projects Around Motivated Individuals.

The first step to respecting this principle was to find a project idea that our team members found interesting and motivating, and that they could see themselves realizing. We also made sure to use tools that most people were comfortable with, in order to make the development environment was suitable for our team's skills. From the beginning of the development, our team established that we should feel comfortable asking others for help if we encountered any difficulty so that everyone felt supported. Another initiative pursued by the Scrum Masters to ensure that support and trust were present in our project was to remind team members of deadlines every once in a while in a motivating tone. This helped make sure that the team kept working without feeling too pressured and stressed. The task list document also included a small board that tracked each task completion. This was a visual way of showing members how the project advanced and it motivated people to get tasks done. Additionally, every member trusted others to get their tasks done, which helped everyone focus on their own tasks and made sprints efficient. Finally, Scrum Masters checked in with developers individually once in a while to see how their tasks were progressing. This was done, not because they didn't trust them, but to make sure that if they needed help they felt comfortable asking and to point them to the team member best suited to help them.

5.6. Emphasis on Face-to-Face Communication

Although Slack and other tools are great tools for communication, face-to-face conversation is extremely important for conveying information between teammates. Weekly meetings at the beginning of the project were the best way to ensure that everyone was on the same page in terms of requirements and expectations. This was very effective for the group as the project evolved; less face-to-face meetings were required since everyone was in agreement with the purpose of the application. Members were then able easily to work on their own tasks, knowing that any uncertainties could be resolved quickly in Slack. In-person meetups would occasionally still occur if big changes to the database or feature logic had to be changed.

5.7. Working Software is the Primary Measure of Progress

According to the Agile principle having simple software that works is more important than having more intricate features that are not fully developed. When working on our project the primary objective was to make sure that our code worked properly. There was an emphasis on having code that worked well, even if it had simple functionality. This was done in favour of larger unfinished code, that didn't work as well but implemented more features. The way we approached this during our sprints was by having a focus on making sure a feature worked well in our remote branches before pushing anything onto the master branch. Because these features were small, they encouraged frequent commits and

once we had the code written, it would be tested multiple times on a REST client to make sure it is functioning as expected. Once pushed to the master, further testing is done through JUnit testing and integration testing. Finally, once the backend functionality is connected to the Android frontend a whole new set of tests are conducted using the emulator to make sure the backend is well integrated with the frontend.

5.8. Sustainable Development

Due to the aforementioned problems with estimation during sprint one, the pace of development was unfortunately unsustainable for the developers at this point. Over the course of Sprint 1, as the team improved at estimation, and during the re-evaluation and planning for Sprint 2, tasks were made to be more sustainable. From the experience gained during Sprint 1, the team was better able to divide tasks into approximately ninety-minute tasks: What would have been one task in Sprint 1, was sometimes divided into two or three tasks. As such, by the end of Sprint 2, the team was working at a sustainable rate for four-week sprints and, with continued experience, could expect to reduce the sprint duration sustainably within a few more.

5.9. Continuous Attention to Technical Excellence and Good Design Enhances Agility

Throughout development, particular attention was paid to testing. No builds that were unable to pass integration tests were allowed into the master branch. Furthermore, each developer was responsible for the code they introduced into the master. This made it much easier for the team to adapt. For example, when changes to stories and features needed to be made at the end of spring one. The existing code was of high quality and required little adjustment to be reconfigured to perform its new functionality.

5.10. Simplicity—the Art of Maximizing the Amount of Work Done

Not going over what is required by the client is an Agile principle that implies that working too much is not needed when the requirements are already met. In order to assure the functionality of the features, unit tests, integration tests and system tests have been implemented as described in the tasklist. These essential tests then evolved when we started looking at the Jacoco report. Through the use of Jacoco, a java code coverage library was generated, and it was easy to see where the code was supported and where the code was not covered by tests. From there, we were able to increase our tests' code coverage where needed to make sure that the features were well supported. The Jacoco report was then sent to SonarCloud, a continuous code quality assurance tool—it is the cloud version of SonarQube. From there, the code coverage, the code smells, and the bugs can be seen. Members of our team can rapidly see the downfalls and correct them before they become too important or irreversible.

5.11. Self-Organizing Teams for Better Architecture, Requirements and Designs

By being self-organizing teams are not bound to outside decisions that may artificially constrain, or hinder development. In our case, we were able to choose the tools and practices that each team was familiar with. For instance, those working on Android were able to choose to use package by layer. This

was not only motivated, by the small scale of the app, but also because the developers were more familiar with the paradigm. This facilitated a smoother development, and thus a faster and higher quality product. Furthermore, because features and stories were selected for implementation by the developers, we were more easily able to select the features and stories that could realistically be delivered during a single sprint.

5.12. Regular Reflection and Adjustment

This Agile principle speaks to the fact that the longer a team works together, the more effective it becomes. That is, team members learn each other's methodologies and find an optimal balance over time. This was also the case over the course of our project. Throughout the semester, we had two official sprints, each lasting one month (four weeks). At the beginning of each sprint, we would create a task list and each team member would be assigned a certain number of tasks whose estimated time would fill the budget. In our case, this budget was around 3 hours of work per member per week. At the end of each week, we would submit an updated task list demonstrating the progress throughout the sprint. As we could not change the number of tasks nor their estimated time after each sprint has started, we learned quickly about the importance of estimating tasks accurately, as well as each team member's skills, e.g. confidence with databases, UI development, backend development, and testing. Consequently, at the start of the second sprint, we had greater ease in creating tasks that had a more accurate time estimate and were more systematically assigned according to skills. Often, Agile development teams spend years working together, and their effectiveness as a team cannot be compared to ours in a two-sprint project. It is important to note, however, that we dramatically improved our efficiency just in a single sprint and shows that, if we were to continue working together, we would also continue to increase our effectiveness.