CorkCollector
Final Report

By

Russell Stirling, Daniel Lorencez and Bailey Hanna

Faculty Advisor: Luiz Fernando Capretz, Ph.D., P.Eng

Software Engineering Design Project (SE4450)
Department of Electrical and Computer Engineering
The University of Western Ontario
London, Ontario, Canada

Abstract

The Niagara region of southern Ontario is home to some of the best wineries in Canada, which boast world-renowned wines. They often hold tastings and vineyard tours, making Niagara a fantastic tourist spot for wine-lovers. Sadly, these opportunities can be frequently overlooked because touring information is not well-presented to those visiting the area.

Today, most people get their information online or through mobile applications. Niagara wineries have fallen behind this current trend, and are suffering for it. Many have minimal or no online presence whatsoever, and are unable to provide current information on their hours of operation, bottles available for tasting, and other features that can allure potential customers. In order to assist all wineries in the region, and modernize their outreach methods, we have developed the application CorkCollector.

Our Android application makes it easy for visitors to learn more about Niagara wineries, get directions, plan their visits and even see which wines are available for tasting at each location. Wineries can choose the information they share, and advertise it directly to their customers. This will expedite the process of getting these wineries up to speed in the digital world, and allow them to take ownership of their online presence to fully promote themselves.

Acknowledgements

We would like to thank our advisor, Luiz Capretz, above all for his guidance and assistance in the creation of this project. His expertise and advice have allowed us to create our vision and allowed us to share our passion with everyone who visits the Niagara region. Secondly, we would like to thank Thane MacKenzie for all of his help in getting up to date and accurate information on the current status of the Niagara wineries which are currently listed on the application. Without his help and guidance we would not have been able to get this application from the prototype stage to a fully viable product.

Table of Contents

1. Introduction

1.1 Purpose

This report serves to define the scope, design and requirements of the Android application *CorkCollector*. It will be used to plan out the details and specifications of the application, and guide the team of programmers who will develop and test the software over the course of the next 6 months. As this application has not yet been programmed, the report will cover all alpha and beta revisions up to the application's primary release with Version 1.0.

CorkCollector will make heavy use of an online database system, the scope and planned implementation of which will be detailed in this document as well. However, *CorkCollector* is considered a separate entity from the database and the two will identifiably distinguished at all points throughout the document.

1.2 Scope

CorkCollector is a multi-functional mobile app for wine connoisseurs and amateurs alike. It will serve as a platform that provides information to the user on local wines, wineries, and wine tastings. Additionally, it will serve as a tool for users to take notes on wines they have tasted, leave public ratings and reviews, and manage a virtual inventory of wines they have purchased.

The primary focus of this application is encouraging tourism and economic of wineries in the Niagara region of Ontario. *CorkCollector* will act as a guide for tourists by promoting local wineries, tours and tasting menus, as well as a promotion service for local users by hosting online discussions about wines and sending updates when a new local wine is available.

As mentioned previously, it will only be developed for Android phones at present.

1.3 Intended Audience and Document Overview

This document is intended to be read by solely our project advisor. As *CorkCollector* is an independent software application with a user-centric design strategy, there is no single client that its development caters to. Our advisor would benefit most by reading the document in order. The background information contained in the earlier sections is vital to understanding the product requirements and development strategy.

1.4 Definitions, Acronyms and Abbreviations

Cellar	Virtual wine inventory where users can track which wines they have purchased, purchase dates and any tasting notes.
VQA	Abbreviation of “Vintners Quality Alliance”, a certified wine appellation organization. Its products are certified to be grown entirely from Ontario grapes.
AWS	Amazon Web Services. A hosting platform for web servcies.
NoSQL	“Not just SQL”. A database implementation that models data differently than the standard SQL tabular structure.

1.5 Document Conventions

Any in-application terminology, as well as *CorkCollector*, the application’s title itself, will be italicized. All references will be hyperlinked, and appended with a number in superscript to denote which reference in 1.6 they are linked to.

1.6 References and Acknowledgements

[1] https://winecountryontario.ca/media-centre/industry-statistics
[2] https://www.niagararegion.ca/about-niagara/statistics/population-and-maps.aspx
[3] http://www.dofactory.com/reference/csharp-coding-standards

1.7 Walkthrough Feedback

Our walkthrough presentation was generally well-received, however there are two questions we were asked that require some further explanation.

“Have you validated your idea? I’m not convinced this is an issue that needs solving”
Statistical data shows**[1]** that VQA wineries are a significant contributor the Ontario Economy, providing a combined $120M in sales as of 2014. After directly speaking with winery owners, we confirmed the weak online presence of independent wineries - and the dire need for a central, accessible hub of information for the smartphone age.

“Can I search or sort wines by my comments?”
The note-taking and *Cellar* systems will be developed with user accessibility in mind. All wines will have user-defined tags - one-word descriptors that can be attached to each wine - that will allow them to be sorted and searched for.

CorkCollector

Software Requirements Specification

Date: April 11th, 2018

Russell Stirling	250757946
Daniel Lorencez	250741921
Bailey Hanna	250718425

Revision History

Date	Version	Description	Author
2017-11-14	1.0	Initial document creation with headings.	Bailey Hanna
2017-11-17	1.1	First draft of individual sections.	Bailey Hanna Daniel Lorencez Russell Stirling
2017-11-20	1.2	Mockup and diagram insertion.	Bailey Hanna Russell Stirling
2017-11-23	1.3	Sections revised by different group member than initial author.	Bailey Hanna Daniel Lorencez Russell Stirling
2017-11-27	1.4	Final draft of all sections completed.	Bailey Hanna Daniel Lorencez Russell Stirling
2017-11-29	1.5	Document formatting.	Daniel Lorencez
2018-03-15	2.0	Sections revised based on feedback from Midterm Submission	Bailey Hanna Daniel Lorencez Russell Stirling
2018-03-17	2.1	Revised sections reviewed and edited by team members other than revision author.	Bailey Hanna Daniel Lorencez Russell Stirling
2018-04-06	2.2	Document formatting	Russell Stirling

1. Introduction

1.1 Purpose

This document will provide an overview of *CorkCollector*'s envisioned software system as a whole, as well as an outline of functional and nonfunctional requirements. Application design criteria and limitations will be portrayed and supported with appropriate architectural diagrams and visual interface mockups. Overall, this document will serve as a beacon to the team during development, guiding our planning, understanding and execution of the application.

1.2 Overview

The SRS will be comprised of four sections: The introduction, the overall product description, the requirement specifications and the supporting information.

The second section will describe general factors that impact *CorkCollector* and its planned requirements. It will also delve into users, their defining factors and how it will impact their use of the application. Effectively, this serves as background information for the requirements.

The third section will provide a thorough description and analysis of the product requirements, both functional and nonfunctional. Each sub-heading will address a specific aspect of the product that ensures its quality, and will be the standard we base our testing plan around. Finally, the section will include a description of *CorkCollector'*s interfaces.

The last section will contain all visual aids and data that has been referenced in any of its predecessors including mockups, as well as system and architecture diagrams.

2. Overall Description

This section will provide a general overview of the limitations and capabilities of the *CorkCollector* system. It will give a high level view of interactions with the systems it uses. The different user types will also be reviewed, along with system constraints and dependencies.

2.1 Product Perspective

CorkCollector will consist of three primary components. The first is an Android application front end which will be used to render different views and UI components for the user to interact with. The app will be implemented using Java and the android SDK targeted at Android devices running Android 4.0 or higher. The app will make use of the Google Maps API to render an interactive map interface that the user can use to view the location of different wineries relative to their current position. This API will also allow us to provide the user directions to the wineries they wish to visit and help us to check when they are actually visiting a winery. Other display pages will be written by us and will display wine and winery information (along with reviews for each), wine menus available through winery or user grouping, and user profile information and statistics. The application will be installed on user devices through the Google Play Store. The app will get the information for these pages through a REST web API.

The REST web API will be a ASP.Net Web API implemented using C#. The API will authenticate users using OAuth 2.0 protocol with token authentication. It will be accessed by our app through http requests (as with any REST API) and will return data in the form of JSON objects or response codes. The app will also be able to make put and post requests for modifying or adding data such as ratings, reviews, check ins, etc. The API will also contain some logic designed to recommend new wines and wineries to users based on their previous selections and ratings. Unit tests will also be written for all our API's functionality using XUnit. This will allow us to quickly check that new feature implementation does not affect any old code features. The actual data objects our API will be accessing will be stored in a NoSQL document database. User account data will be stored in a SQL database.

The database is being implemented using RavenDB. RavenDB integrates extremely well with C# and provides a free (community edition) server to host the NoSQL document database which also provides a user friendly GUI for querying, viewing and modifying data objects within the database. RavenDB also allows the creation of custom indexes which will allow us to query the data in an efficient manner. The database server will also be secured using a Let's Encrypt certificate and require a client authentication certificate to access. The SQL server will be implemented using the AWS RDS service with a basic SQL server. This will contain user account data.

Both the web API and the database will be hosted using AWS (Amazon Web Services). During development the API and database will both be hosted on an EC2 cloud server. However, for deployment the API will be moved to an Elastic Beanstalk web app hosting (also done by AWS).

2.2 User Characteristics

There will be three types of users that interact with the *CorkCollector* system: tourists, connoisseurs, and moderators. Each user will interact with the application in a different way and may have different abilities based on their intended use of the system, therefore a specific scope must be defined for each.

The first user type is tourists, who would be individuals who are intending to use the application as a one time thing. Tourists would be most interested in interacting with viewing the different wineries and tasting menus and viewing past reviews. They would still need to be logged into the system since we would still want to encourage them to rate/review the places that they visit and the wines that they taste.

The second user type is connoisseurs, who would be individuals who frequent the area and are intending the use the application on a fairly regular basis. They would be less interested in the ratings/reviews (after their first time at the winery) and more interested in being notified when a tasting menu is updated, hearing about winery promotions, and being able to keep track of the different wines that they have purchased. Since these users will be most likely to take advantage of the Cellar it is important to them to be able to view their old comments and search through them.

These users will need to be registered users within *CorkCollector*. This allows for an application community to form since the application is relying on its users to post updates about the wineries, the tasting menus, or any other changes that may need to be made about the information presented. Finally, it is important to note that the account holding user base will range in age from 19-85 (since they must be legal drinking age within the Niagara Wine Region), can be any gender, and come from any education level.

2.3 Constraints

There are several constraints that need to be taken into consideration when developing cork collector. A major constraint upon the development team will be the time schedule to implement the system. By needing to build the product in less than five months, the team will need to be efficient and organized to ensure all necessary features are developed and tested prior to February 27th.

Due to this time constraint there are certain areas of the application which would be left to future development. This includes things like the the wineries have their own user profile to promote the winery. This would make the application more profitable as wineries could pay to sign up but this is just not feasible to complete within our timeline. Another thing that we will not be addressing in this version of the application is that we will not be supporting the addition of wines that are purchased from the LCBO. You will need to be within the proximity of the winery in order to access the wine list or to interact with it in any way above just viewing the current menu and information. This is due to a time constraint and the fact that this does not allow for promotion of the Niagara region.

CorkCollector is a mobile application which means that it will need to be supported on multiple platforms. It will be limited to being supported on only Android applications and only on Android OS that is 4.0 or higher. This means that we will be losing out on a market in the iOS OS side but that would be considered in future work.

All of these constraints will need to be taken into consideration during the development of the application as they do impact our customer base and the functionality which will be added. These constraints will all be addressed in future versions of the application that will continue to be developed post phase 1, or presentation of our capstone.

2.4 Assumptions and Dependencies

CorkCollector will need to be built with several assumptions and dependencies. These will allow the development of the system to be done with greater efficiency and result in a more robust system.

The first assumption we need to make is that the users will allow location access. If we do not know the user's location then we cannot possibly let them know about the wineries in their area and help them to plan a course throughout the day. Although the app would still allow access to the map functionality it would be very unsuccessful in plotting a route for them to take as it would not know their current starting location.

The next assumption that will be made is that users will want to be part of the winery community and will therefore be willing to help update the information provided. Since *CorkCollector* is a very user centric application it relies on users to help improve the application over time. Users will be responsible for reporting new tasting menus, a change in hours or any other changes that may occur within the wineries over time. If users do not wish to improve and contribute to the application then it will become outdated and it will no longer be able to serve it's purpose.

CorkCollector also needs to depend on different systems. It will need to be dependent on the Google Maps API which allows for the wineries to be displayed, if that API goes down or changes then the application will cease to function properly. In a similar vein, if the connection to the database is lost then the users will not be able to view the information left about the wineries or about the wines themselves. *CorkCollector* is entirely dependent on both of these systems so it is vital that they both be given the appropriate amount of thought put into making them reliable.

These assumptions and dependencies are required for the application to be functional and thus must be taken into consideration during development. There must be a fallback plan put in place in case any of the above happen to fall through or break at any point.

2.6 Apportioning of Requirements

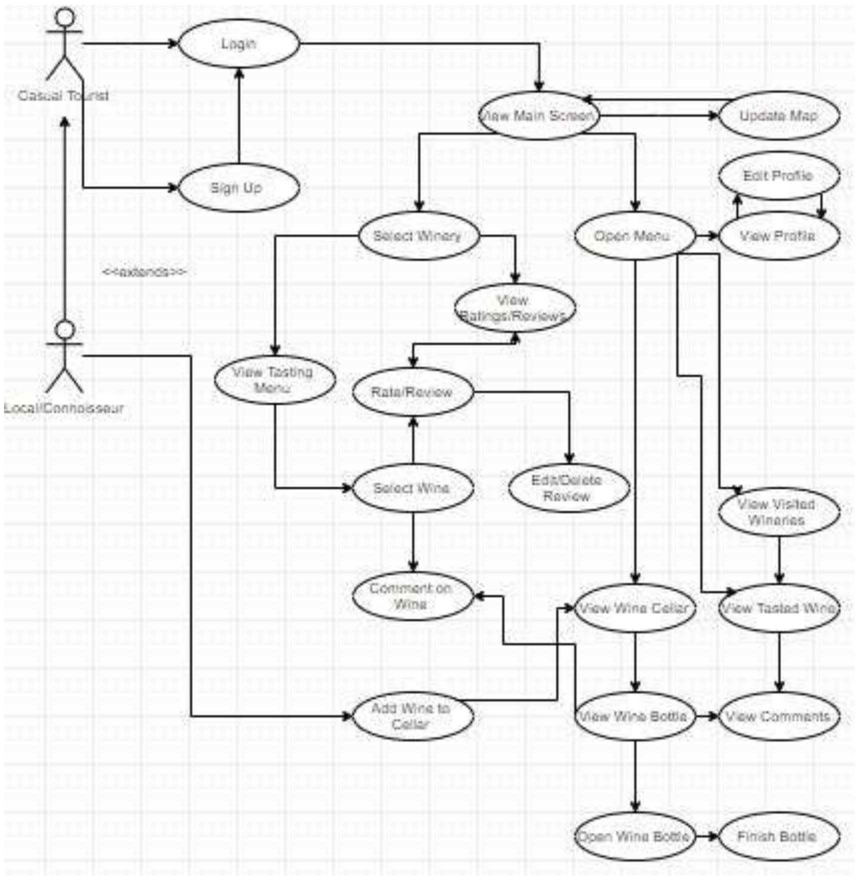
In software development projects, there is always the possibility that there will be delays in the overall development and shipping of a system. Therefore, it will be important to identify which features of the system are critical to the success of the system so that they may be prioritized the highest in the development process, and which features will only be reasonable enhancements to the system.

The team will be using an Agile Scrum methodology for the development process. Therefore, the team will be using sprint cycles of two weeks in length, with deliverables achieved at the end of every sprint. As core features are completed, the team will outline further functional requirements that can be implemented within the timeline of the project which will be determined during the sprint planning phase. After each sprint there will be a retrospective which identifies the issues in the last sprint or things that went well so they may be improved or remedied going forward. This also allows for pulling in tickets which address critical issues found during the testing phase of the last sprint.

For the *CorkCollector* application, the critical features are around the ability to view information about wineries and to view the information about a specific wine. The entire purpose of the application is to enhance the experience of visiting the Niagara region and to encourage a more active participation in the economy so the ability to view information and to interact with wineries is absolutely critical to the success of the application. The list of functional requirements defined in section 3 will be critical features, while further requirements will be outlined upon completion of the critical features.

3. Specific Requirements

3.1 Use Case Diagram



3.2 Functionality

3.2.1 Create User

Description: New users need to be able to create a new user account within the application which will be stored so that they can log back in the next time that they use the application. They will sign up with their first name, last name, email address and a password.
Dependencies: None

3.2.2 Log In

Description: Existing users will need to be able to log into the application in order to use it. They will need to enter their account information which will be validated by the server before they are able to access the application's main window.
Dependencies: 3.1.1

3.2.3 View Map of Nearby Wineries

Description: Users will be able to view a map with pins dropped to indicate the wineries closest to them. The default will be for the screen to be centered around the user's current location showing them the wineries closest to them within a 25km radius. The user can then manipulate the map to view different areas by dragging the map, zooming in, or zooming out.
Dependencies: 3.1.2

3.2.4 Change Password

Description: Users should be able to change their password if they supply their email address and previous password. This can also be done by following a confirmation email link if they have forgotten their previous password.
Dependencies: 3.1.1

3.2.5 View Profile

Description: Each user should have a viewable profile where they can see their own personal information as well as their statistics for use of the application. This will display a count of the wineries which they have checked-in to, a display of the number of wines sampled, and a display of wines which are currently in their Cellar.
Dependencies: 3.1.2

3.2.6 View Visited Wineries

Description: Users will be able to view a list of the wineries which they have visited and checked-in to in the past.
Dependencies: 3.1.2 and 3.1.5

3.2.7 View Tasted Wines

Description: Users will be able to view a list of the wines which they have sampled before. The information will display the wine name, the winery it came from, the year and the style of wine. This can be sorted by name, winery, style or year.
Dependencies: 3.1.2 and 3.1.5

3.2.8 View Wine Bottle

Description: Users will be able to view a single bottle of wine. This will display information on the wine including basic information and the wineries description of the wine. This will also display the user's personal comments on the wine bottle, if there are any.
Dependencies: 3.1.2 and 3.1.7 or 3.1.14 or 3.1.21

3.2.9 View Wine Ratings/Reviews

Description: Users will be able to view the ratings and reviews which have been left on the bottle of wine selected.
Dependencies: 3.1.2 and 3.1.8

3.2.10 View Winery

Description: Users will be able to view information about a winery. This can include the winery name, their own description, and a photo of the location.
Dependencies: 3.1.2 and 3.1.6 or 3.1.?

3.2.11 View Winery Ratings/Reviews

Description: Users will be able to view the ratings and reviews which have been left by other users about the winery selected.
Dependencies: 3.1.2 and 3.1.10

3.2.12 Check-in At Winery

Description: Users will be able to check-in to a winery to state that they have been there. In order for a user to check-in at a winery they must currently be visiting it. This winery will then be added to their Visited Winery list.
Dependencies: 3.1.2 and 3.1.10

3.2.13 Rate/Review Winery

Description: Users will be able to rate/review the winery as long as they have checked-in at the winery. This rating/review will be left for all other users to see (it is public, there is no option to make it private). You may only rate/review wineries you have visited.
Dependencies: 3.1.2 and 3.1.11 and 3.1.12

3.2.14 View Tasting Menu

Description: Users will be able to view the tasting menu which is available at any given time for a winery. This will tell them what is available for tasting as well as the wineries prices for tastings.
Dependencies: 3.1.2 and 3.1.10

3.2.15 Taste Wine

Description: Users will be able to check off a wine on a tasting menu as having been tasted. This will allow the user to track which they have tried. It will then be added to their list of "Tasted Wines"
Dependencies: 3.1.2 and 3.1.8

3.2.16 Rate/Review Wine

Description: Users will be able to rate/review a wine that they have tried. This will be available for other users to view. You may only rate/review wines you have tasted.
Dependencies: 3.1.2 and 3.1.9 and 3.1.15

3.2.17 Edit/Delete Review

Description: Users will be able to remove a rating/review.

Dependencies: 3.1.2 and 3.1.9 or 3.1.11

3.2.18 Comment

Description: Users will be able to leave a personal comment on a bottle of wine. This will be a private comment which is not available to the public, only that specific user.
Dependencies: 3.1.2 and 3.1.8

3.2.19 Add Wine to Wine Cellar

Description: Users will be able to add a bottle of wine to their “Wine Cellar” when they purchase that bottle from the winery.
Dependencies: 3.1.2 and 3.1.8

3.2.20 View Wine Cellar

Description: Users will be able to view the bottles of wine which they have purchased from wineries and added to their cellar. This displays basic information on the bottle (name, winery, year, style) as well as number of bottles on hand and can be sorted on this information.
Dependencies: 3.1.2 and 3.1.5

3.2.21 Finish a Bottle of Wine

Description: Users will be able to mark a bottle of wine stored in their cellar as being “Finished”. This will take it out of the “Wine Cellar” list and move it to the “Cork” list.
Dependencies: 3.1.2 and 3.1.21

3.3 Usability

Usability is a key aspect of *CorkCollector*. Its major components are broken down in the subheadings below.

3.3.1 Simple and Intuitive User Interface

CorkCollector’s target demographic is broken down into two main categories: Tourists and Connoisseurs. In the first case, due to the diversity of its users, the application must be simple enough for any potential user to pick up and quickly understand - regardless of their wine knowledge or proficiency in English. In the latter, due to the higher average age of locals and connoisseurs, it must be simple enough to use repeatedly without frustrations, and cater to individuals with all kinds of technological aptitude. In both cases, users will often be interacting with *CorkCollector* under the influence of alcohol, and for the application to be successful, it must still be easily legible and comprehensible while in a slightly inebriated state.

3.3.2 High Accessibility of Key Functions

CorkCollector makes use of a variety of functions to engage its users and tailor their experience with the application. Researching and reviewing wines and wineries provides users with an opportunity to feel heard, and connect with other users across the VQA community. *Cellar* functionality allows them to track the age and origin of any VQA wines they purchase, and note-taking features ensure their initial opinions can guide future purchases and tasting experiences. Keeping these main functions accessible to the user and easy to find at all times ensures a higher level of variety and interaction with the application, and gives users an enticing reason to frequently return to *CorkCollector*.

3.3.3 Navigable and Cohesive Application Design

In keeping with the previous requirements, *CorkCollector* should be completely self-contained, with no reason for users to need to access any external information sources. *Cellars*, tasting, menus, maps, notes and review pages should all be accessible from the menu, and their content should occupy the same screen (one at a time). A master menu, or easily accessible control source, should be available for the user to efficiently switch between the different components at their leisure.

3.3.4 Usability Metrics

The usability of the application will be measured by a trial series of users - ideally ones using the application as intended in Niagara. As this field is largely subjective, the success of our product to meet our targets will be measured in an informal questionnaire. It will contain entries such as:

- On a scale of 1 to 5, how easy was the application to use?
- On a scale of 1 to 5, how easy was it to find your cellar, profile and other pages?
- Are there any changes you would make to the layout?

The numerical values will be used to measure trends over time. User feedback will be incorporated into further iterations of the application's design.

3.4 Reliability

Due to its simple design scheme and lack of computationally taxing operations, *CorkCollector* should not have any issues remaining stable. Its specific requirements are outlined below.

3.4.1 Serve Over 250,000 Requests Daily

CorkCollector requires a variety of different interactions and server requests from each user to offer a full experience. Even if the user is only a tourist, they are expected to make several requests with *CorkCollector* in one session. Statistical data shows an estimated 1.8 million[1] tourists visit the Niagara region each year, and the population of the Niagara region growing to nearly half a million[2]. Factoring in seasonality, during peak times we expect the number of daily users to reach 2500, and by extension, requests to reach upwards of a quarter million. We expect this requirement will not go beyond the capabilities of our server system, and can be achieved without failure or sharp drops in performance.

3.4.2 Minimize Server Downtimes to 2 Minutes

While the sporadic user pattern is not expected to cause heavy strain on the server, in the event that an unforeseen issue arises, our target is to keep any downtimes to under 120 seconds. During this time, the application will not crash or close, but cache their activity so that it can be restored once reopened, display a message informing the user of server errors and remind them to check back in a few minutes. This time will be used for system administrators to reboot the server and address any issues that may have arisen.

3.4.3 Nearly Eliminate Server Failures

CorkCollector is not an application that is expected to garner heavy traffic over winter months, due to the reduced frequency of outdoor wine tastings and tours, and the seasonal lack of tourists. In summer months, usage is expected to be widely and evenly spread across the time period, maintaining a relatively low and stable frequency of server interaction. While it is impossible to say with certainty that the server will permanently run without issue, barring an unpredictable error, we do not expect it to fail. To accomplish this, measures must be taken to maintain performance, including clearing memory storage with automated methods.

3.4.4 Preemptively Limit Defect Rates to Under 10 Percent

CorkCollector's development will feature a heavy focus on agile sprints and integrated testing. As such, faults in the program are expected to be discovered and addressed as early as possible to prevent more serious issues later on. The primary focus of the simultaneous testing and development is to find and fix these issues, keeping a consistent maximum deficit rate to a reasonable 10 percent at all phases of project implementation.

3.4.5 Attain Nearly-Perfect Application Availability

Users will need to access the application at a multitude of times and situations, while at a wine tasting or at home. As such, *CorkCollector* must always be available for them. Naturally, server components cannot be expected to work perfectly for an indefinite time, but barring any accidents and major system updates, we will aim to make the application consistently available.

3.4.6 Reliability Metrics

As reliability is a varied field, certain techniques will work best for certain components. For server goals, such as service requests, downtimes and failures, programs can be run on the backend to mock application usage. These can create fake requests to put the server under some duress and test its ability to meet the team's requirements - and cause changes if necessary. As for less tangible qualities, like the latter two sections above, these can be measured by communicating with users and between the development team and making adjustments to the product cycle or server settings as necessary.

3.5 Supportability

While the initial loadout of *CorkCollector* may not require a high degree of developer support, all planned extensions and added features beyond our initial scope will make heavy use of it. Below are requirements that lay the groundwork for attaining those goals.

3.5.1 Consistently Follow a Coding Standard

Code written by multiple authors inevitably lapses into conflict when personal standards and preferences are at play. In order to avoid errors, and keep code clean, readable and uniformly formatted, we will ensure all written C# files follow dofactory's[3] coding conventions. We expect this will greatly simplify communications and edits between different members of the team, and eliminate formatting and readability issues outright.

3.5.2 Write Reusable and Modular Front-End Code

CorkCollector will be programmed in modern, highly functional languages that lend themselves well to all aspects of reusability and modularity. Given the tools at our disposal, we will ensure all of *CorkCollector's* code is efficiently written, with no unneeded repetition or crossover. This will minimize time to add new basic features, and ensure the application is built cleanly, with consistency for developers and users in mind.

3.5.3 Minimum Code Coverage of 80% for Unit Tests

To maintain a consistent level of functionality and accuracy, we aim to create unit test that cover a large majority of the code. This coverage level will enable the development team to catch and monitor any unforeseen issues that arise prior to launch, and ensure a lack of complicated errors when any additions or extensions to the source code is made. Due to the relatively small size of the development team, we believe 80% is a fair baseline target that will root out and prevent enough problems for the application to run smoothly.

3.5.4 Supportability Metrics

As supportability is not an innately measurable field, some interpretative liberties must be taken in this section. In this spirit, the first two sections cannot be strictly measured, but the development team will strive to work with them as guidelines. Nevertheless, code coverage is able to be tracked and for all relevant parts -of the application, unit testing is compulsory. Tickets and tasks during development are not marked as complete until they have been properly tested, resulting in coverage that spans the application - with records for future reference.

3.6 Performance

3.6.1 Support Up to 80 Transactions Per Second

We expect *CorkCollector* to be used heavily during winery tours, where a multitude of users will access the app to write notes, or post a review for a winery they are currently visiting. These interactions will likely take place in the same concentrated time frame, so to ensure a smooth experience for all users we aim to have the servers handle up to 80 transactions every second.

3.6.2 Minimize Loading times to 3 Seconds

Minimizing wait time is a major component of successful mobile application design. This is especially important to *CorkCollector* because it is intended to be used during wine tours and tastings. If users experience a delay of anything more than three seconds, we can expect them to become bored and return to their activity. To accomplish this goal, we plan to strategically load content on each page - if it is not in use, it will not be loaded until the user requests it.

3.6.3 Minimize Latency to 3 Seconds for Niagara Region Users

As latency is dependent on distance to the server, our launch plan for *CorkCollector* depends largely on placing our primary server in the region. We expect this will provide an even and short latency time for all regional users. We expect users who do not reside in the Niagara region to use the application far less frequently (if at all), but those in Ontario will still encounter a reasonable latency time and other tourists using the application will have no issue with latency while they are visiting Niagara.

3.6.4 Support Over 2 Million Users

As survey data suggests[2], nearly 2 million tourists visit the NIagara region every year. Our target user capacity matches this number to deal with projected growth of the tourism industry - even if only a fraction will use the app - as well as local users. While we consider it unlikely to reach full capacity within the first few years of release, it is better to be prepared for such a scenario. To handle this, we will implement a cloud-hosted scaling system to keep pace with user demand. Should the user base exceed limitations, we will remove accounts that have been inactive for a prolonged period of time, from one-time tourists with no further use intended.

3.6.5 Performance Metrics

The best way to measure backend functionality is to monitor the server and database information. This can be used to track user count and transactions per second, and make adjustments if they are not within the target range, or if the application cannot handle the technical demands placed upon it. As for measuring loading and latency times, individual tests will be carried out across a multitude of cellular devices to paint a clear picture of the application’s speed. If changes are required, they will be made accordingly, keeping in mind that certain variables like network speeds are beyond the development team’s control.

3.7 Design Constraints

The next section will describe constraints involved in the development of Cork Collector based on the system components we plan to use.

3.7.1 ASP.NET C# Web API

The ASP.NET Web API framework provides us with built in support REST API conventions. It also has strong support for URL routing that will allow us to create clear routes based on MVC semantics. The framework is lightweight and adaptable as it can handle multiple output formats (JSON, XML, ATOM, etc.) and can be queried by mobile and web browser clients alike. Furthermore, numerous plugins and libraries have been developed for this framework and can be leveraged through the Nuget packages to assist in the development of *CorkCollector*.

3.7.2 RavenDB - NoSQL Document Database

A NoSQL database provides fantastic scalability and the ability to modify the schema during development with minimal effort. However, the document database structure means we need to have a good understanding of the data structure and the queries we will need to make before implementing anything. This is because, the document structure creates a nested format that provides easy access to top level entities but would make finding deeply buried entities a high cost operation. This means we need to plan what data is needed most frequently and design a structure that allows us to quickly query this data. The RavenDB implementation provides us with numerous plugins for accessing and modifying data inside the database.

3.7.3 Let’s Encrypt Digital Certificates

Let’s Encrypt is a service that provides free secure certificates to clients for use in their applications and websites. In order to make use of this service, we will need to own a domain name to create the certificates under. The certificates come in two forms: staging and live. Staging certificates are unlimited and can be reissued as often as we want for our application. These will be used during development. The live certificates have a limited number of requests per day but have a much later expiration date. These certificates will be used for our production version.

3.7.4 XUnit Unit Tests

Unit tests for our web API will be written using XUnit. XUnit is a testing library that integrates well with C# applications and allows us to quickly identify any areas that errors are appearing in. XUnit comes built in with assertion functionality and can be used to call and check the outputs on any functionality within our code base. XUni also provides support for utilizing an embedded RavenDB in our testing so we can be certain the data used by the test is up to date and accurate. The full test suite can be ran at any time (it can even be set to run on every build) and will inform us of any errors.

3.7.5 Android Application - Java using Android SDK

The Android application will be targeted to Android devices on version 4.0 or greater. This prevents us from utilizing Android SDK functions implemented in versions after 4.0. Version 4.0 was chosen because it targets the majority of Android users. The application will be implemented using Java.

3.7.6 Github and GitFlow

The code for our application will be stored in a Github repository. The development of this code will be managed using a GitFlow model. GitFlow is a development model which creates two primary branches: Development and Master. Master contains the tested and released code and is marked with tags to track the version number. The Development branch holds the most up to date code. Users branch off the development branch into features for new development and are not merged back until they have been fully tested. Release branches are created from development and merged into Master for releasing new features. This model allows us to accurately track the features that have been integrated into the system, the features that have been released and allows developers to work on new code, test, and patch simultaneously without affecting others work.

3.7.7 Agile Development Process

We will be using an Agile development process to develop the *CorkCollector* system. The Agile approach will allow us to effectively target the most important features of the system first. By using Trello to track our progress we will also be able to review what we accomplish in each sprint and plan for better sprints every time. Focusing on specific issues will also help us to ensure that we meet all requirements of each feature before moving on.

3.7.8 Octopus Deploy Server

We will be using an Octopus Deployment server in order to deploy new versions of the web API. Octopus deploy allows us to stop the old API that is running and deploy the new selected version in its place with just the push of a button. This will allow for very quick updates and minimal user interruption. Octopus also keeps track of all old releases of a project and allows you to roll back to a previous version using the same process.

3.8 Online User Documentation & Help System Requirements

CorkCollector needs to be very intuitive to use due to the fact that one category of users will only be using it once. Due to this there should be very little learning curve and therefore there should be very little need for help documentation. There will be information options on each page to allow the user to get a quick description of the page and what it’s capabilities are should they absolutely need it. There are also the users who function as moderators who will help to control the overall health and energy within the application in regards to the user ratings and reviews. Should there be any issues with the system, there is the ability to contact the *CorkCollector* team to raise any kind of concerns or questions that may need to be addressed and cannot be done from the user’s own abilities.

3.9 Purchased Components

CorkCollector will be making use of primarily free components and services for the development of the application. This section will explain the components that would require a purchase if the application is implemented on an industry scale. First, the free community edition of RavenDB’s

NoSQL database would need to be upgraded to their “Professional Edition” at 750$ per year. Next, for hosting we will be developing using components in the AWS Free Pricing Tier which allows for 12 months use of the EC2 Cloud Computing Services. When this expires the pricing would switch to a dynamic pay for use method which, based on CPU usage, ranging from $0.0087 per hour to up to $10.28 per hour for each necessary server (https://aws.amazon.com/ec2/pricing/on-demand/). The number of servers would be based on the amount of users who regularly use the app. A domain name would also need to be purchased for a corporate website which would cost approximately $12.00 per year. Finally, the Octopus deployment server trial would need to be upgraded from the trial version to the standard version at a cost of 300$/year.

3.10 Interfaces

The following sections will explore the user interfaces that will be developed for *CorkCollector*. The hardware and software interfaces will also be explored, as well as communication interfaces. For all Figure references, please refer to Appendix A.

3.10.1 User Interfaces

The first page displayed to a user is simply the main home screen (Figure 1). This screen will simply show a quick button which allows you to click to open up the log in or sign up screen. The quick access button will display the *CorkCollector* logo once it has been finalized.

In order to use the applications users will need to be logged in. Therefore one of the interfaces that they interact with will be the log in/sign up screen (Figure 2). The reason for requiring log in is that the application is entirely focused around users rating/reviewing so that this will enhance the experience for other users visiting the area. Once a user has logged in they will be presented with the main screen (Figure 3) which displays a map of the area to them. This map is populated via integration with the Google Maps API and allows for the normal interactions that a Google Map would allow (zoom in, zoom out, move around etc.).

Once a user has actually selected a winery from the map they will be presented with the wineries home page (Figure 4). This page will display information about the selected winery. Each winery will have the following information:

1. Winery name
2. A photo of the winery
3. Average rating of the winery
4. Address
5. Phone Number
6. Action for viewing that winery’s tasting menu
7. Action for getting directions to the winery
8. A list of ratings/reviews from other users
9. Action for leaving a rating/review

This page will also have a search bar which allows the user to search for other wineries.

From the winery page a user may select to view the winery's current tasting menu. This will pull up the winery tasting menu page (Figure 5). This page will display some very basic information about each wine on their tasting menu. Each tasting menu will display (in list form):

1. The wine name - clickable to provide the action of displaying wine information
2. Average rating of the wine
3. Type of wine
4. Year of bottling

It is important to make it obvious that each wine is a clickable link to bring them to the wines full page so they will be displayed underlined. This page will also be the same layout used for our page on tasted wines, which displays the list of wines a user has tasted, the wine cellar, which displays the list of bottles that a user had purchased and not consumed, and the corks collected page, which displays the list of wine bottles a user has consumed.

Once a user has selected a wine from the tasting menu they will be presented with the wine home page (Figure 6). This page will display information about the selected wine. Each wine will have the following information:

1. Wine name
2. Photo of the bottle
3. Average Rating
4. Type/Year
5. Winery's description of the wine
6. Action to mark the wine as tasted
7. Action to add a bottle of the wine to your cellar
8. A list of ratings/reviews from other users
9. Action for leaving a rating/review
10. Action for leaving a personal comment that only

This page will also have a search bar which allows the user to search for other wineries.

On the winery page and on the wine page the users are presented the option to leave a rating/review. The rating/review page (Figure 7) is the same for each, but when the data is saved it will have a reference to whether it is to a winery or a wine as well as a reference number to which winery or wine it is for. The options presented a fairly basic but will be as follows:

1. Text input box to leave the rating
2. A rating star system where you select the maximum number of stars to give
3. Action to submit the rating/review

This page will also be used for leaving the user's personal comments. The difference for that page will simply be that the ratings will not appear and the title will be "Comment" instead of "Rating/Review".

The last page that a user is able to view is their own profile page (Figure 8). This will be accessible from the menu in the upper left hand corner. This page will display the users basic information as well as statistics. The information displayed will be:

1. The user's name

2. The user’s photo
3. Their reward star rating (based on how much they have contributed to the application)
4. Their current level (i.e. Wine Newbie, Wine Lover, Wine Expert etc. based on contributions to the application)
5. User Statistics
    a. Wineries Visited (out of the current total wineries in the area)
        i. Average given to the wineries visited
    b. Wines Tasted
        i. Average given to the wines tasted
    c. Wines currently in the user’s cellar
        i. Average of the wine’s in the user’s cellar
    d. Corks Collected - the wine bottles that the user has finished

The user statistics will be clickable links which bring them to the pages which list the wines in those lists for that user. These lists have been described above as being similar to the winery tasting menu page.

As has been stated above, many pages are being reused and simply displaying different information to the user. This is going to help us maintain a very easy to follow flow throughout the application. By making our pages dynamic and robust we can reuse the same layouts and simply populate the data differently based on which page the user is trying to access. We are hoping this will assist in the usability of the application as users will not need to learn new page layouts for different areas of the application.

3.10.2 Hardware Interfaces

Since *CorkCollector* is strictly a software application, it does not rely on any hardware interfaces. The only interaction with hardware is the device that the user uses to visit the application, and the servers that host the web services for the system. *CorkCollector* will also use cloud solutions to ensure scalability of the hardware components is handled automatically and will not interfere with the production of the application.

3.10.3 External Interfaces

CorkCollector will make use of the Google Maps API to leverage its existing functionality for location based services. To access the google API we will be given an API key that will identify *CorkCollector* to the Google Maps API. We will then be able to access accurate, up to date information for geographical information and a UI Map display that we can use on the front end to display the information. This map can be stylized (much like what Uber has done with their application) with our own markers and display formats.

3.10.4 Communications Interfaces

CorkCollector will rely on an internet connection for its operations. If a user’s device loses this internet connection, the application should not crash but should inform the user of the missing internet connection on any future request attempts and remain in its most recent state until the connection problems are resolved.

The application will also be relying on GPS location services from user devices. Since this feature can be turned off by the user the application, and the user may not want to provide us with location information due to privacy preferences, the application should be able to function without this data. This means the user should be able to use the Map interface to navigate around to different areas without providing their own information. However, since directions and check-ins will rely on knowing the user's current location, these features should inform the user if they attempt to use them without location data, and request permission for that data.

3.11 Legal, Copyright, and Other Notices

CorkCollector is an information platform, and as such does not incur any liabilities related to the selling or distribution of wine. Furthermore, it is not accountable for any issues that may arise as a result of a user visiting a winery, purchasing a bottle of wine or taking part in a wine-tasting tour. All of these legal disclaimers will be presented to the user when they register.

Additionally, because *CorkCollector* is a user-based application, the development team will not be held responsible for the accuracy of information displayed by the application. In its current version, all users have the tools to update the system with new information, and flag erroneous information and inappropriate comments. The development team will intervene in the case of problem users who repeatedly break community standards.

3.12 Applicable Standards

In its initial release, *CorkCollector* will cater to the Android platform only. As such, its functionality and compatibility will be tested with all current varieties of the Android Operating System. Any incompatibility will be noted presented to users upon release.
In order to cater to users with some degree of visual impairment or colour blindness, user interfaces and application layout will both make heavy use of bright, contrasting colours and simple, easy-to-read text. See Section 4.1 Appendix A for an example.

4. Supporting Information

4.1 Appendix A: Mockups

Figure 1. Home Screen

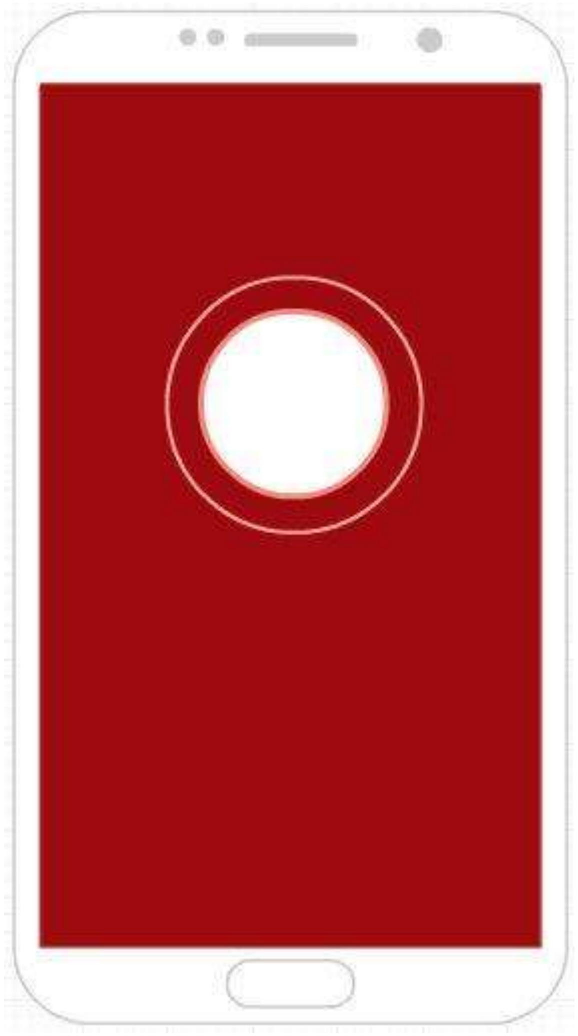


Figure 2. Log in/Sign up Screen

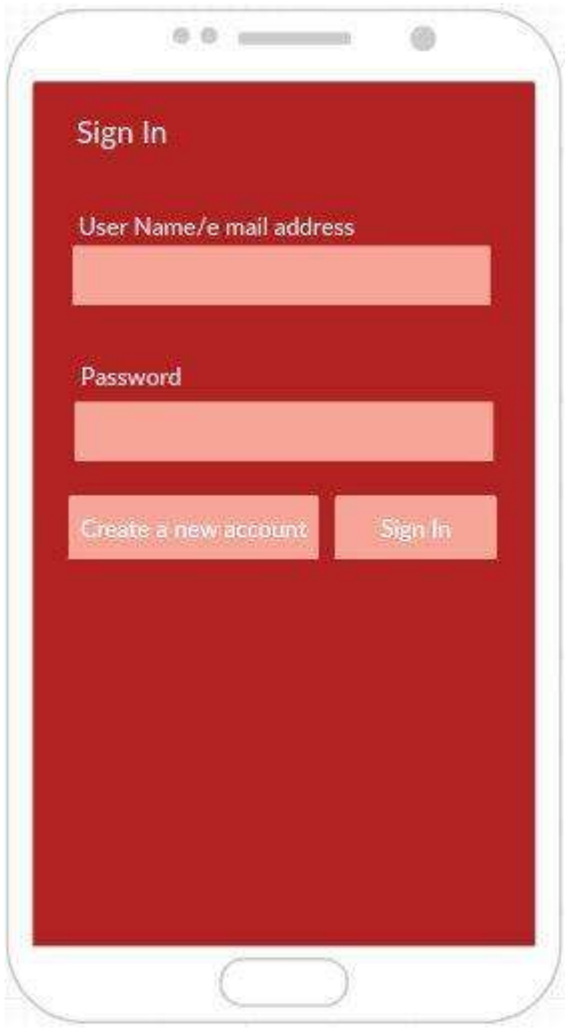


Figure 3. Google Maps API main screen

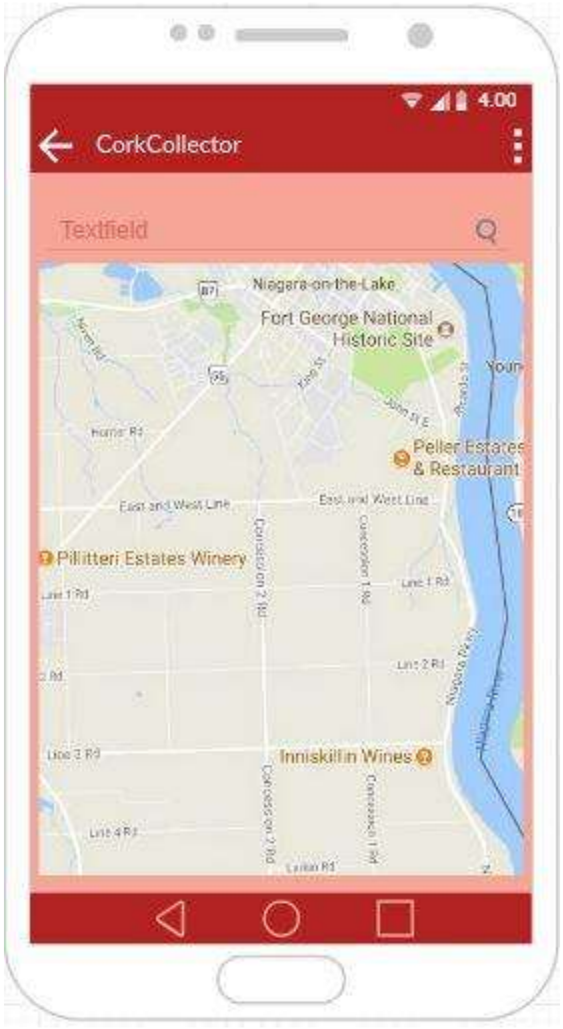
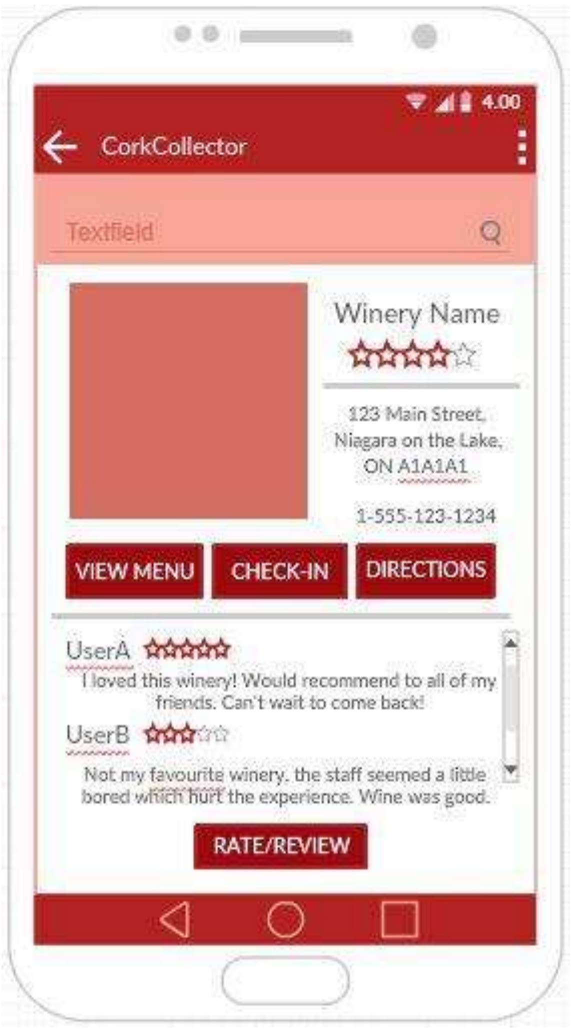

4:00
CorkCollector
Textfield
Niagara-on-the-Lake
Fort George National Historic Site
Peller Estates & Restaurant
Pillitteri Estates Winery
Inniskillin Wines
East and West Line

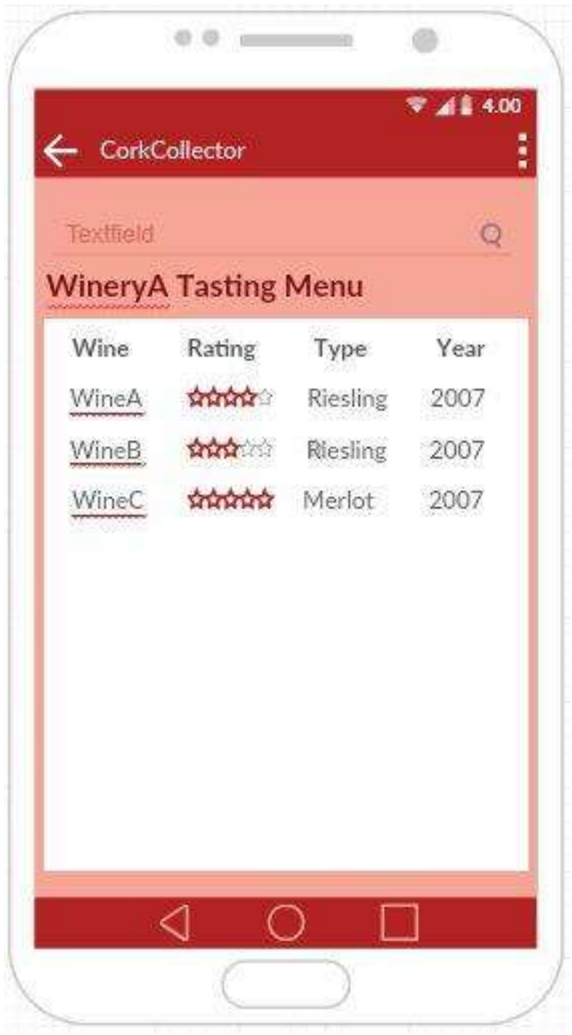
Figure 4. Winery Page



I loved this winery! Would recommend to all of my friends. Can't wait to come back!

Not my favourite winery. the staff seemed a little bored which hurt the experience. Wine was good.

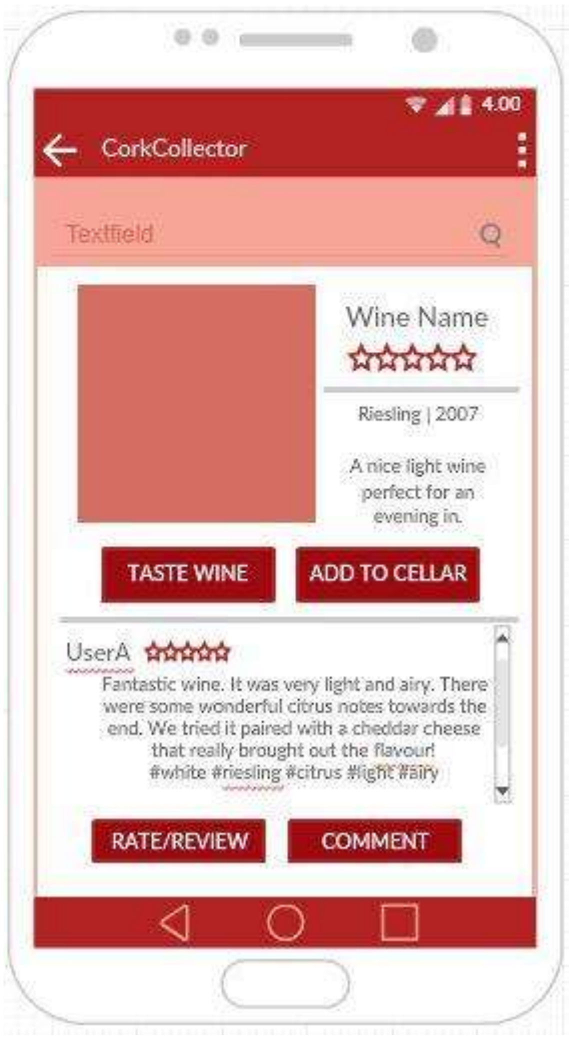
Figure 5. Tasting Menu for WineryA



WineryA Tasting Menu

Wine	Rating	Type	Year
WineA	★★★★☆	Riesling	2007
WineB	★★★☆☆	Riesling	2007
WineC	★★★★★	Merlot	2007

Figure 6. Wine Page



Riesling | 2007

A nice light wine perfect for an evening in.

UserA ★★★★★

Fantastic wine. It was very light and airy. There were some wonderful citrus notes towards the end. We tried it paired with a cheddar cheese that really brought out the flavour! #white #riesling #citrus #light #airy

Figure 7. Rating/Review Page

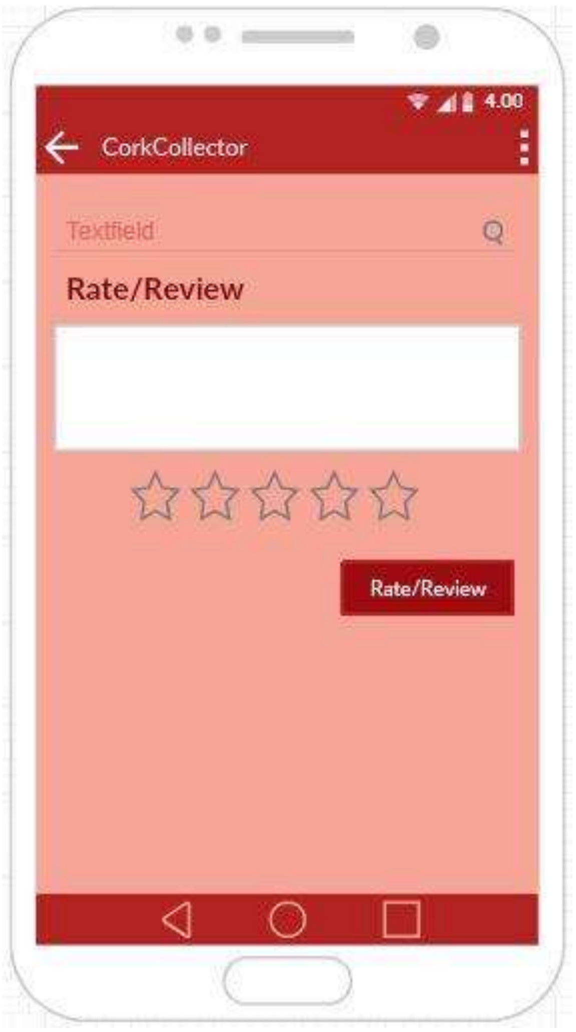
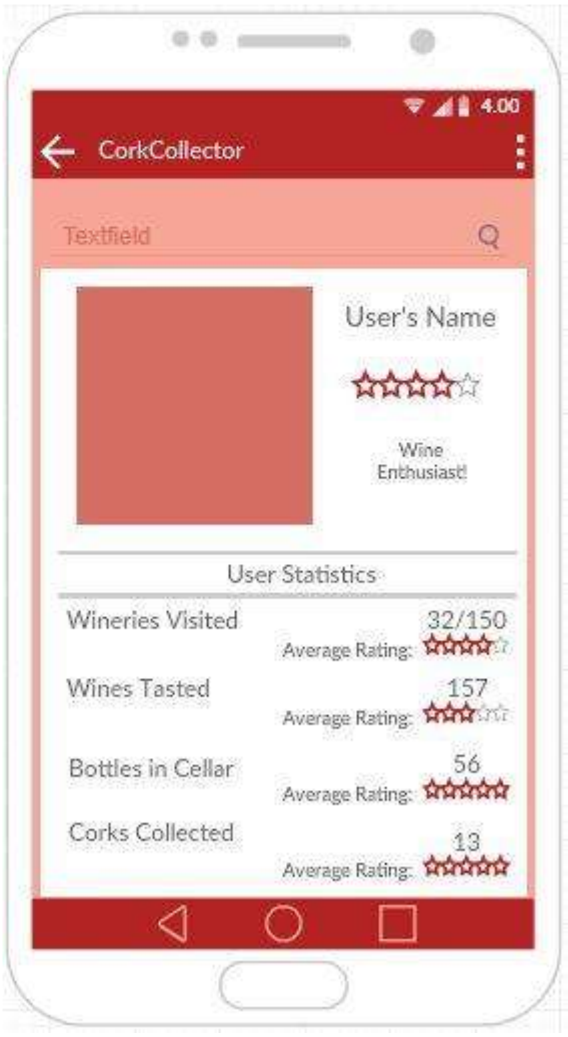


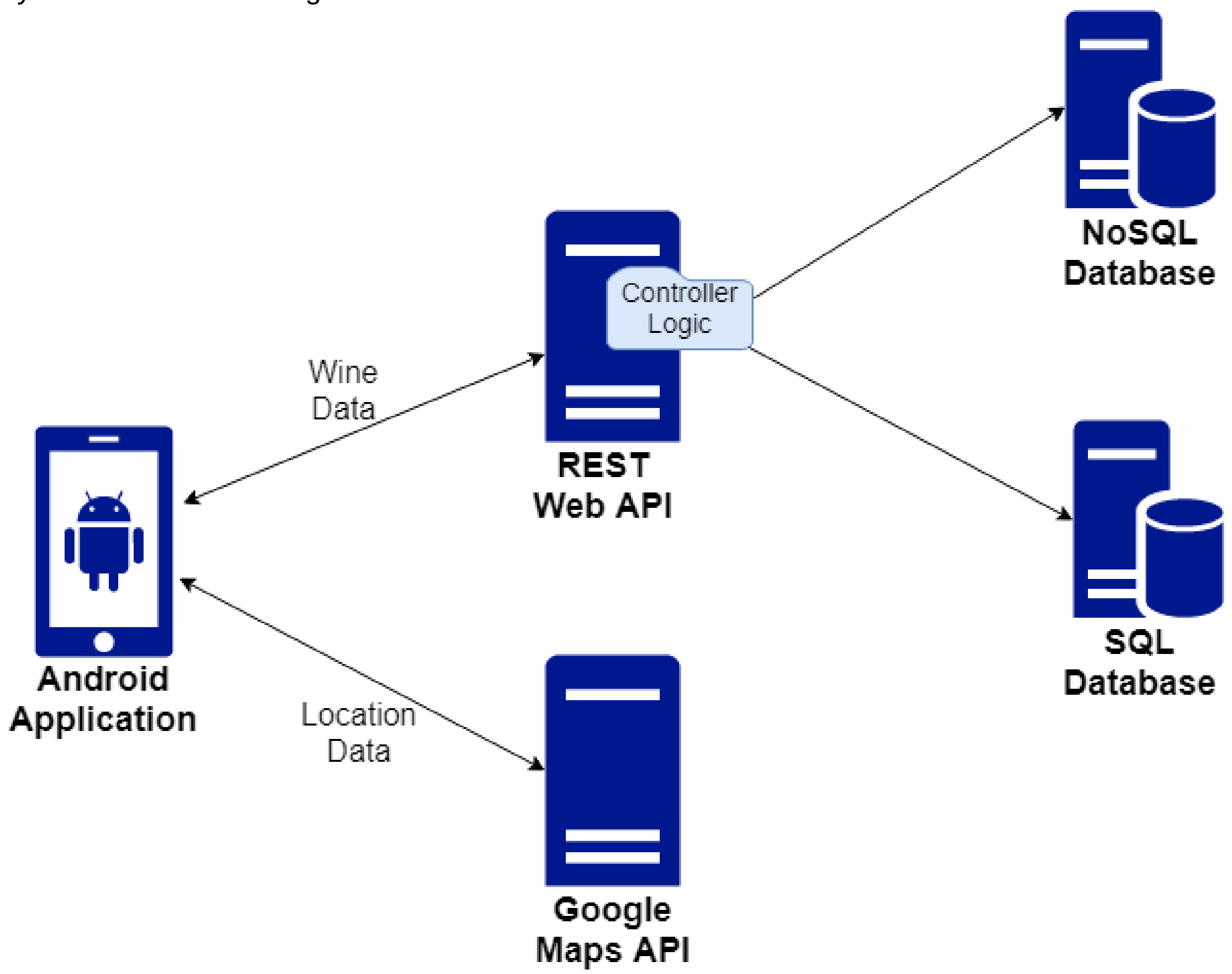
Figure 8. User Profile Page



Wineries Visited	32/150	Average Rating:
Wines Tasted	157	Average Rating:
Bottles in Cellar	56	Average Rating:
Corks Collected	13	Average Rating:

4.2 Appendix B: Architecture

System Architecture diagram



CorkCollector

Software Design Specification

Date: April 11th, 2018

Russell Stirling	250757946
Daniel Lorencez	250741921
Bailey Hanna	250718425

Revision History

Date	Version	Description	Author
2017-11-15	1.0	Initial document section headers created.	Daniel Lorencez
2017-11-17	1.1	Sequence and use case diagrams created and inserted.	Bailey Hanna
2017-11-19	1.2	First draft of individual sections.	Bailey Hanna Daniel Lorencez Russell Stirling
2017-11-22	1.3	Class, package and database diagrams added.	Russell Stirling
2017-11-24	1.4	Sections revised by different group member than initial author.	Bailey Hanna Daniel Lorencez Russell Stirling
2017-11-28	1.5	Final draft of all sections completed.	Bailey Hanna Daniel Lorencez Russell Stirling
2017-11-29	1.6	Document formatting.	Russell Stirling
2018-03-17	2.0	Added user manual and price plan.	Bailey Hanna
2018-03-18	2.1	Added abstract.	Daniel Lorencez
2018-03-20	2.2	Updated class diagrams, package breakdown and NoSQL structure diagrams based on changes made in development.	Russell Stirling
2018-03-25	2.3	Added conclusion and recommendations and revised based on midterm report feedback.	Bailey Hanna Daniel Lorencez Russell Stirling
2018-04-02	2.4	Document formatting.	Bailey Hanna

1. Introduction

1.1 Purpose

This document serves as a detailed guide and reference for the system architecture of *CorkCollector*. It includes all background information, detailed analysis and planned implementation strategies for *CorkCollector*'s logical architecture and functional components. It will serve as a plan and reference point for the team during development, as well as a simple, comprehensible breakdown for the project advisor.

1.2 Overview

This document contains four main sections: The introduction (which you are currently reading), the logical architecture summary, a detailed description of system components and an elaboration on the system design and why it was chosen.

The second section will lay out a high-level view of the system architecture, including package descriptions and an MVC diagram. The third section will provide a close-up view of the system's class and functional architecture, supported by class and sequence diagrams and an in-depth description of all relevant components. The final section will elaborate on the system's overall architecture and provide a concise explanation as to why it is optimally designed.

2. Logical Architecture

This section will be going through a high level overview of the various components to be used in *CorkCollector*. It will focus on the primary components of the application, the systems design architecture and provide a breakdown of the various packages involved.

2.1 Overview

This section will discuss the four primary components of *CorkCollector* and the responsibilities each component has

2.1.1 User Component

The user component will be dealing with information related to user profiles. This involves handling authorization during login, creating new user profiles, updating user profile information and statistics, and displaying all this information to end users. This component will also have various management functionalities, available only to developers, for user management on a large scale.

2.1.2 Wine/Winery Component

The Wine/Winery component will focus on handling activities related to wines and their wineries. This involves creating and updating the data to describe wines and wineries, grouping wines into menus based on the winery tasting menu or the user’s cellar, and displaying pages with this information to the user.

2.1.3 Location Component

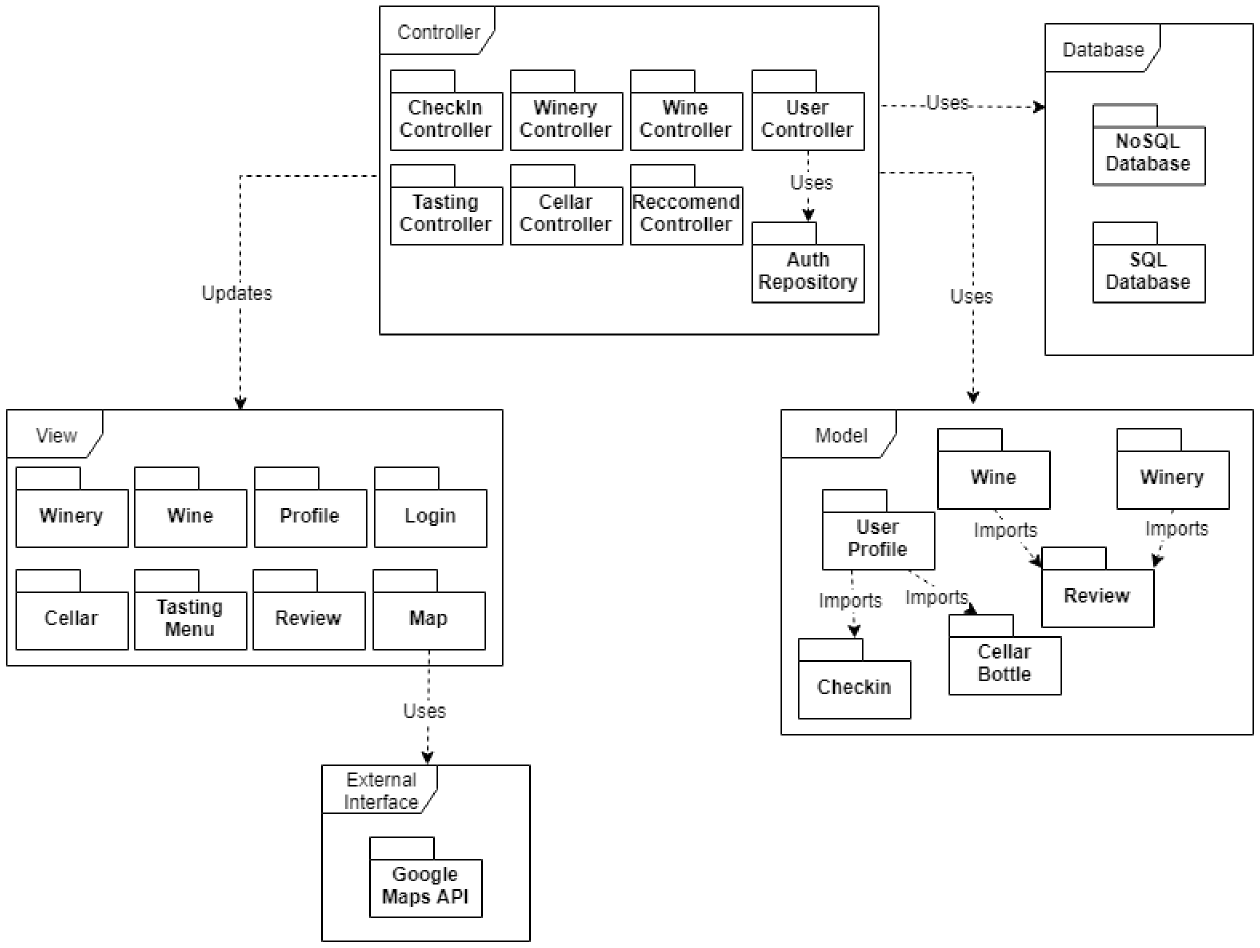
The Location component will be handling functionality related to location tracking and display. It will integrate with the Google Maps API to display a detailed map of the Niagara region with known wineries marked, show the user their current location relative to these wineries and allow the user to request directions to any of the wineries. This component will also be used to ensure a user is within a certain distance of a winery when they check in.

2.1.4 Review Component

The Review component will be handling the review and ratings created by users. It will associate them with wines and wineries as necessary and update average wine and winery ratings based on the reviews. The component will also recommend new wines to users to try based on the previous ratings they have left. Safeguards will need to be in place to ensure users cannot leave vulgar or inappropriate reviews.

2.2 Package Breakdown

The *CorkCollector* application was structured in the Model-View-Controller design pattern. This section of the document divides the components of the application into packages and subpackages based on this design pattern and describes the purpose of each.



2.2.1 View Package

The view package is responsible for handling user interaction with the app. This includes displaying views and getting input from the user. The view package contains seven sub-packages based on the different unique views available within the app.

2.2.1.1 Winery

The Winery view will be responsible for displaying all pertinent information about the winery and providing the user with actions such as checking in, leaving a review, getting directions, or viewing the tasting menu.

2.2.1.2 Wine

The Wine view will be responsible for displaying all pertinent information about a specific wine. It will also provide users with wine related actions such as tasting, adding to their cellar, reviewing, or leaving a personal note.

2.2.1.3 Map

The Map view will be responsible for displaying the user's current location and all nearby wineries.

2.2.1.4 Login

The Login view will be responsible for providing the user with a form that can be used to either login or sign up for a new account.

2.2.1.5 Tasting Menu

The Tasting Menu view will display a list of wines with basic information about each wine. The user can use the search field to query for specific details and select a wine to view it's wine page.

2.2.1.6 Review

The Review view will display a user review of a specific wine or winery. When creating the review a text field and numeric rating scale will be available to set by the user. Once a review has been created the view will provide the ability to edit or remove the view for users who created it.

2.2.1.7 Profile

The Profile view will display individual user profile information such as statistics of the user's app usage, their overall rating, etc. The user will also be able to access their cellar from this page.

2.2.1.8 Cellar

The Cellar view will display the list of bottles stored in the users cellar along with their counts and personal comments.

2.2.2 Controller Package

The controller package is responsible for accessing the database via the models, and retrieving or modifying the data. It returns data to the view to be displayed and contains four sub packages based on the types of data it needs to handle.

2.2.2.1 Wine Controller

The Wine controller is responsible for handling wine data. It handles retrieving the data when viewing wines, wine lists, their reviews and modifying the data when necessary.

2.2.2.2 Winery Controller

The Winery controller is responsible for handling winery data. It handles retrieving the data when viewing wineries, their reviews, or returning a list of nearby wineries and modifying the data when necessary.

2.2.2.3 Recommend Controller

The recommend controller is responsible for generating possible recommendations for users who request them.

2.2.2.4 User Controller

The user controller is responsible for handling requests related to user data. It deals with user authentication, modification and creation of user data, and retrieval of user data.

2.2.2.5 Auth Repository

The auth repository handles connecting to the SQL database to create new user account information or check for existing account information.

2.2.2.6 Tasting Controller

The tasting controller is responsible for posting and retrieving user tastings.

2.2.2.7 CheckIn Controller

The checkin controller is responsible for posting and retrieving user checkins.

2.2.3 Model Package

The model package provides concrete data structures for loading and interacting with data from the database.

2.2.3.1 User

User Models will provide data structures for storing data related to the user, such as profiles, authentication tokens, etc.

2.2.3.2 Review

Review Models will provide data structures for review data objects such as review, rating, etc.

2.2.3.3 Winery

Winery Models will provide data structures for the wine related data objects such as wineries, wines, tasting menus, etc.

2.2.4 Database Package

The database package contains database implementations for storing relevant app data.

2.2.4.1 NoSQL Database

The NoSQL database stores document data for the rapidly growing and changing winery information.

2.2.4.2 SQL Database

The SQL database stores important user account information such as usernames, hashed and salted passwords, emails, etc.

2.2.5 External Interface Package

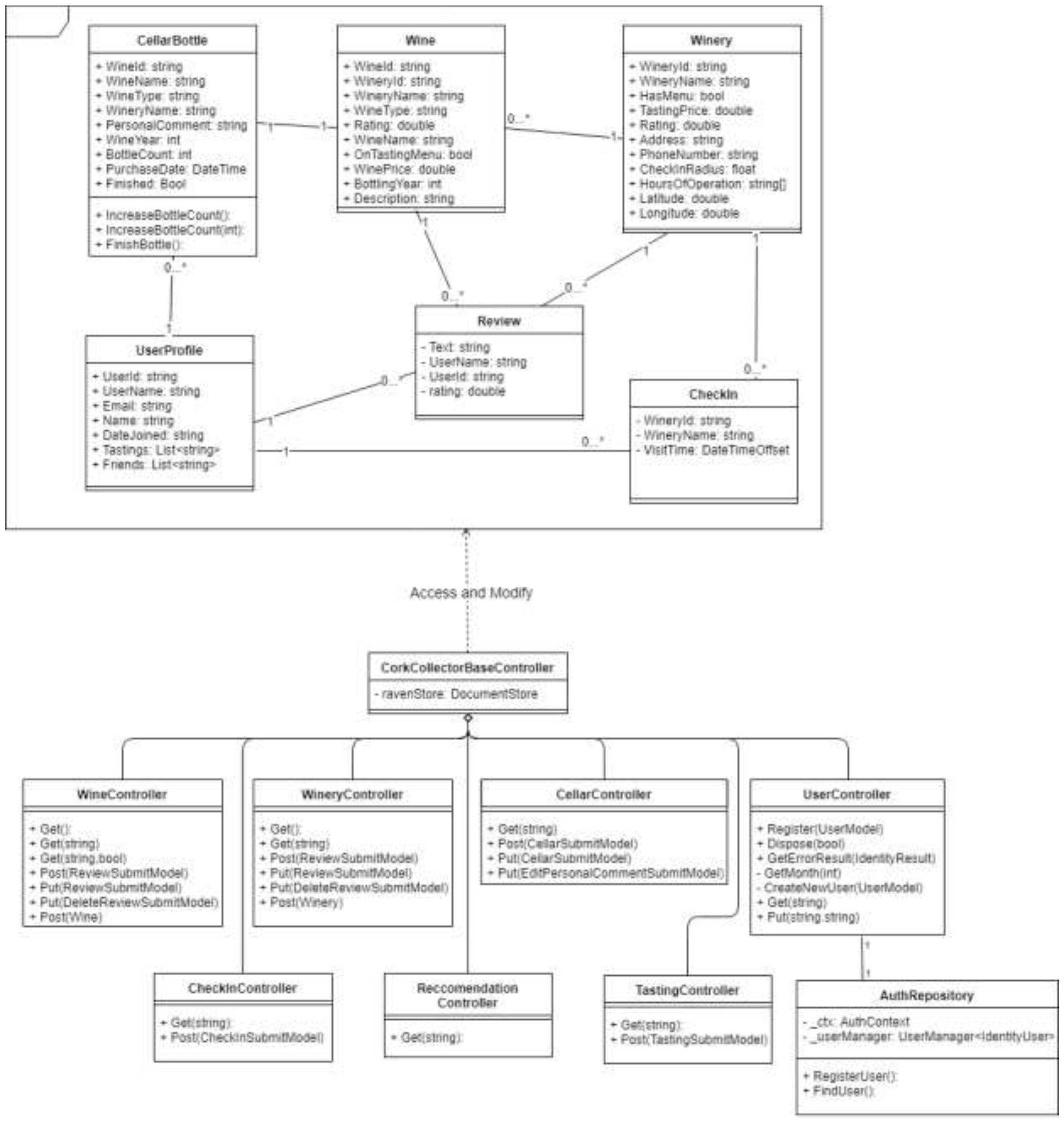
The external interface package contains external resources used by the application.

2.2.5.1 Google Maps API

The google maps API provides location data to be used by the maps view.

3. Detailed Description of Components

3.1 Class Diagram



3.1.1 UserProfile

3.1.1.1 UserProfile Attributes

Name	Type	Description
UserId	string	An ID component used to identify each user in the database.
UserName	string	The user's preferred name, provided by them upon registration. Used to login to the system.
Email	string	The user's email address, provided by them upon registration.
Name	string	The users chosen name.
DateJoined	string	A string representation of the date the user joined.
Tastings	List<string>	A list of wines the user has tasted, stored as wine id's.
Friends	List<string>	A list of friends the user has, stored as string id's.

3.1.2 Review

3.1.2.1 Review Attributes

Name	Type	Description
Text	string	The text of the comment, created by a user.
UserId	string	The database id of the user who left the review.
UserName	string	The display name for the user who left the review.
Rating	double	The rating selected by the user.

3.1.3 CellarBottle

3.1.3.1 CellarBottle Attributes

Name	Type	Description
WineId	string	An ID component used to identify the wine of each bottle in the database.
WineName	string	The name of the wine.
WineType	string	Basic wine type tagging (ie. Shiraz, Cabernet Franc, etc.)

WineryName	string	Name of winery bottle is from for display purposes.
PersonalComment	string	Users personal thoughts on the wine
WineYear	int	Year wine was bottled in.
BottleCount	int	Number of these bottles in the users cellar.
PurchaseDate	DateTime	The date and time the user purchased the wine.
Finished	bool	If the count of bottles in the cellar reaches zero, the bottle class item is considered finished (true)

3.1.3.2 CellarBottle Methods

Name	**Type**	**Description**
IncreaseBottleCount()	void	Increases the BottleCount attribute by one and sets finished to false if it is true.
IncreaseBottleCount(int)	void	Increases the BottleCount attribute by int passed and sets finished to false if it is true.
FinishBottle()	void	Decreases the BottleCount attribute by one and sets finished to true if bottle count is now zero..

3.1.4 Wine

3.1.4.1 Wine Attributes

Name	**Type**	**Description**
WineId	string	An ID component used to identify each wine in the database.
WineryId	string	The database id of the winery that produced the wine.
WineryName	string	The name of the winery that produced the wine.
WineType	string	The basic type of the wine (ie. Shiraz, Cabernet Franc, Pinot Grigio, etc.)
Rating	double	The average rating of the wine across all user reviews.
WineName	string	The name of the wine.
OnTastingMenu	bool	Tells us whether the wine is available on the wineries tasting menu.
WinePrice	double	The price for a bottle.

BottlingYear	int	The year the wine was bottled.
Description	string	A summary of the wine, flavours, pairings, etc.

3.1.5 Winery

3.1.5.1 Winery Attributes

Name	**Type**	**Description**
WineryId	string	An ID component used to identify each winery in the database.
WineryName	string	The name of the winery.
HasMenu	bool	The phone number of the winery.
TastingPrice	double	The price for standard tasting.
Rating	double	The average rating over all reviews.
Address	string	The winery user friendly address.
PhoneNumber	string	The winery phone number.
CheckInRadius	float	The max distance a user can be to check into a winery.
HoursOfOperation	string[]	Array size of 7 specifies the range of time when the winery is open each day of the week.
Latitude	double	The geographic latitude of the winery.
Longitude	double	The geographic longitude of the winery.

3.1.6 CheckIn

3.1.6.1 Checkin Attributes

Name	**Type**	**Description**
WineryId	string	An ID component used to identify each checkin in the database.
WineryName	string	The winery name for display purposes.
VisitTime	DateTime	The day and time of users check in.

3.1.7 CorkCollectorBaseController

3.1.7.1 CorkCollectorBaseController Attributes

Name	Type	Description
ravenStore	DocumentStore	A ravenDB object that is used for connecting to and querying the database.

3.1.8 UserController

3.1.8.1 UserController Methods

Name	Type	Description
Register (UserModel)	Task	Handles new user registration. UserModel contains basic username, password and email info that is used to generate user account info that will be stored in the database. Calls AuthRepository RegisterUser to add user to sql database and CreateNewUser to create the profile in RavenDB.
GetErrorResult (IdentityResult)	IHttpActionResult	Analyses basic error causes and returns result.
GetMonth(int)	string	Takes the month as integer input and returns it as string text (ie GetMonth(5) = “May”)
CreateNewUser (UserModel)	void	Creates a UserProfile object and stores it in the RavenDB.
Get(string)	UserProfile	Gets the UserProfile with the id that matches the input string from the RavenDB
Put(string,string)	HttpResponseMessage	Adds a new friend to users friend list. The first string is the user to update id, the second is the friend’s id.

3.1.9 AuthRepository

3.1.9.1 AuthRepository

Name	Type	Description
_ctx	AuthContext	Context used to access SQL database.
_userManager	UserManager <IdentityUser>	Identity object used to access built in user management methods.

3.1.9.2 AuthRepository Methods

Name	Type	Description
RegisterUser (UserModel)	IdentityResult	Asynchronously creates a new user in the SQL database.
FindUser (string,string)	IdentityUser	Finds and returns the user stored in the SQL database with standard user info.

3.1.10 WineryController

3.1.10.1 WineryController Methods

Name	Type	Description
Get()	List<Winery>	Returns a list containing all currently active wineries in the database.
Get(string)	Winery	Returns the winery whose id matches the string input or null.
Post (ReviewSubmitModel)	HttpResponse Message	Generates and stores a review by a user on a winery. User, winery, review text and rating are specified in submit model.
Put (ReviewSubmitModel)	HttpResponse Message	Edits the existing review made by a user and returns a response code.
Put (DeleteReviewSubmit Model)	HttpResponse Message	Deletes the user review and returns a response code.
Post(Winery)	HttpResponse Message	Stores a new Winery in the database.

3.1.11 WineController

3.1.11.1 WineController Methods

Name	Type	Description
Get()	List<Wine>	Gets a list of all wines in the database.
Get(string)	Wine	Gets the wine associated with the submitted string id.
Get(string,bool)	List<Wine>	Gets a list of all wines from the wnery specified by the string in the input. If the bool is set to true it only returns the wines that are on the tasting menu of the winery. Otherwise it just returns all wines the winery has had.
Post(ReviewSubmitModel)	HttpResponseMessage	Generates a new review by a user for a wine. Review text, rating, the user posting and the wine it is about are specified in the submit model.
Put(ReviewSubmitModel)	HttpResponseMessage	Modifies a review by a user for a wine. Review text, rating, the user posting and the wine it is about are specified in the submit model.
Put(DeleteReviewSubmitModel)	HttpResponseMessage	Removes a review by a user for a wine. User removing the review and the wine it is about are specified in the submit model.
Post(Wine)	HttpResponseMessage	Generates and stores a new wine in the database.

3.1.12 TastingController

3.1.12.1 TastingController Methods

Name	Type	Description
Get(string)	List<TastingListItem>	Finds the user associated with the submitted string, gets the tasting ids and queries the wines and wineries associated with them.Generates list of TastingListItem which simply reorganizes the data for better use in android app.
Post (TasteSubmitModel)	HttpResponseMessage	Adds a new tasting to the user’s profile. User and wine tasted specified in submit model.

3.1.13 CheckInController

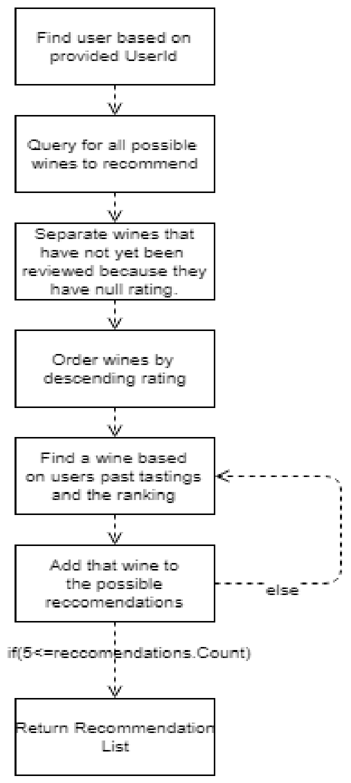
3.1.13.1 CheckInController Methods

Name	**Type**	**Description**
Get(string)	List<CheckIn>	Gets all the checkins a user with the id matching string input has.
Post (CheckInSubmitModel)	HttpResponseMessage	Generates a new checkin on the user profile of the user specified in the CheckInSubmitModel and at the winery it specifies.

3.1.14 ReccomendationController

3.1.14.1 ReccomendationController Methods

Name	**Type**	**Description**
Get(string)	List<Wine>	Generates a list of possible recommendation wines and sorts them based on rating and users previous activity. Returns the top 5 best wines.


Find user based on provided UserId
Query for all possible wines to recommend
Separate wines that have not yet been reviewed because they have null rating.
Order wines by descending rating
Find a wine based on users past tastings and the ranking
Add that wine to the possible reccomendations
else
if(5<=reccomendations.Count)
Return Recommendation List

3.1.15 CellarController

3.1.15.1 CellarController Methods

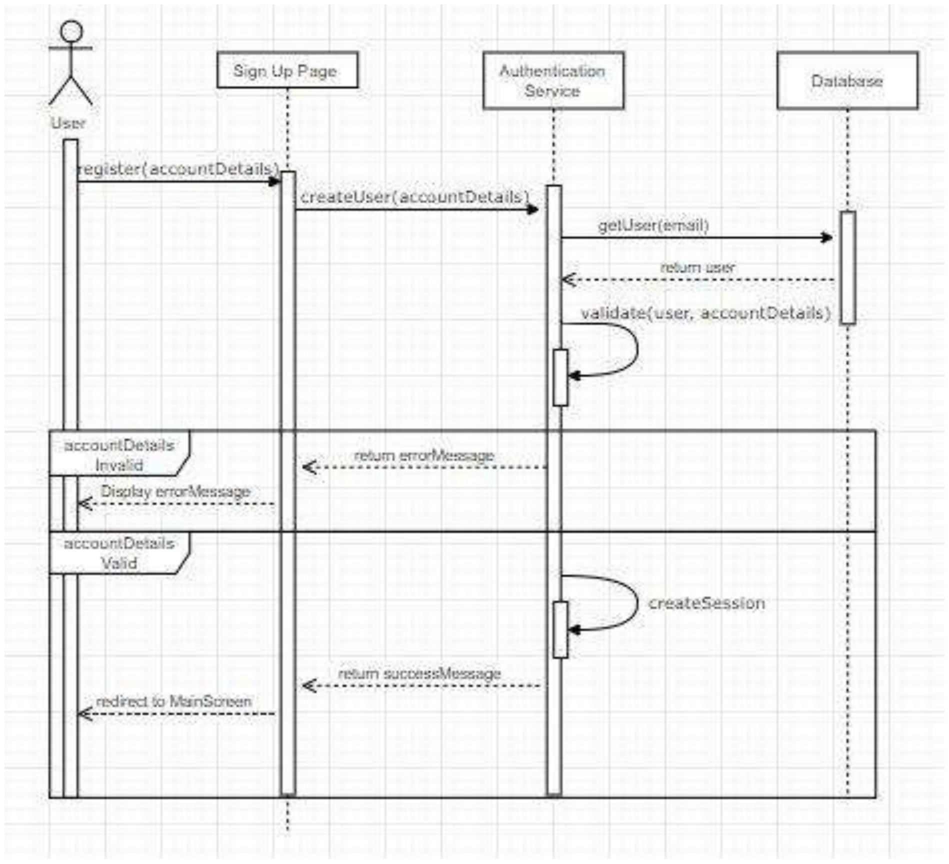
Name	**Type**	**Description**
Get(string)	List<CellarBottle>	Returns the cellar bottles in the user's cellar. User specified by id in submitted string.
Post(CellarSubmitModel)	HttpResponseMessage	Adds a new bottle to the user's cellar. If the user already has this bottle it will instead increase the bottle count. The user, wine, quantity, and notes are specified in the submit model.
Put(CellarSubmitModel)	HttpResponseMessage	Decreases the bottle count of a wine in a user's cellar. The user, wine, quantity, and notes are specified in the submit model.
Put(EditPersonalCommentSubmitModel)	HttpResponseMessage	Allows a user to edit the notes made on a wine in their cellar. User, wine and new notes are specified in the submit model.

3.2 Sequence Diagrams

The following section defines sequence diagrams for various tasks that can be performed in *CorkCollector*. It can be assumed that all user inputs will be properly sanitized before querying them against the database.

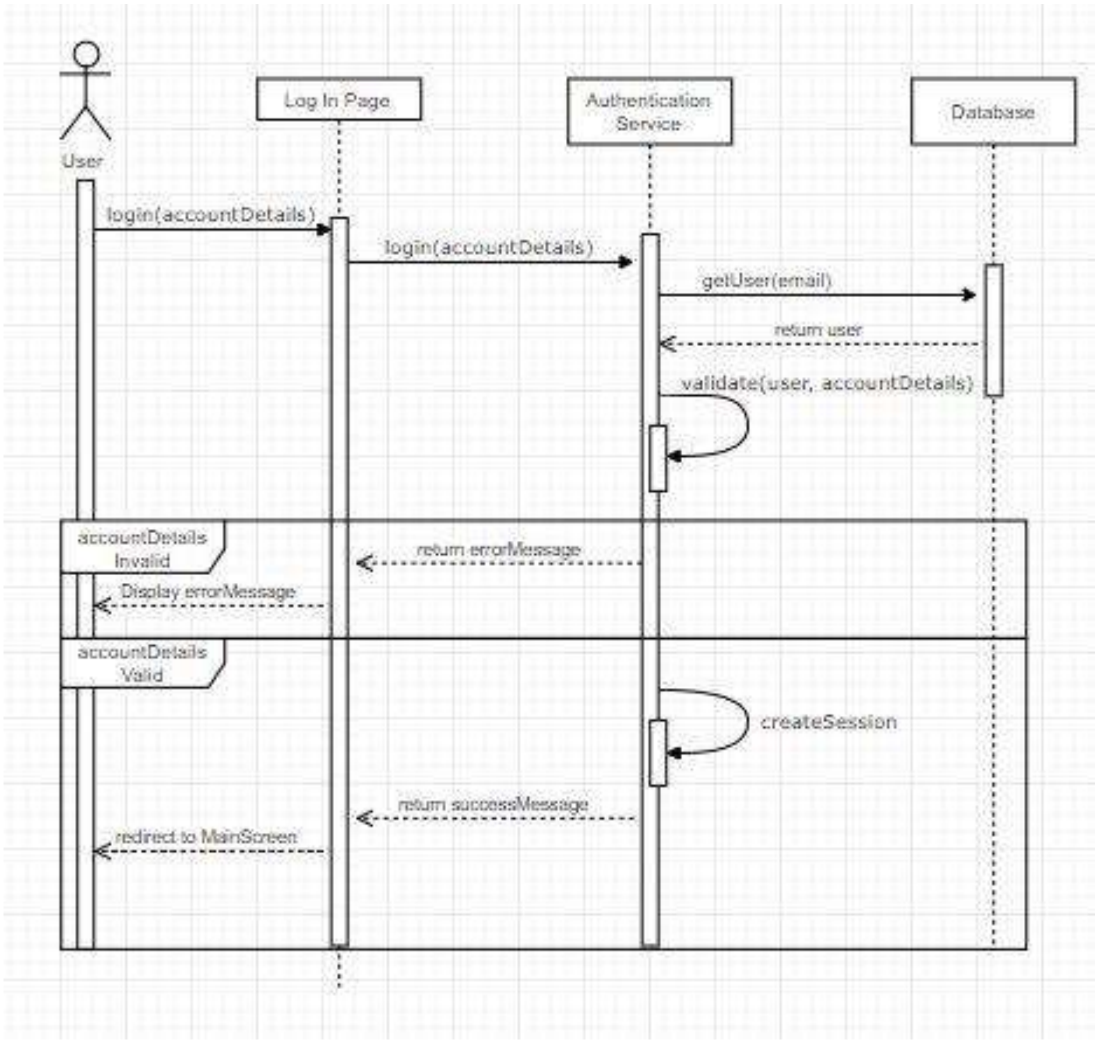
3.2.1 Sequence Diagram 1: Create User

This diagram demonstrates how a new user can be created.



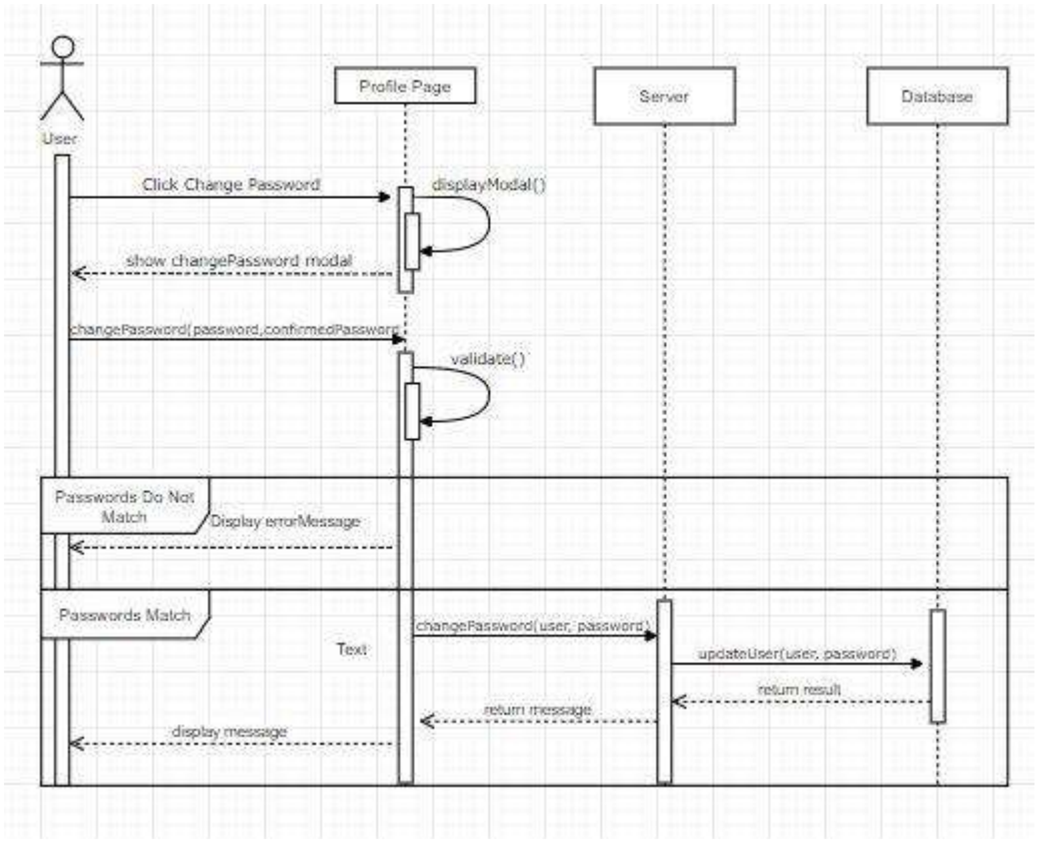
3.2.2 Sequence Diagram 2: Log In

This diagram demonstrates how a user can log into the system once they have a profile.



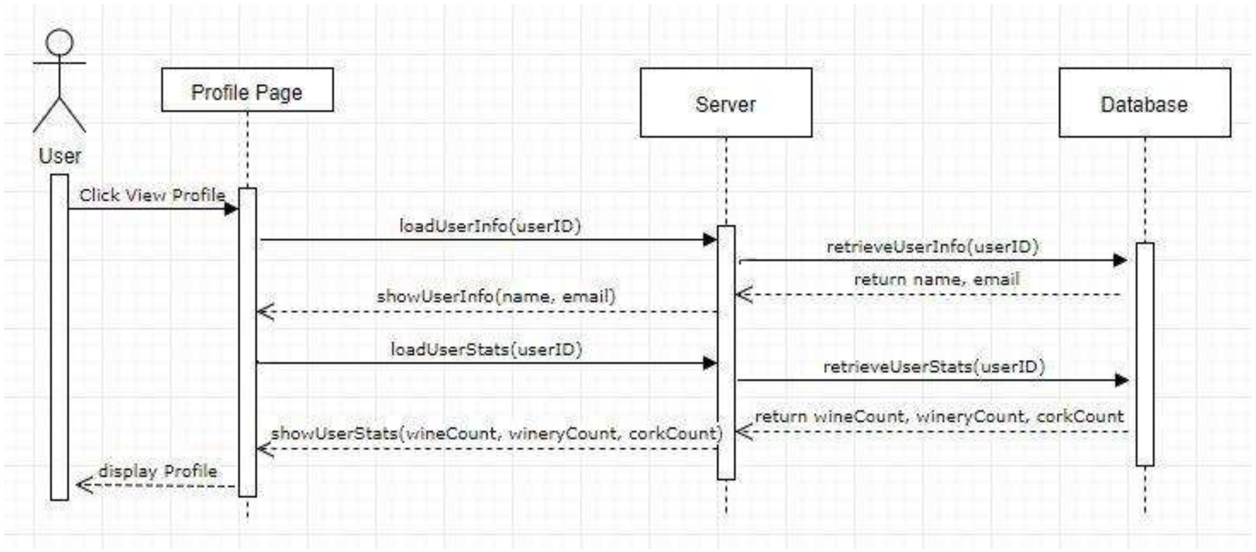
3.2.3 Sequence Diagram 3: Change Password

This diagram demonstrates how a user may change their password.



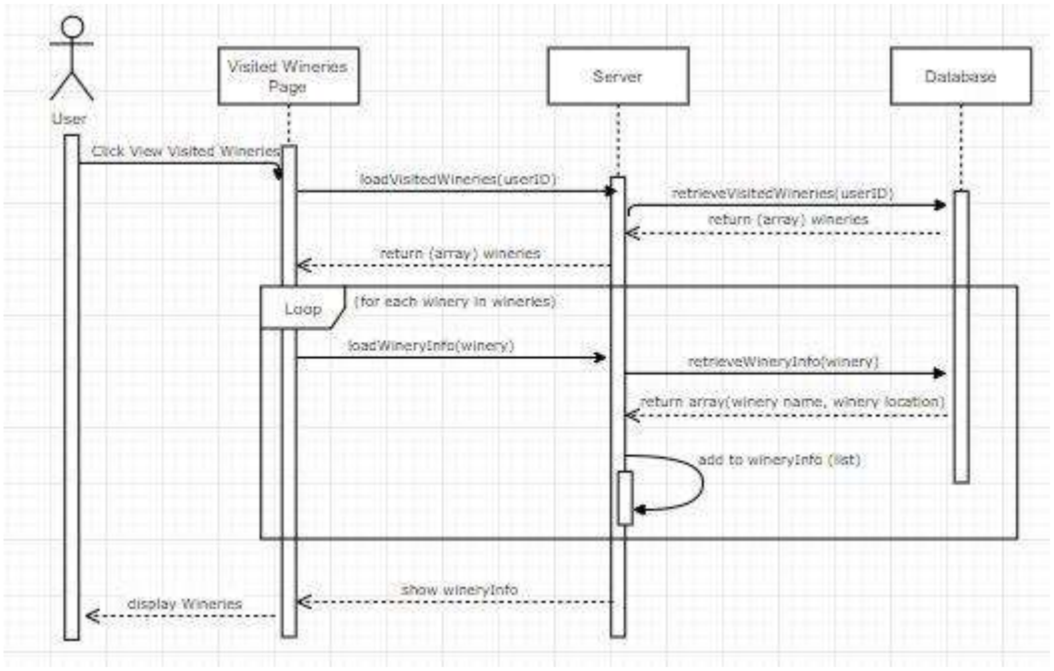
3.2.4 Sequence Diagram 4: View Profile

This diagram shows how the user may view their own profile.



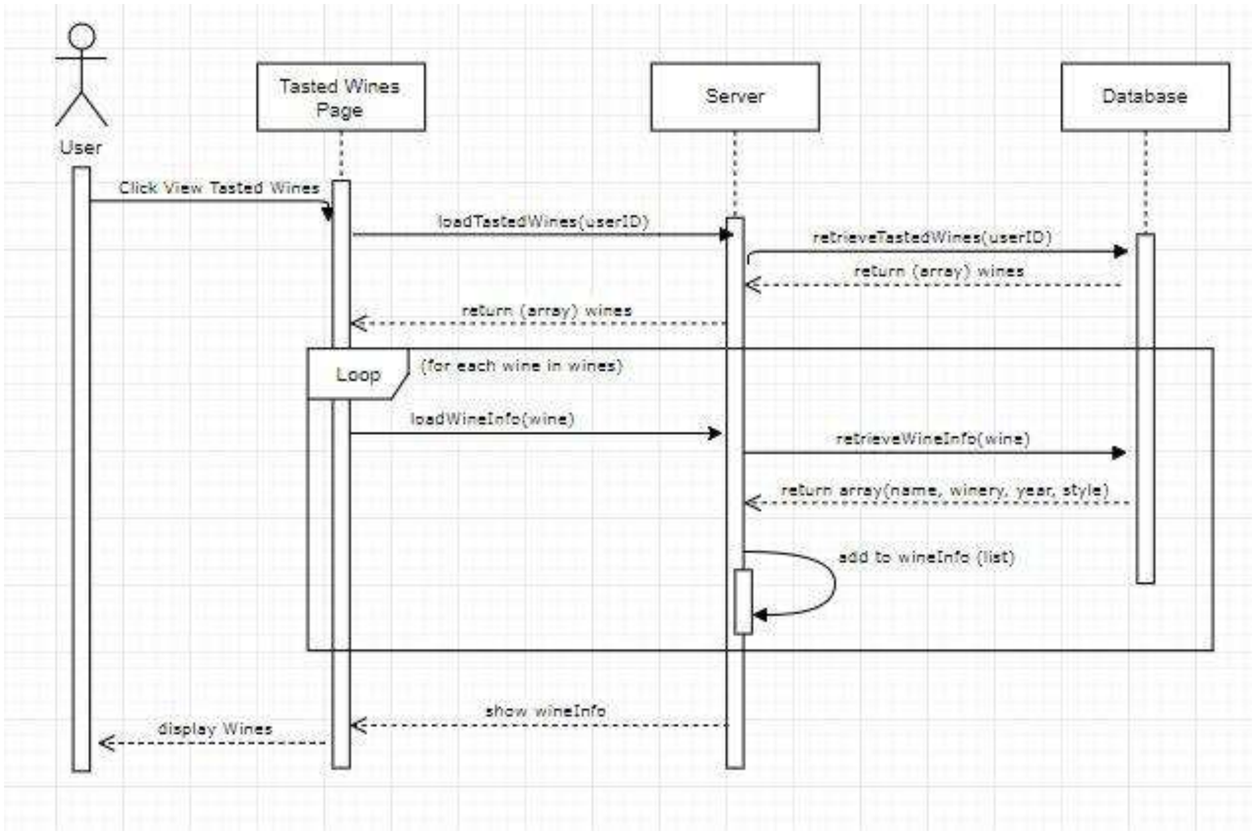
3.2.5 Sequence Diagram 5: View Visited Wineries

This diagram will show how the user may view the list of wineries that they have already visited.


User
Visited Wineries Page
Server
Database
Click View Visited Wineries
loadVisitedWineries(userID)
retrieveVisitedWineries(userID)
return (array) wineries
return (array) wineries
Loop
(for each winery in wineries)
loadWineryInfo(winery)
retrieveWineryInfo(winery)
return array(winery name, winery location)
add to wineryInfo (list)
show wineryInfo
display Wineries

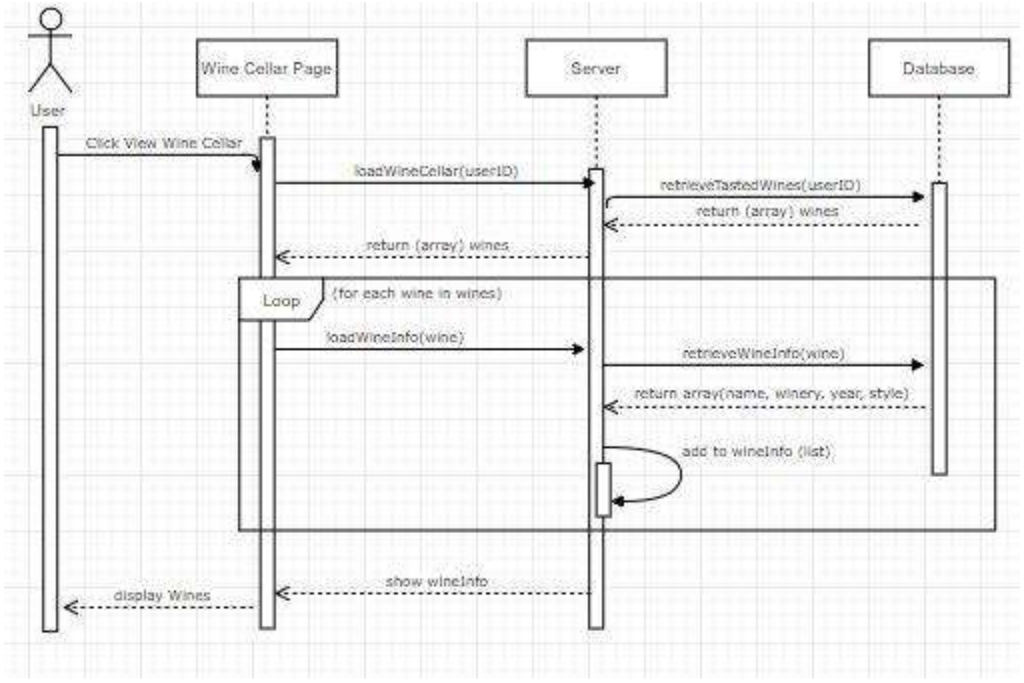
3.2.6 Sequence Diagram 6: View Tasted Wines

This diagram will show how a user can view the list of wines that they have tasted in the past.



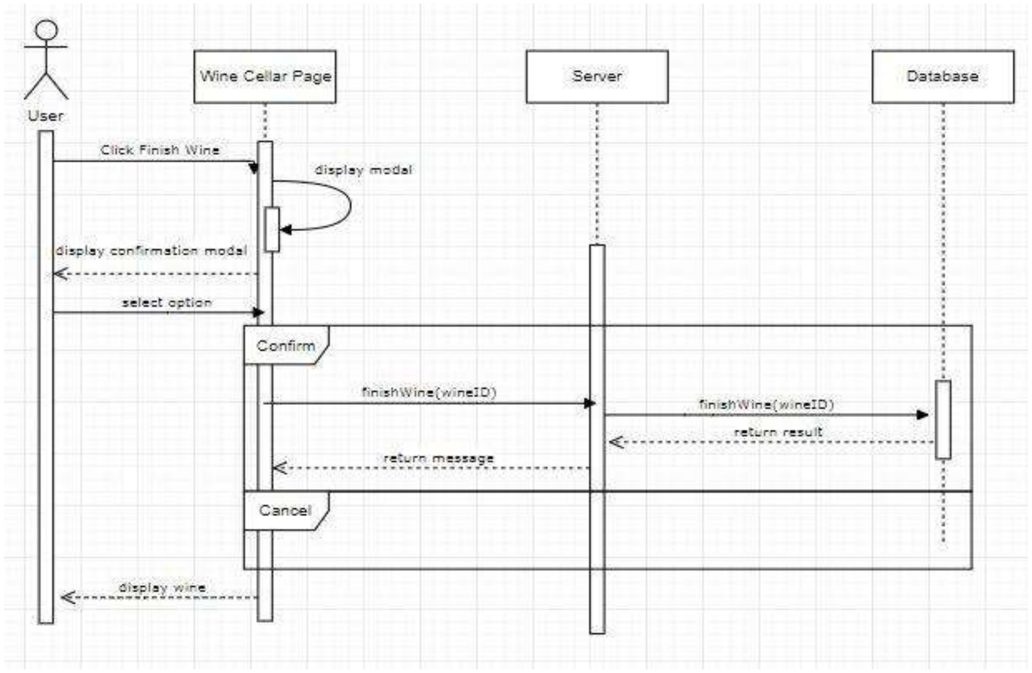
3.2.7 Sequence Diagram 7: View Wine Cellar

This diagram shows how the user may view the list of wine bottles saved in their cellar.



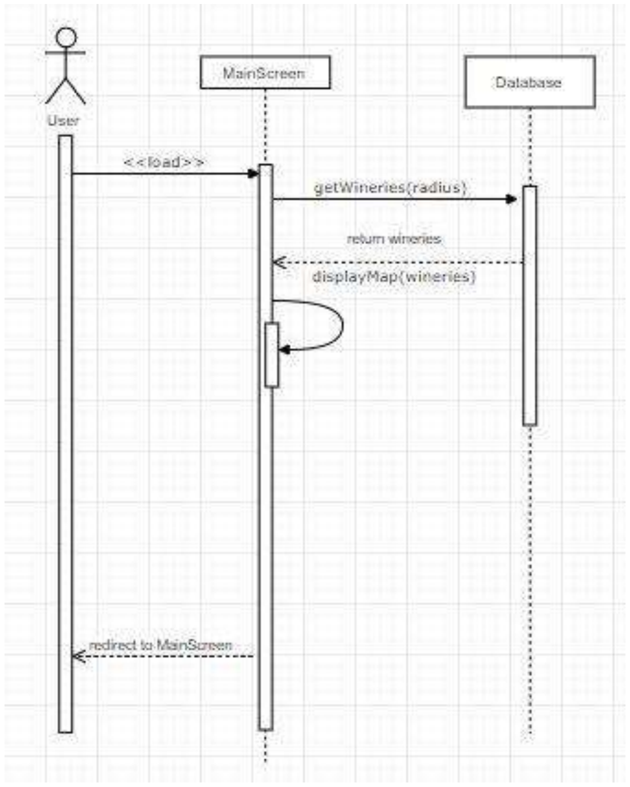
3.2.8 Sequence Diagram 8: Finish Wine Bottle

This diagram shows how the user can mark a bottle as finished when they have consumed it. This meaning it will move from their wine cellar to the corks section.



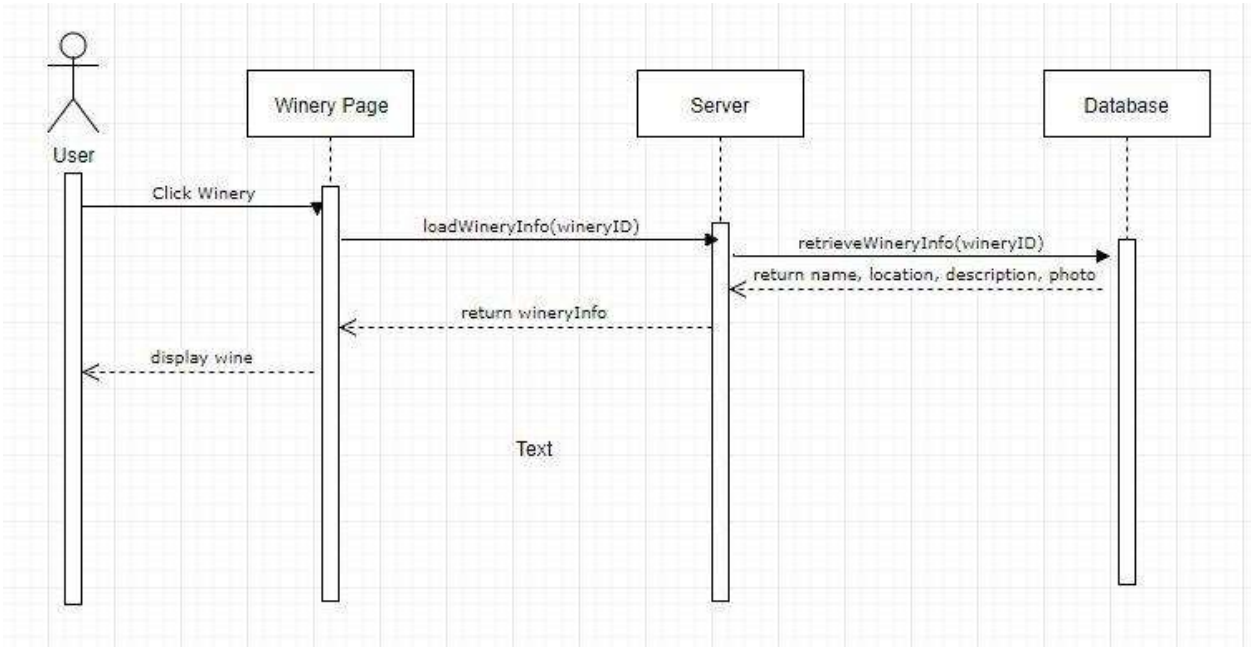
3.2.9 Sequence Diagram 9: View Map of Wineries

This diagram shows how the user may interact with the map displaying the wineries.



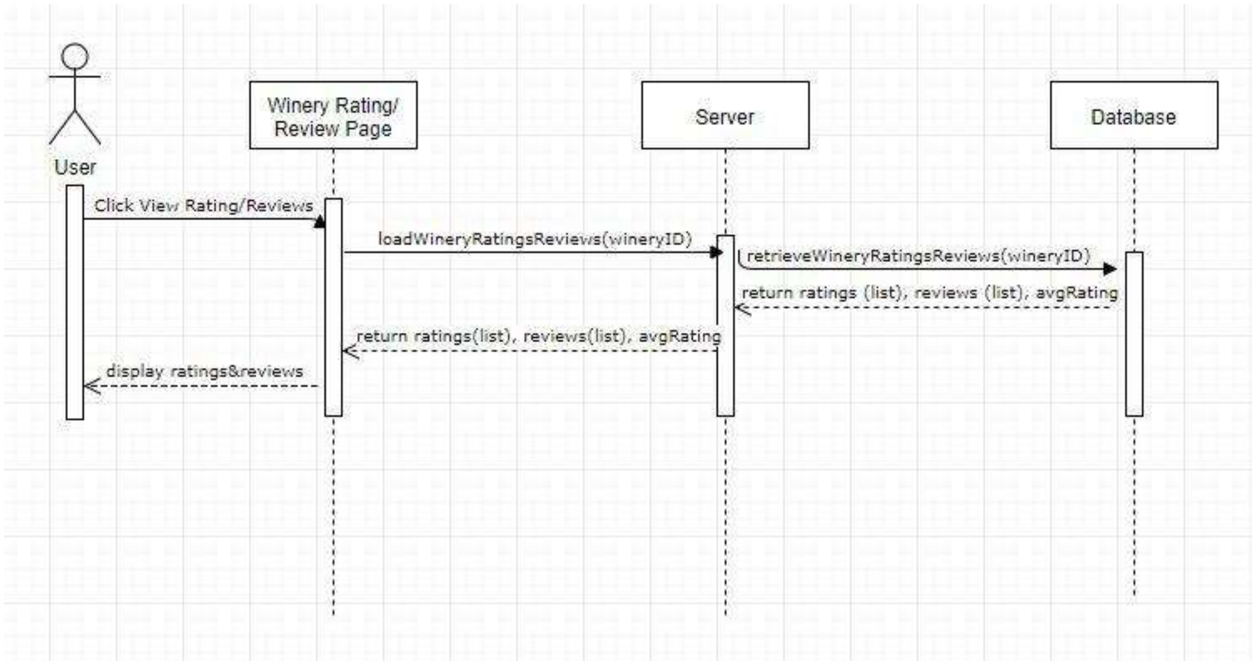
3.2.10 Sequence Diagram 10: View Winery

This diagram shows how a user can view the information about a winery.



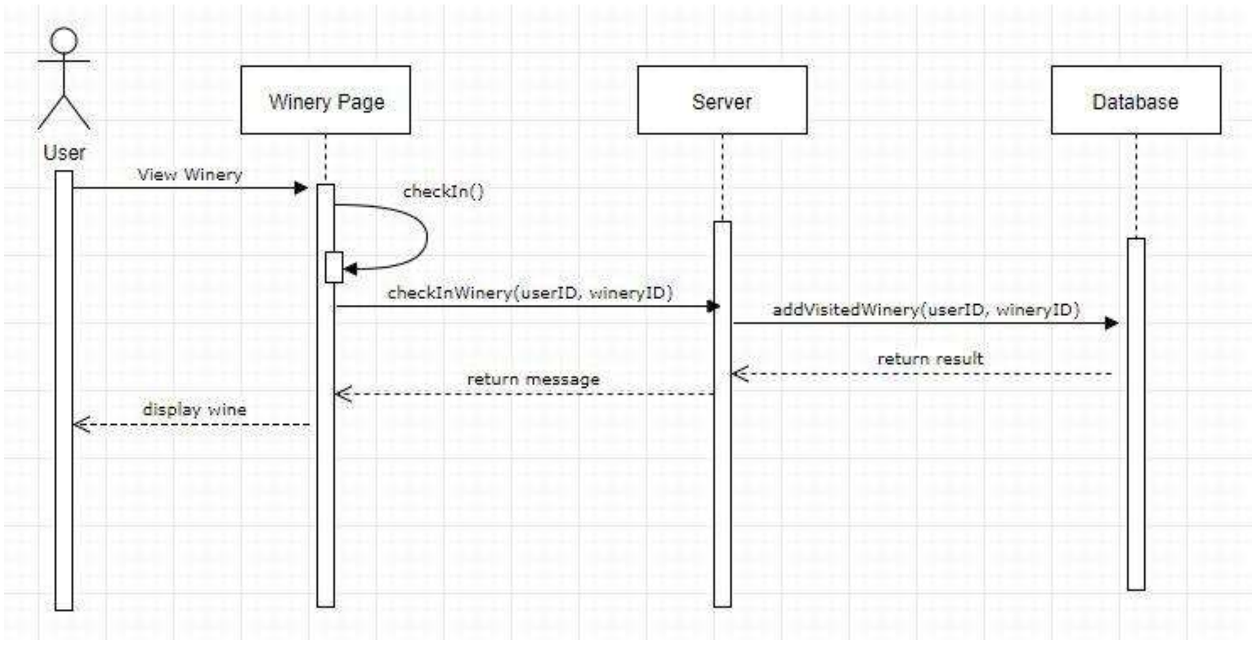
3.2.11 Sequence Diagram 11: View Winery Ratings/Reviews

This diagram shows how a user can view the ratings/reviews left about a winery.


User
Winery Rating/ Review Page
Server
Database
Click View Rating/Reviews
loadWineryRatingsReviews(wineryID)
retrieveWineryRatingsReviews(wineryID)
return ratings (list), reviews (list), avgRating
return ratings(list), reviews(list), avgRating
display ratings&reviews

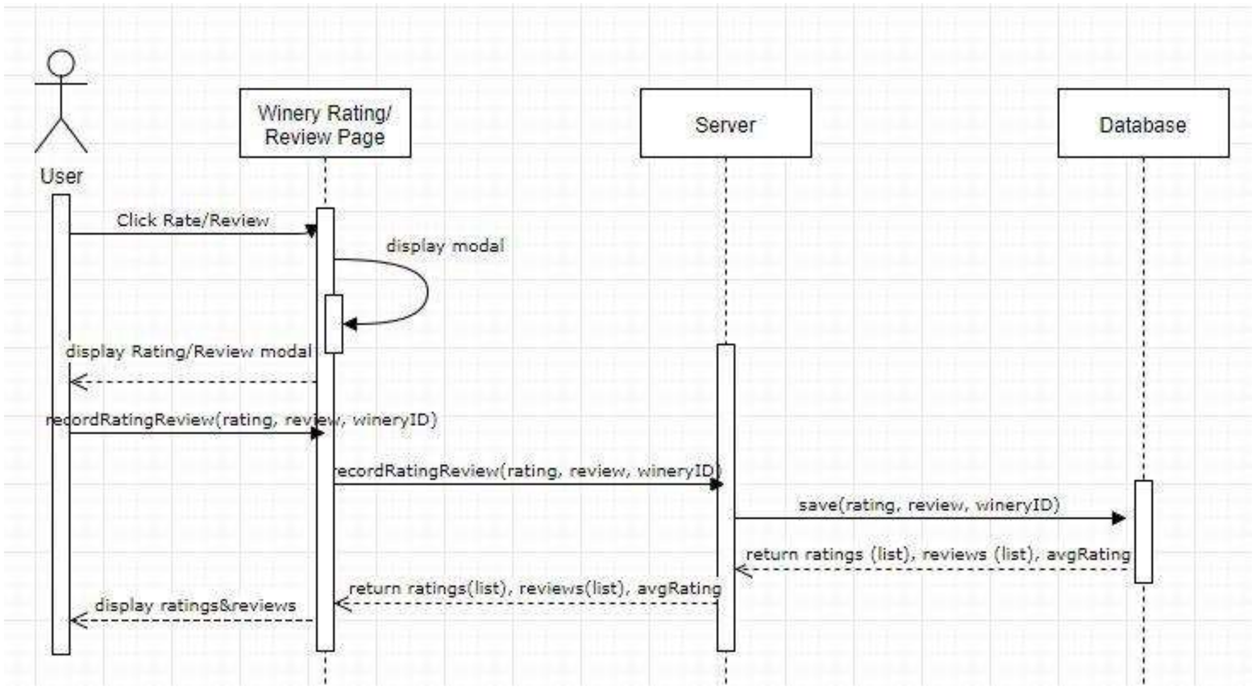
3.2.12 Sequence Diagram 12: Check-in at Winery

This diagram shows how a user may “check-in” at a winery and have it added to their list of visited wineries.



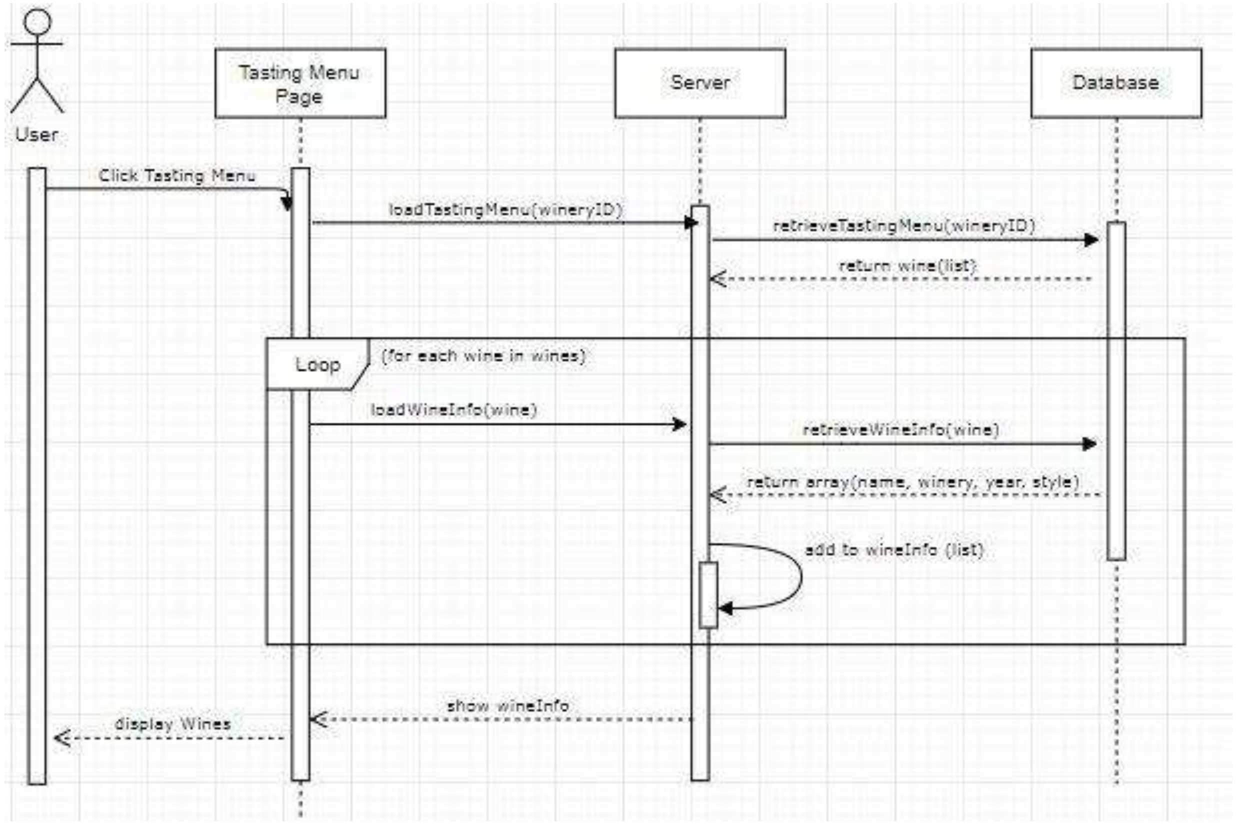
3.2.13 Sequence Diagram 13: Rate/Review a Winery

This diagram shows how a user may leave their own rating/review on a winery


User
Winery Rating/ Review Page
Server
Database
Click Rate/Review
display modal
display Rating/Review modal
recordRatingReview(rating, review, wineryID)
recordRatingReview(rating, review, wineryID)
save(rating, review, wineryID)
return ratings (list), reviews (list), avgRating
return ratings(list), reviews(list), avgRating
display ratings&reviews

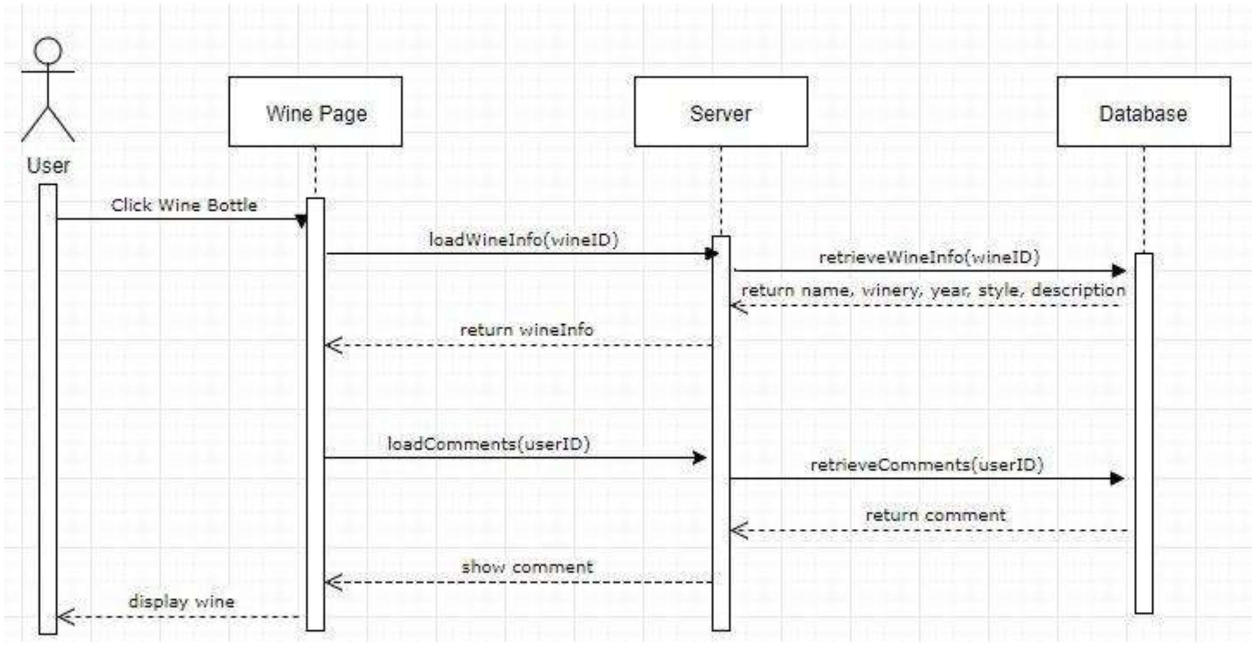
3.2.14 Sequence Diagram 14: View Tasting Menu

This diagram shows how a user may view the tasting menu for a specific wine.



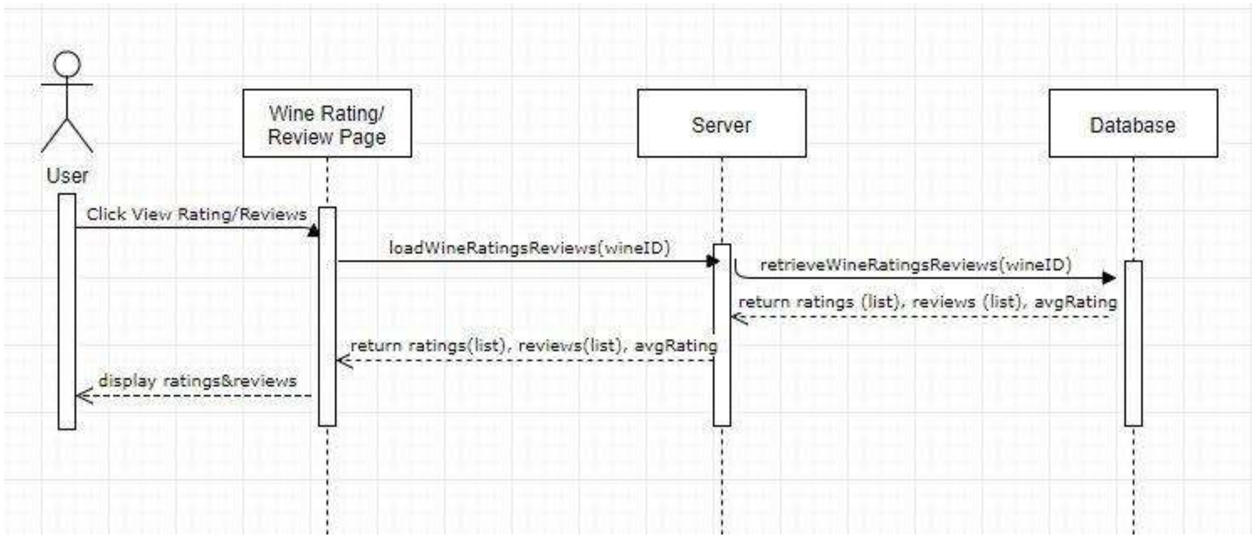
3.2.15 Sequence Diagram 15: View Wine

This diagram shows how a user may view information about a specific wine.



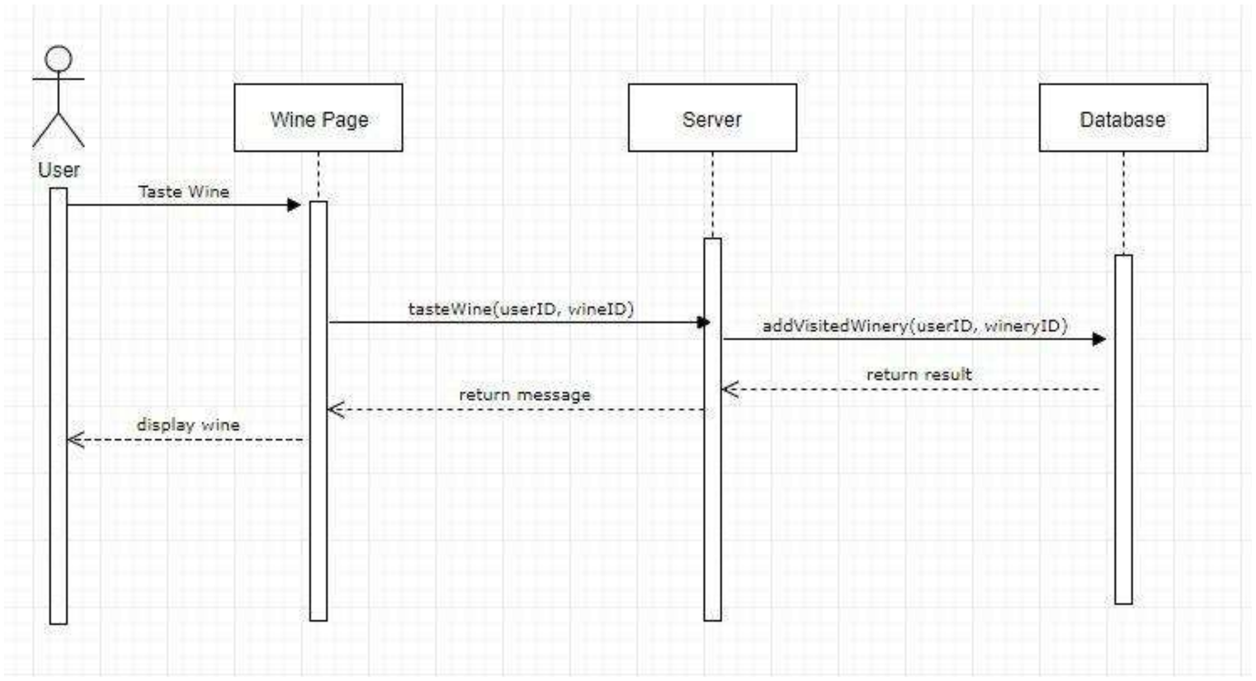
3.2.16 Sequence Diagram 16: View Wine Ratings/Reviews

This diagram shows how a user may view the ratings/reviews on a wine.


User
Wine Rating/ Review Page
Server
Database
Click View Rating/Reviews
loadWineRatingsReviews(wineID)
retrieveWineRatingsReviews(wineID)
return ratings (list), reviews (list), avgRating
return ratings(list), reviews(list), avgRating
display ratings&reviews

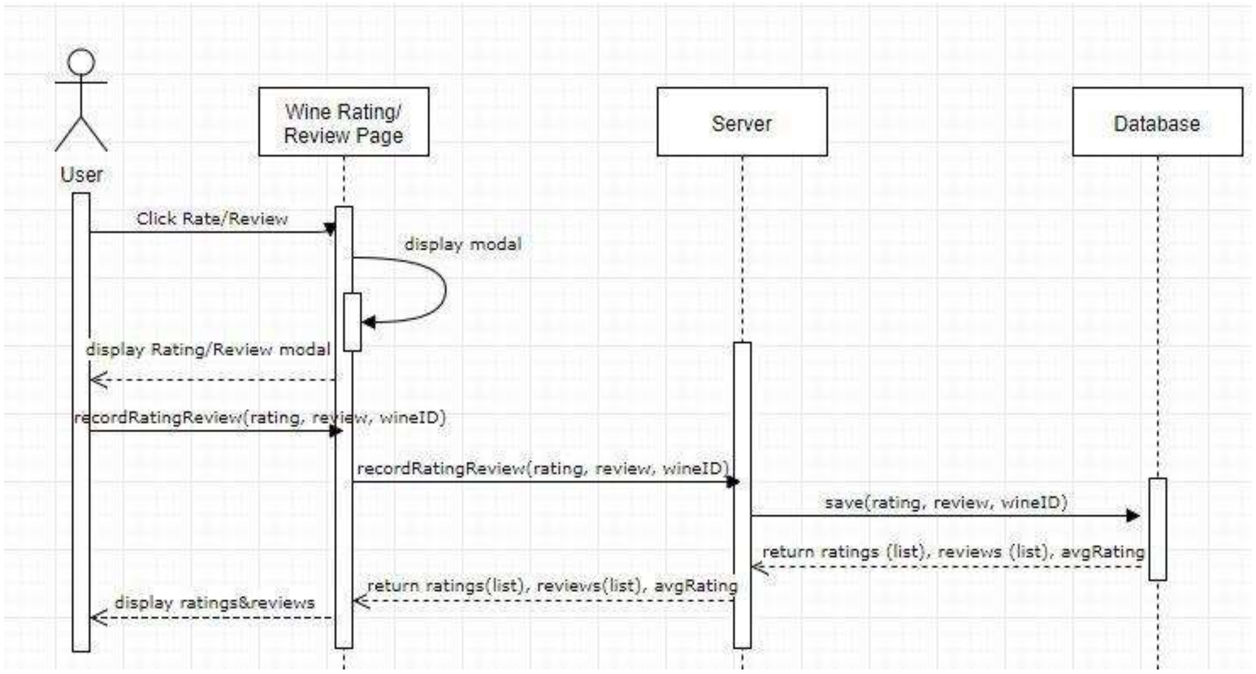
3.2.17 Sequence Diagram 17: Taste Wine

This diagram shows how a user may mark a wine as tasted and have it added to their tasted wines list.



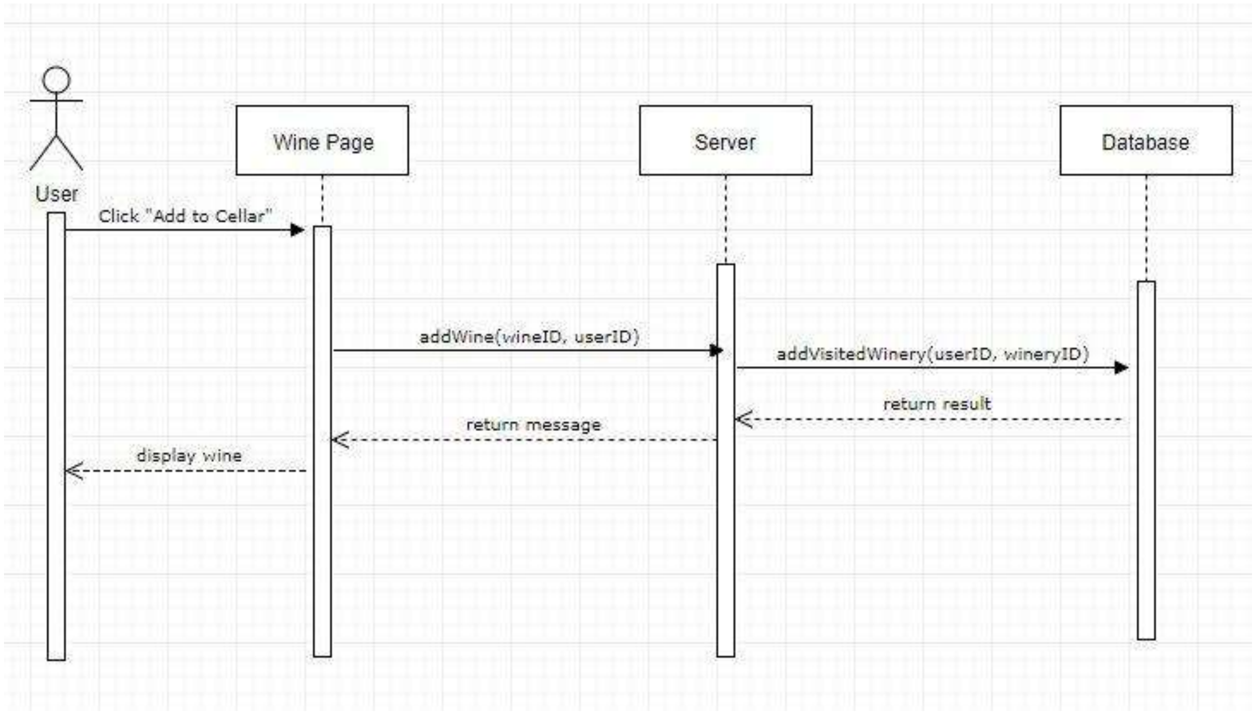
3.2.18 Sequence Diagram 18: Rate/Review a Wine

This diagram shows how a user may leave their own ratings on a wine.


User
Wine Rating/ Review Page
Server
Database
Click Rate/Review
display modal
display Rating/Review modal
recordRatingReview(rating, review, wineID)
recordRatingReview(rating, review, wineID)
save(rating, review, wineID)
return ratings (list), reviews (list), avgRating
return ratings(list), reviews(list), avgRating
display ratings&reviews

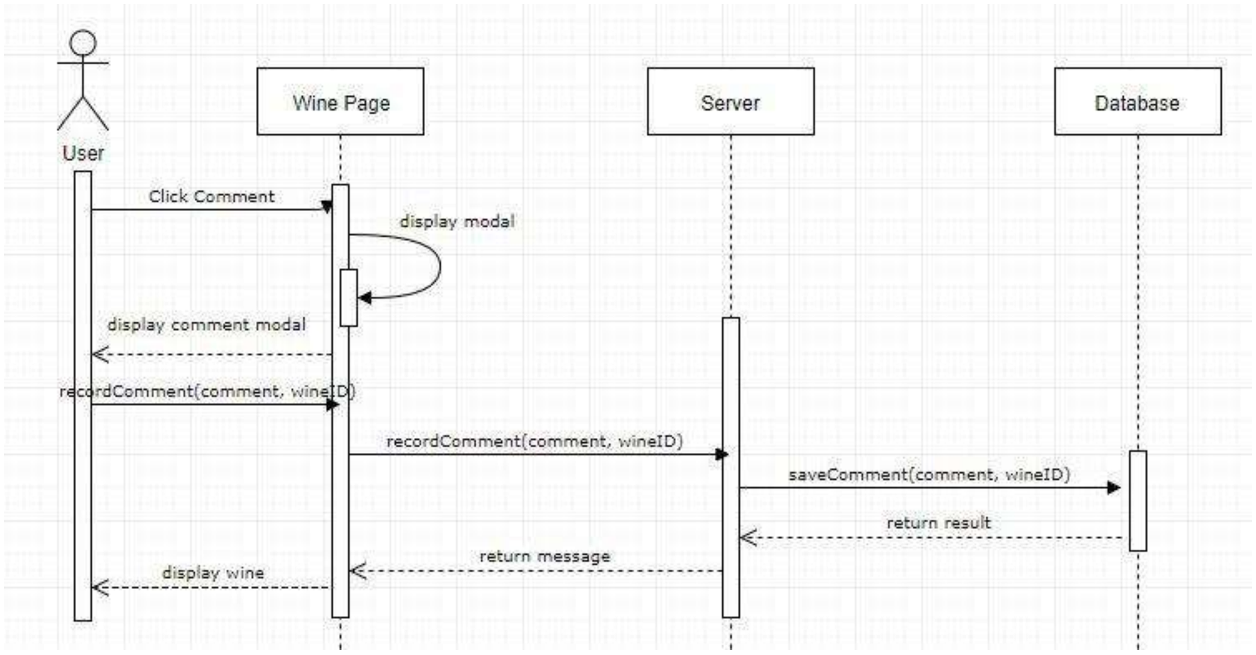
3.2.19 Sequence Diagram 19: Add Wine to Cellar

This diagram shows how a user may add a wine to their wine cellar list.



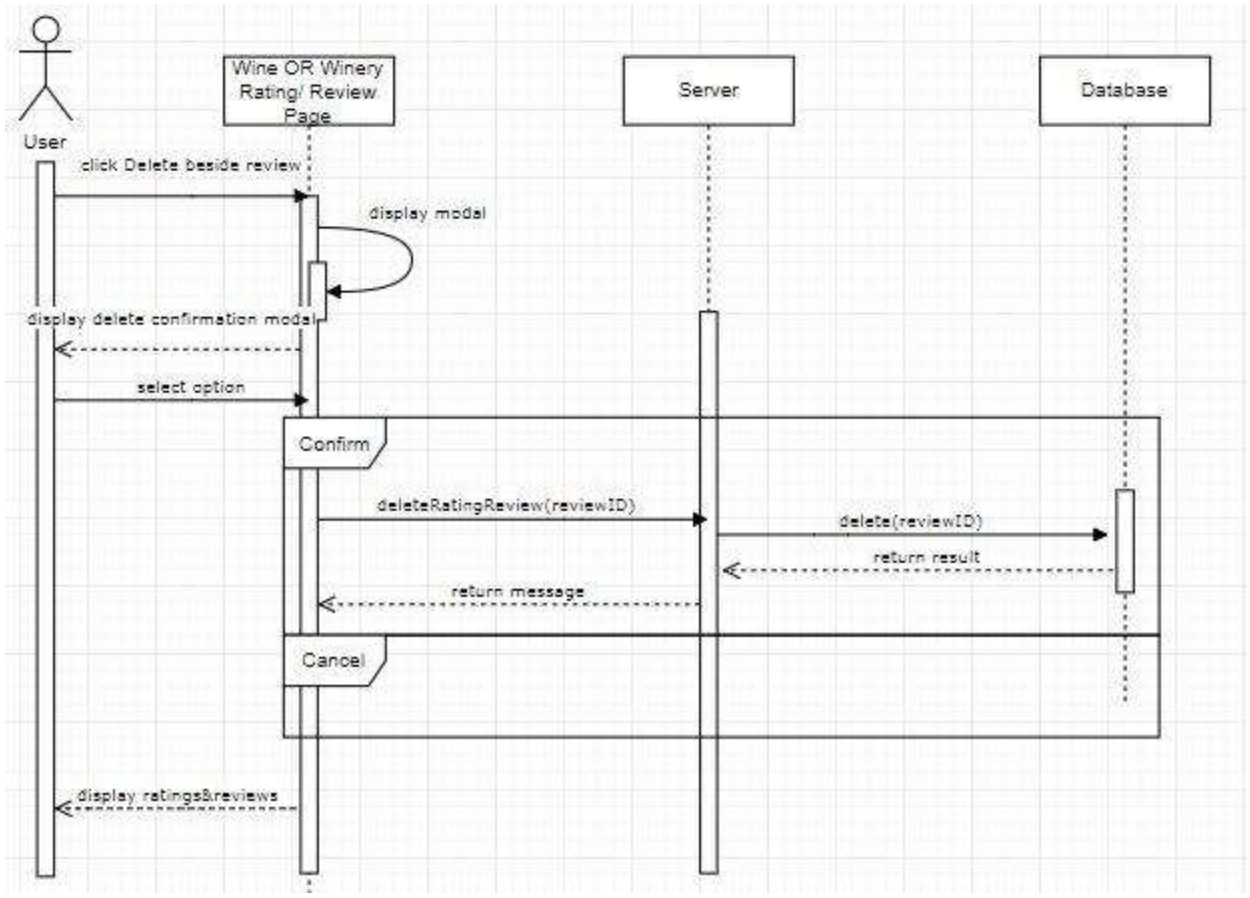
3.2.20 Sequence Diagram 20: Add a Comment

This diagram shows how a user can add a personal comment to a bottle of wine.



3.2.21 Sequence Diagram 21: Delete a Rating/Review

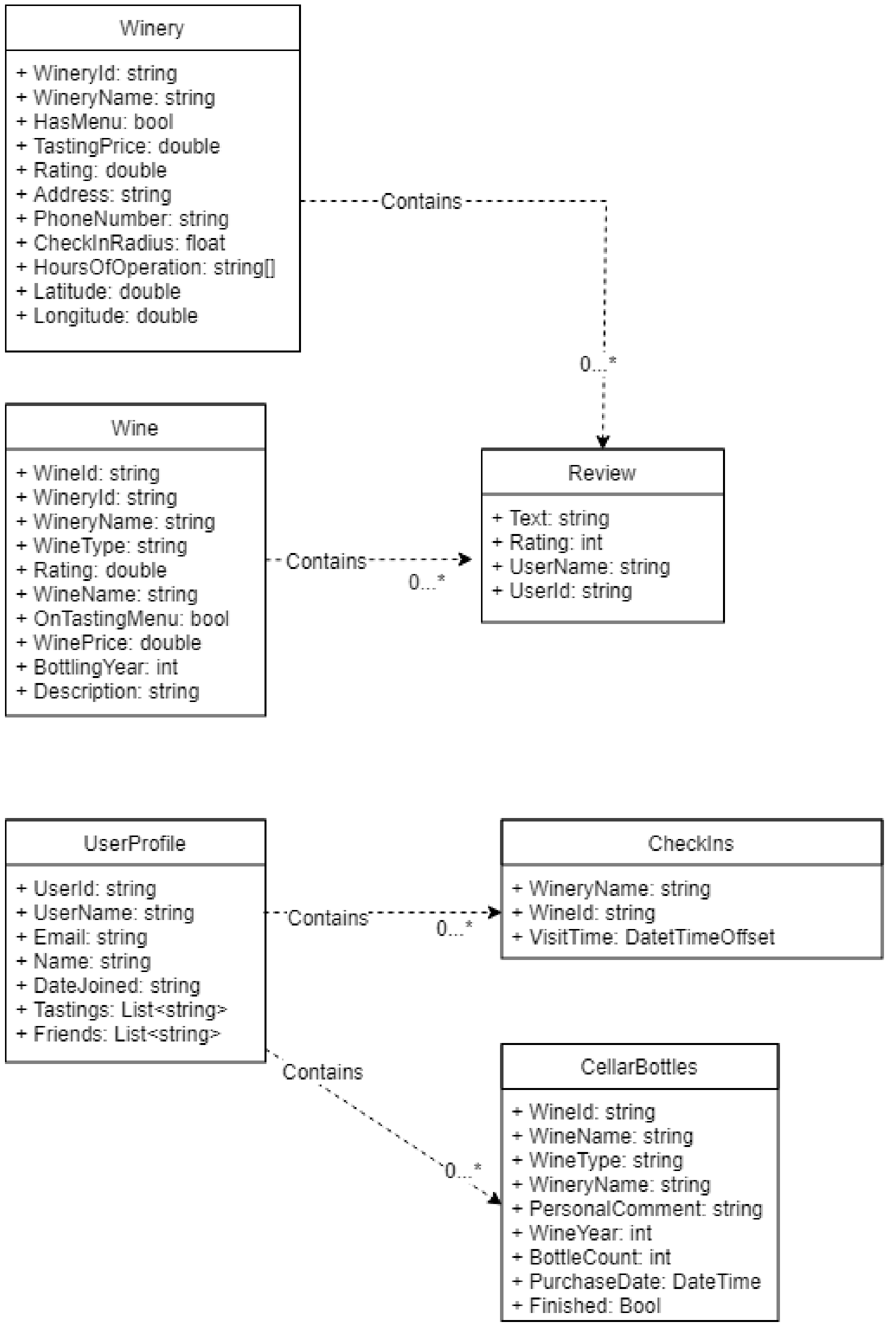
This diagram shows how a user may delete a rating/review that they have left on either a bottle of wine or a winery.


User
Wine OR Winery Rating/ Review Page
Server
Database
click Delete beside review
display modal
display delete confirmation modal
select option
Confirm
deleteRatingReview(reviewID)
delete(reviewID)
return result
return message
Cancel
display ratings&reviews

3.3 NoSQL Database Schema

3.3.1 Database Schema Diagram

CorkCollector uses a NoSQL document database rather than a relational database schema. This means a traditional relational database schema will not adequately describe the structure of our database. Instead the JSON structure below represents the nested structure we have designed for our document database. The three primary heads of this nested structure are: **wineries**, **wines** and **users**.



3.3.2 JSON Structure of Schema:

```
Winery:{
        WineryId: <string>,
        WineryName: <string>,
        Reviews:
        [
                {
                        Text: <string>,
                        Rating: <int>,
                        Username: <string>,
                        UserId: <string>
                }...],
        HasMenu: <bool>,
        TastingPrice: <double>,
        Rating: <double>,
        Address:<string>,
        PhoneNumber: <string>,
        HoursOfOperation: string[]
        CheckInRadius: <float>,
        Latitude: <double>,
        Longitude: <double>,
}
Wine:{
        WineId: <string>,
        WineryId: <Guid>,
        WineName: <string>,
        WineryName: <string>,
        WineType: <string>,
        OnTastingMenu: <bool>,
        BottlePrice: <double>,
        BottlingYear: <int>,
        Description: <string>,
        Reviews:[
                {
                        Text: <string>,
                        Rating: <int>,
                        Username: <string>,
                        UserId: <string>
                }...],
}
```

```
UserProfile:{
        UserId: <Guid>,
        UserName: <string>,
        Name: <string>,
        Email: <string>,
        DateJoined: <string>,
        Friends: <List<string>> //the stringsare user ids,
        Tastings: <List<string>> //the strings are wine ids,
        CheckIns:[
                {
                        WineryName: <string>,
                        WineryId: <string>,
                        VisitTime: <DateTimeOffset>
                }...
        ]
        CellarBottles: [
                {
                        WineId: <string>,
                        WineName: <string>,
                        WineType: <string>,
                        WineryName: <string>,
                        WineYear: <int>,
                        PurchaseDate: <DateTime>,
                        PersonalComment: <string>,
                        BottleCount: <int>,
                        Finished: <bool>,
                }...
        ]
}
```

4. Design Rationale

The design of *CorkCollector* involved numerous design decisions which will affect the overall effort of implementation and the features that will be available for the application. These decisions included based application infrastructure, database infrastructure, implementation platforms and hosting services. This section will explain the reasoning for the decisions we've made thus far.

The overall design for *CorkCollector* attempts to follow a modular approach and keep the data access and manipulation separate from the user interfacing component. To achieve this *CorkCollector* will use a REST ASP.Net web API to access and modify the data from our database. The front end of the application will communicate with the API through http requests and responses passing JSON objects between the two. This will allow future front end implementations (such as iOS or a website) to get the necessary data without modifying the data access components. It will also provide higher security by ensuring users only have access to API calls they are authorized for. C# was chosen for the implementation language because it is a language the entire team has experience in and it has many libraries and plugins available to help us with authorization functionality, JSON object manipulation, and more. Node.JS also had many of these abilities but do to the teams higher level of experience in C# and fairly similar performance abilities, C# was chosen.

For our database implementation we had to decide between implementing a relational database schema or a non-relational database schema. For this situation we chose to go with a NoSQL document database (non-relational) which we intend to implement using RavenDB. We decided on NoSQL primarily due to the structure of our data and the higher scalability of NoSQL. First, our application is designed in such away that our data will grow exponentially as new wines and wineries are created. This means that the scalability of the database is a huge factor to consider for us. Plus, since we are regularly accessing the same type of data on our pages, we can easily structure the data for quick access and querying by our pages, and RavenDB allows the creation of custom indexes which can be used to improve the efficiency of these queries. Furthermore, creating the database in NoSQL will allow for easy changes to the schema as necessary. This will be useful for us when we intend to add new features that would involve adding new information to existing data objects. RavenDB also integrates well with C# as it has numerous plugins designed for data access and modification.

Let’s Encrypt certificates will be used to ensure security on our web API and database server. These certificates are free and automatically renew based on a custom job which will minimize cost while still providing effective security for our application. The application itself is being hosted using AWS. AWS is a well known web hosting framework that has reliable services for hosting our application components. Furthermore, the platform allows us to scale the resources based on the traffic that occurs. AWS also has a free tier of pricing for the first year that we intend to make use of for the development of the application.

Our first iteration of the application will be written with an Android application front end. We chose android because it has a large user base to target with our application. Specifically we are targeting Android versions 4+ as this will ensure the application is available to almost all Android users. Once we have created a successful Android version we can then focus on moving the front end to iOS and, since the data is accessed through a separate web API, we will only need to focus on creating iOS designed front end views.

Cork collector will be leveraging the Google Maps API for our location based services. The API will allow us to quickly implement a Map UI on the main screen, display the locations of the nearby wineries on the map interface and generate accurate directions for the user from location addresses. This would have been a huge undertaking to code from scratch but by making use of the existing Google Maps services we can implement them quickly and effectively.

4.1 Satisfying Non-Functional Requirements

The development team’s choice of tools to build *CorkCollector* reflects our goals to efficiently meet the goals we have placed for our project. They will be laid out below to demonstrate how our application will satisfy the requirements we placed during its inception.

4.1.1 Satisfying Usability

Android Studio greatly lends itself to application design with user-friendly tools. A clear page header is presented on every screen, so the user is always aware of how they care using the application, and their available functions are always accessible through an omni-present toolbar. Page-specific functions are provided through clear and obvious button prompts, and popup screens always clearly label their purpose. As is standard for Android applications, the user can revert to the previous screen with the touch of a button, incurring no data loss in the process. The manner in which pages and activities are loaded in *CorkCollector* also ensures that key data and variables are never lost, regardless of how the user traverses the pages of the application. Finally, the use of the Google Maps API provides a clear and intuitive interface for the key functionality of the application, which leverages the intuitive use of a touchscreen.

4.1.2 Satisfying Reliability & Performance

As mentioned in the previous section, the *CorkCollector* runs on a NoSQL database. Not only does this ensure connections with the application are secure and fast, but it also reliably scales with the application to accommodate information and timing targets as the user base of the application grows. The server has run reliably, responding to multiple requests in a timely manner, and can be consistently updated without error to extend the scope and parameters of the database. The frontend of the application handles requests with Volley, a standard, reliable networking library that allows it to make secure requests and execute important functions immediately after they have been received - further reducing latency and application delays. Finally, while network speeds and other details are not under our control, it is evident through multiple demonstrations and usages that the application meets speed requirements, likely as a result of the flexible, savvy and scalable nature of its design and construction.

4.1.3 Satisfying Supportability

The bulk of application construction is complete for *CorkCollector*, but its supportability targets are still a crucial part of the maintenance and improvement process. Various changes have been made from the original design to support more user-friendly tasks (eg. deleting and editing reviews or notes), and have been implemented very quickly due to the modular and supportive nature of the application as a whole. Furthermore, during the main cycles of development, the structure and functionality of many frontend classes and pages were re-used with slight modification due to their very simple nature (eg. Wine and Winery pages) - thus saving countless hours in development time and allowing for more testing before and after integration.

CorkCollector

Test Plan

Date

Russell Stirling	250757946
Daniel Lorencez	250741921
Bailey Hanna	250718425

Revision History

Date	Version	Description	Author
2017-11-13	1.0	Initial document section headers created.	Bailey Hanna
2017-11-17	1.1	First draft of individual sections.	Bailey Hanna Daniel Lorencez Russell Stirling
2017-11-20	1.2	First draft of individual sections.	Bailey Hanna Daniel Lorencez Russell Stirling
2017-11-22	1.3	Editing and revision of different sections.	Bailey Hanna Russell Stirling
2017-11-27	1.4	Final Draft of all sections.	Bailey Hanna Daniel Lorencez
2017-11-29	1.5	Document formatting.	Bailey Hanna
2018-04-01	2.0	Revision of sections based on midterm report feedback.	Bailey Hanna Daniel Lorencez Russell Stirling
2018-04-03	2.1	Updates based on testing done during development.	Bailey Hanna
2018-04-05	2.2	Document edit and formatting.	Bailey Hanna

Executive Summary

Objective

The objective of this document is to define a testing plan for the initial release of *CorkCollector*. This document will analyze the methods of testing in order to prepare the development team for the required testing efforts needed to ensure the quality of the product. It will define each type of testing as well as the extent to which each type of testing will be executed.

Overview

The following will provide an overview of the testing phases that will be executed for the *CorkCollector* application.

Phase	**Responsible Person(s)**	**Description**
Unit Testing	Developer	Unit testing is the act of testing individual components of the system without external interaction. Unit tests should be written for every section of code and should be merged with the code.
Functional Testing	Development Team	Functional testing is the act of designing tests that incorporate many different units together to see how they interact to provide a specific feature to the product.
System/Integration Testing	Development Team	System testing is the act of testing that subsystems and functions interact appropriately with their specific interfaces according to the system design.
Stress Testing	Test Team	Stress testing is the act of testing how the system will react under extreme scenarios that place functionality at a higher risk.
User Acceptance Testing	Test Team	User acceptance testing is the act of testing whether or not the application can handle real world situations and scenarios. It is done to ensure that the product meets the user's needs.
Exploratory Testing	Test Team	Exploratory testing is the act of testing the system in unique and potentially destructive ways. This type of testing looks to explore the unhappy paths and to verify that there are no issues when the system is used inappropriately.

1. Unit Testing

1.1 Definition

For our purposes, unit testing is defined as the process of testing an individual software component within an isolated environment. Its primary purpose is to ensure that specific units of code meet their technical design. This is done by verifying that each designated unit properly handles its designated paths and outcomes, based on a strategically chosen selection of input variables. Any external dependencies a unit may have are functionally replicated to ensure isolation between all components.

1.2 Objective

Unit testing is a critical component to proper software development, and a vital part of our development plan. As such, testing specifications will be developed prior to the code being written, and act as the first trial for each unit of code once it is written. We expect it to provide the highest rate of defect discovery out of all components in the testing plan.

1.3 Testing Depth

Our unit testing strategy centers on treating every method as an individual unit to be tested. As stated in the SRS, our goal is to reach at least 80% code coverage through unit testing. We will test the outputs of each unit, given a suite of carefully selected input parameters and focus specifically on boundary values and other known problem areas. Depending on the variables, equivalence classes or decision tables will be used (whichever is more appropriate).

1.4 Approach

All members of the development team will utilize XUnit to verify the code they have written, creating a test suite for each function and eventually refocusing on feature confirmation. Each test will aim to validate factors including internal logic, error handling, data calculation, information processing and input validity, with a specific focus on stretch points and edge cases. All tests will follow a uniform best practices guideline to ensure consistency and clarity. Test results will be documented and saved both externally and internally, by specifying test results in an outside file as well as comments in the code. If any changes are made to remedy a failure, they will be directly addressed alongside the updated testing result.

1.5 Test Data

Each member of the development team will create their required test data prior to designing their unit testing suite. Data will vary depending on the unit being tested, but will always consist of a selection of specified inputs and expected outputs for the unit. This will allow each developer to easily outfit unit tests with proper values when they are implemented. Values will be mainly derived from equivalence class and decision table design, but our team will additionally focus on high stress, edge cases and critical values for each unit.

1.6 Entry Criteria

Prior to undertaking unit tests, each team member must verify the following.
1. The test environment must be up to date and ready for use.
2. All testing tools must be properly installed in the environment. Specifically, XUnit.
3. The specific unit being tested must be available.
4. The required supporting test data must be developed and accurate.

1.7 Exit Criteria

Prior to accepting unit tests as complete, each team member must verify the following.
1. All planned tests must have been implemented and run successfully.
2. The level of code coverage exceeds 80%.
3. An absence of outstanding defects with critical or high severity.
4. All high risk areas are thoroughly tested.
5. All development tasks for the specific unit have been completed.

1.8 Test Procedure API

Unit tests within the API focus on retrieval and modification of data. Each controller within the application will have a corresponding test class containing methods designed to test each method within the controller. The data used in these test cases will be generated based on valid, invalid, and edge case scenarios to ensure the entry and exit criteria are met.

2. Functional Testing

2.1 Definition

Functional testing is the act of testing if the features of the system work as expected when the different units of code are put together. Functional testing is a black box testing technique that is based off of the specifications of the application. It is used to answer questions such as “Does a feature work as expected” or “Can the user perform a task?”.

2.2 Objective

The objective of functional testing is to ensure that the features of the system are working as you would expect. These features are features required for regular users to perform the basic tasks associated with the function of the Cork Collector system, without them the application is considered unusable.

2.3 Depth of Testing

The scope of our functional test suite will cover the core features of the *CorkCollector* application that are deemed necessary for the application to be usable by a customer. It will also handle user input and string verification to ensure that a user cannot break the system with something that they input (i.e. malicious SQL injections). Installation onto a mobile device will also be tested as part of the functional test suite to ensure that users are able to access the application to begin with.

2.4 Approach

The test cases for the functional test suite will be determined based on the requirements elicited for *CorkCollector*. They will be divided into necessary core functionality and additional functionality. The required functions deemed to be core will each have test cases written for them so that the feature has complete coverage. All test cases will be automated using xUnit to increase productivity vs manually testing each case.

2.5 Test Data

The test data used in functional testing will be a specific set of inputs used to test that when input the expected output is still generated by the system. As mentioned above this will also handle invalid inputs to ensure that they are rejected or cleaned by the system prior to execution of the database query. This input data will be determined from equivalence, boundary and decision table test plan creation.

2.6 Entry Criteria

Several conditions need to be met before the functional test suite may be developed:

1. Ensure a test environment is set up and ready to use by every team member
2. Ensure all testing tools are installed on the environment (xUnit)
3. Ensure the equivalence, boundary, and decision table test plans have been created and reviewed for accuracy.
4. For each test case:
   a. Ensure that all of the the features different units have been completed and have appropriate unit tests around them.
   b. Ensure that the appropriate test data has been generated as determined by the equivalence, boundary and decision table test plans.

2.7 Exit Criteria

The criteria for accepting the functional test suite is as follows:

1. Ensure all planned test have been implemented and output the expected result.
2. Ensure that the planned tests cover all of the features identified as being core and necessary for application functionality.
3. Ensure a code coverage level of 90% or higher.
4. Ensure that the feature under test does not cause any regressions of previously implemented features.
5. Ensure that the functional test suite may be run by any team member to ensure future feature development does not negatively impact existing features.

3. System & Integration Testing

3.1 Definition

System and Integration testing refers to testing the integration between the software subsystem and their interfaces. This can also be referred to as end-to-end testing. This phase of testing involves ensuring that data is transferred reliably and securely between the different layers of the system as well as between components of the software subsystem layer. It will also be used to test how the system behaves in different environments, in this case on different Android devices.

3.2 Objective

The objective of system and integration testing is to ensure that all of the components of the system interact properly with their interfaces and meet the requirements of the systems design. The main focus of this phase is about testing the cross-functionality of the data, rather than testing within one unit or function. It will ensure that the application is able to share data among its different components and systems without a loss of security or reliability to the user.

3.3 Depth of Testing

System and integration testing is used to ensure that the system is robust and that it operate effectively in all environments. As such, it is very important to be very thorough. Due to the nature of it testing how components interact with each other, it needs to be tested (at least at a high level) after each new component is integrated. It will ensure that the feature was integrated properly and that it is able to behave properly in the system without disturbing the other features already in place. This will be done strategically and ensure that the areas that could be effected as tested (we need not test every environment every time). Towards the end of the project (and at key milestones throughout) there will be a full system and integration test where end to end testing will be performed on all features on multiple environments.

3.4 Approach

In order to appropriately verify that the components are working together there will need to be some manual verification. Integration testing will focus on components of the application such as navigation within each interfaces and error handling when issues are encountered. Testing the user interfaces. Since our team has a dedicated test team having this tested manually will not be an issue and we will therefore not be implementing a UI test environment in code. There will also be the ability for the development team to implement coded tests in XUnit to test what happens during things such as system rollbacks, restores, data migrations etc. These can be tested with different inputs and verifying the expected results. There is also the opportunity for the development team to code tests to verify that all components can work together, even if they will not do so from the front end of the application to ensure that the application is more robust for future development.

3.5 Test Data

The test data for system and integration testing will be generated based on the ability of the system to replicate certain scenarios such as a database rollback. The test cases for the system and integration testing will be well documented in a separate “System/ Integration Test Cases” document which will be generated once the system is more developed. For environment testing the most popular Android devices will be tested, such as Samsung S8, Nexus, Pixel, HTC Touch. The tests performed on these devices will be recorded and evaluated against their past test and each other each time system and integration testing is performed.

3.6 Entry Criteria

Several conditions need to be met before the system and integration test suite may be developed:

1. All required features have been implemented and there are no missing components to the system.
2. All issues have been fixed/closed.
3. The product satisfies all non-functional requirements outlined in the SRS document
4. All the above documentation has been created and is ready for use.

3.7 Exit Criteria

The criteria for accepting the system and integration test suite is as follows:

1. The documents mentioned above have been filled out and no defects have been found that need to be addressed.
2. The application runs as expected on all supported devices and OS's

4. Stress Testing

4.1 Definition

Stress testing refers to putting the system under extreme circumstances to observe how it behaves. Stress testing should be used to simulate user actions with the system while it is subjected to specific simulated environments that will increase its risk of system failure. Stress testing often targets hardware configurations and evaluates how the hardware implemented handles interactions from the software components of the system.

4.2 Objective

The objective of stress testing is to determine if/when the system will be led to failure as a result of extreme conditions so that measures may be put in place to fail more gracefully or to warn users prior to failure. Stress testing allows of the identification of specific components weaknesses or vulnerabilities so that they may be strengthened or have precautionary measures put in place so that the users experience is not hindered under those extreme circumstances.

4.3 Depth of Testing

Stress testing can be done on nearly all components of a system in some capacity. Due to the limited time line of the *CorkCollector* development cycle the stress tests will only be performed on the most likely to happen extreme cases. In this way, there will be minimal stress testing done except in the case of project early completion in which case the surplus of resources will allow for additional stress testing to be done.

4.4 Approach

As was discussed above, only the most likely extreme cases will be tested. In order to do so, we must first identify the most likely extreme cases. This will be done through developer analysis during the development stages. For each feature completed, the developer will inform the development team of the different areas that could encounter an extreme case. These will then be documented in the “Stress Test Candidates” document. During planning, the team will discuss which of the stress test candidates they feel are the most likely to occur (also based on industry experience of mobile applications) and those will undergo stress testing as part of their testing phase. If the stress test is better done as a coded test, then it will be required that the developer also create a written xUnit test to cover that scenario as identified by the team during planning. It will be expected to be complete by the end of the testing phase of that feature/component.

4.5 Test Data

The test data for each scenario will depend on the area that it needs to be stress testing. Similarly to the scenarios, this will be identified by the developer when added to the “Stress Test Candidates” document. The test data will then be generated by the test team either for manual use or for the developer to verify their coded test that they are writing to cover that particular stress test.

4.6 Entry Criteria

Several conditions need to be met before the stress test suite may be developed:

1. All other tests must be completed for the feature/component in question.
2. The scenario needs to be part of the Stress Test Candidates document.
3. The scenario must be discussed by all members of the team in the planning meeting to verify its likelihood.
4. The test data needs to have been generated by the test team.

4.7 Exit Criteria

The criteria for accepting the stress test suite is as follows

1. Ensure that improvements have been made where necessary to the strength of components.
2. Ensure that the user will receive a helpful message in case of system failure (if no further strengthening can be achieved)
3. Ensure all high risk extreme scenarios have been stress tested.
4. Ensure all results have been observed and documented for future reference.

5. Exploratory Testing

5.1 Definition

Exploratory testing is the act of using the application in unique ways to explore the different paths that may be taken by a user. This type of testing is used to explore the “unhappy” paths of the software that may have been forgotten or that may not have been identified in the other phases of the testing process. Exploratory testing is unique in that it does not follow a preplanned script and instead relies on the tester’s ability to think of scenarios on the go that may cause issues within the system.

5.2 Objective

The objective of exploratory testing is to ensure that there are no ways that a user could interact with the system that could cause a failure. Since it is not restricted by a predetermined script it allows the tester to explore different parts of the application and try to use the application in a way that was not anticipated to ensure the integrity of the application remains. Exploratory testing frequently finds paths that would not have been thought of that cause different components to interact in an unexpected way which can cause issues if they were not created to interact.

5.3 Depth of Testing

The depth of exploratory testing changes depending on the feature under test. Some features require very little testing while others will require extensive exploratory testing if it is more of a fundamental change to the system. Each new component will undergo testing as determined by the test team determined based on size of change, how much it interacts with the rest of the system etc.

5.4 Approach

Due to the unique position of the *CorkCollector* team having a dedicated test team each new component or feature committed will undergo exploratory testing. This allows for incremental testing that can ensure that new features interact as expected with other features and allows for testing around existing features to ensure they do not get broken by a new change. As mentioned previously, there is no predetermined test plan in exploratory testing. Instead, test plans are written at the time of testing depending on the feature under test. These notes on the test plan are noted in the ticket associated with the change itself.

5.5 Test Data

Test data for the exploratory testing will be determined by the tester at the time of testing. The test data used will be recorded in the ticket similarly to the test plan. This will also allow for the data to be reused in the future if there is any issues discovered during this phase of testing in the application.

5.6 Entry Criteria

Several conditions need to be met before exploratory testing may begin:

1. A demo must be obtained from the developer outlining the change and the associated tests that have been written and committed with it.
2. A test plan must be written in the ticket associated with the change
3. Test data must be procured or created by the tester for use in the testing.

5.7 Exit Criteria

The criteria for completing the exploratory testing is:

1. All defects or issues found have been discussed with the developer
2. Tickets have been filed for the defects, as well as notes on how to automate tests for all defects found.
3. Suggestions for additional tests have been made to the developer which would be created prior to integration of the feature.

6. User Acceptance Testing

6.1 Definition

User acceptance testing is the final phase of the software testing process. User acceptance testing must be performed before the product is released because it is the act of testing whether or not the software meets the user's needs. User acceptance testing has also been called beta testing as it is done by the client (or potential clients) or the project stakeholders.

6.2 Objective

The objective of user acceptance testing is to validate that the system is able to perform its basic everyday functions appropriately, and to ensure that it is fit for use. If an application fails user acceptance testing it will not be shippable.

6.3 Depth of Testing

The amount of user acceptance testing that must be performed directly correlates to the number of requirements and user stories for the application under test. Test cases will be prioritized in terms of importance for everyday use of the system. User acceptance testing will also retest unexpected/ invalid inputs from users to verify nothing has been missed that could cause system issues based on user input.

6.4 Approach

User acceptance testing is a manual testing task. Since we need to see how the application performs when a user is actually interacting with the UI it would be inefficient to use coded testing. Due to the nature of *CorkCollector* it will be user acceptance tested in the Niagara region by friends and family of the team members. They will use the application in the setting in which it is intended to be used, while exploring wineries in the region. Due to the nature of *CorkCollector*'s user acceptance testing it will need to be planned in advance. This allows our users to also take advantage of the rewards program as they will be the first to use the application, they have a head start in earning rewards within the application as an added incentive to assist in testing.

6.5 Test Data

The test data for the user acceptance testing is entirely up to the user. Since they will be using it in the region it should be able to handle all input from users that would come up in the day of exploring wineries. The wineries explored will be up to those performing the tests as we plan to support all wineries in the area.

6.6 Entry Criteria

Several conditions need to be met before the user acceptance test may be performed:

1. All required features have been implemented
2. All critical GitHub issues have been fixed/closed
3. Any other issues may not be predicted to have a negative impact on user experience
4. The product satisfies all non-functional requirements
5. All other test suites have been completed and documented

6.7 Exit Criteria

The criteria for accepting the user acceptance testing is as follows:

1. Beta user feedback has been received
2. The results of the user acceptance test have been recorded
3. Any defects that were found during the testing phase have been fixed/closed

7. Defect Management Process

All defect management for *CorkCollector* will be handled through GitHub Issues. GitHub Issues is integrated directly into the projects private development repository, allowing the team to quickly and effectively track defects through the entire development process.

7.1 Benefits

Some of the helpful features that GitHub Issues has that the team will be taking advantage of are:

1. Minimal content required to create an issue - this allows for the developers to work more quickly to open and assign issues.
2. Since issues may be assigned it is tracked who is responsible for fixing an issue and this can be correlated for sprint planning.
3. Issues can be assigned labels for quick categorization or filtering to identify problem areas.
4. Issues can be associated to a milestone, in our case a sprint, so that there is a timeline for the issue to be fixed.
5. Issues can have comments left from anyone on the development team which allows for contribution of ideas from team members who are not assigned the issue.
6. Issues are created based on commits that introduced them and this can be tracked within the issue which makes testing the fix much faster
7. Issues are displayed on a very user friendly dashboard which allows the team to track all issues assigned to them as well as all other issues the team is handling currently.

7.2 Defect Opening

Once a defect or issue has been discovered then the tester or developer who found the issue will create a new GitHub Issue. A brief description will be added and the developer responsible (who worked on the ticket that introduced the issue) will be assigned to the ticket. A verbal conversation will also occur to alert the developer to the issues creation. The issue will be discussed in the next team meeting and once an agreement has been reached a severity label will be applied to the issue for tracking purposes and for sprint planning purposes.

7.3 Severity

As discussed above, there will be different severity levels for defects found. The four levels of severity are outlined below with their respective actions required.

7.3.1 Low

A low severity issue means that the defect does not cause serious interference with the operation of the application. This type of defect is generally more of a cosmetic issue or something that is not of great concern if it was released to customers as it does not hinder the experience of using the application greatly.

7.3.2 Medium

A medium severity issue is something that makes the system difficult to use. If something is of medium severity it would cause trouble for the user but it would not render the application unusable, simply would be very inconvenient and should be fixed prior to release.

7.3.3 High

A high severity issue is something that means some part of the system is currently unusable. This type of defect would render parts of the application unusable for a user and it would therefore block any type of release as we would be releasing a product that does not fully meet the requirements of the system.

7.3.4 Critical

A critical severity issue is something that renders the entire application unusable. Critical defects need to be fixed right away, meaning that other work may be dropped to get eyes on a critical issue faster. Critical defects cause the entire application to be deemed unusable and can include things such as security breaches, data manipulation vulnerabilities etc.

7.4 Defect Fixing

Defect fixing will be in parallel with feature development. The number of defect fixes vs features pulled into a sprint to be worked on will depend on the severity of the issues currently identified in the issue list. Defects of a higher severity will be fixed before those of a lower priority. If a critical issue is found during a sprint then it will become the top priority and other work in the sprint will halt until a time when that critical issue has been handled appropriately. This allows for the team to fix the most vital issues in a timely manner.

When a defect has been fixed the assigned developer should do their best to ensure that there is appropriate testing put in place to ensure that this issue does not return. After the coded test have been completed then it will be handed back off to a tester who will verify the fix as well as go through regular testing procedures as they would with a feature ticket having landed. The individual who filed the ticket will also be asked to review both the coded tests and the test plan of the tester to ensure that all areas that need to be tested will be. If an issue returns, or is not fixed, then the issue may be reopened in Git Issues so that the team is aware. There will also be a note added to the issue if it is a recurring issue to delve into why the issue keeps returning. If the issue is fixed as verified through testing, then it will remain closed and efforts can be focused elsewhere.

Conclusion & Recommendations

The development of the CorkCollector application was a complete success. The application is fully functional and allows for users, both old and new, to better experience the Niagara region. They will be able to interact with the wineries and the wines that they have available for tasting. Users can now have a better way to track their adventures through the region and they are able to keep a record for the future. One of the highlights of the application is ability for the users to add wines to their cellar, thus allowing them to track all of the wines that they currently have at home and review the notes that they have left for themselves for the future. Another great aspect is that those who frequent the area are able to view recommendations based on the different wines that they have tried and liked in the past so that they can more adequately plan their next trip. All of these features and the overall ease of use of the application will let all visitors to the area enjoy their experience and be more aware of all that the wineries have to offer.

At this point in this the CorkCollector Android Application is fully functional for all regular users. However, some of the original functionality that was defined has since been changed. The rewards system has been removed from the application in favour of moving towards an architecture where wineries will have their own user logins and will be responsible for keeping their information up to date. This switch was done so that the necessary costs that will be added when the user base increases can be covered by the winery advertising fees. Additionally, in order to expand our user base we will be porting the application to also be able to run on iOS devices in addition to Android.

The first change that was introduced was the switch from a rewards system for users in order to keep the application information on the wineries up to date. After discussion with some of the wineries in the region it was determined that it would be better to involve them in the application. As such, the wineries will be able to sign up as a winery user instead of a regular user and will be able to keep all of their winery information such as hours, tasting menu, prices etc. up to date instead of relying on users to do so. In order to achieve this they will be getting their own winery user interfaces on the application that are different from that of users. This will allow them to view their current tasting menu, winery information and the reviews which have been left on their winery and their wines. They will be able to update their tasting menu or information and respond to user reviews. They cannot remove the review but they are able to respond to the review and clarify so that they can improve the experience for users moving forward.

Next, in order to allow more individuals to become users we will be porting the application to the iOS platform. This will allow us to reach the vast majority of the mobile user market. This switch will also ensure that all of the wineries can access the application since the current model would require them to all have access to an Android phone. In order to facilitate this additional programming effort we will be using a porting program in order to port the majority of our front end code to xcode. Our backend API will not need to change since the calls being made to the API will remain the same. The additional programming effort to ensure that the front end is ported properly is estimated to take about 3-4 weeks at which time we will be applying to release the application on the apple store.

Due to this planned increase of the user base we will need to update our servers from the current AWS Free Tier that we are using to a paid tier. Fortunately, since AWS is a cloud based service provider we are able to dynamically scale based on the number of users at any point in time. This means that the costs incurred are hard to estimate until we have a better idea of the number of users that will be using the application at any point in time. However, it does mean that we will not be paying for services which are going unused as the system will automatically provision more resources when the users increase and decrease them when they are no longer necessary. This is the primary benefit and why are planning to stay with AWS for the foreseeable future.

In order to help with these costs we will be offering advertising services to the winery users. Currently, when they have a promotion or event at a winery they rely on mainly word of mouth advertising. By adding them to the application they now have the ability to pay to advertise through the application. A banner would appear at the bottom of a user's screen that would announce the upcoming event/promotion and would not be invasive to the user experience. Another reason why we do not believe that this advertisement style will have a negative impact on regular users is that the advertisements are application based. They will not be getting advertisements that are irrelevant to them and they will only be for the wineries in the application. The pricing plan is shown in the appendix. This way the wineries are able to use the application in a basic form for free or be able to increase their potential revenue through minimal advertising costs.

We believe that these changes will enable our application to be used and increase the experience for a wider audience. None of these changes are radical enough to have a negative impact on the team and we are confident in our ability to maintain the application moving forward.

Appendix

1. Pricing Plan

1 Week Advertisement	$50
2 Week Advertisement	$75
1 Month Advertisement	$100
Ability for users to subscribe to update notifications from winery	$300/yr

2. Android Application Source Code

Source code can be found at: https://github.com/Russ-Stirling/CorkCollectorAndroidApp

3. Web API Solution Source Code

Please note that to run the projects in this repository a client authentication certificate is required for the database currently in use. If this is needed for marking please email russell.k.stirling@gmail.com and one will be provided.

Source code can be found at: https://github.com/Russ-Stirling/CorkCollectorWebAPI

CorkCollector

Android Application

User Guide

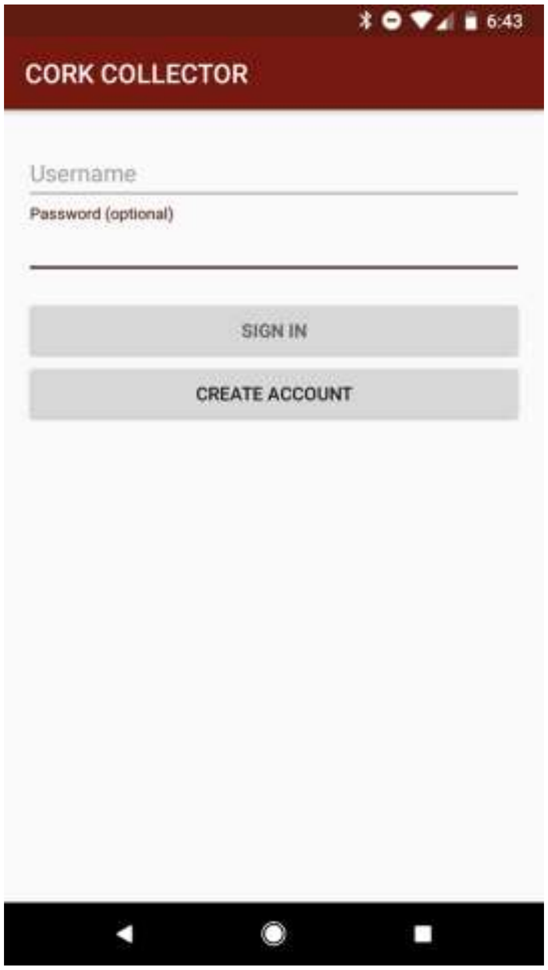
Introduction

CorkCollector is an application that allows users to plan their trips to the wineries around the Niagara region. It allows the users to view the different wineries, get directions, view the tasting menus, and the wines at different wineries. They may also view the wineries that they have visited, the wines they have tasted and the wines that they have purchased for later consumption.

This guide will take users through the different functionality and help with troubleshooting potential issues.

If issues arise, please contact the team at bhanna4@uwo.ca, rstirli2@uwo.ca, or dlorence@uwo.ca

Login

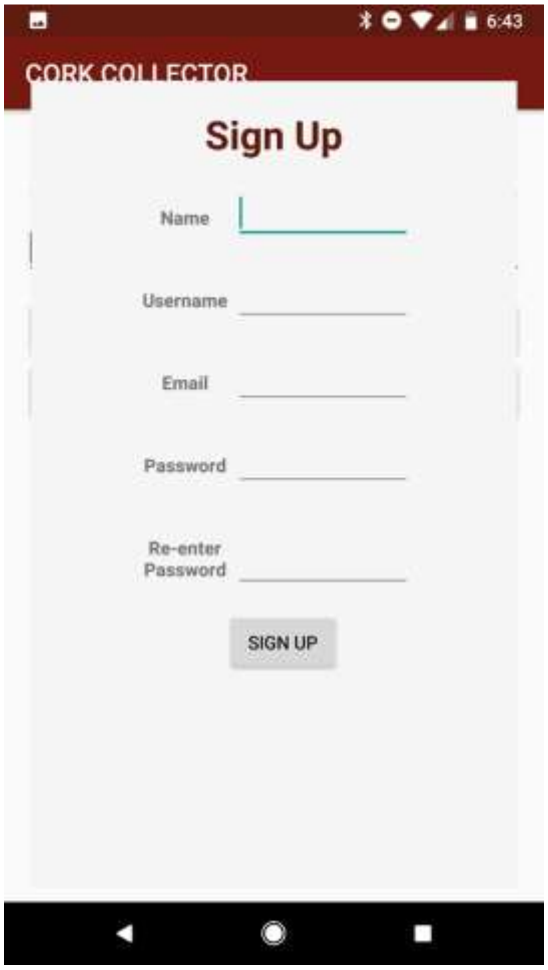


Users who have previously registered for the CorkCollector application must log in by:

Entering their **username**
Entering their **password**
Clicking **Sign In**

Sign Up
Users who have not previously used the site must create a new account by clicking **Create Account** to access the "Sign Up" page.

Sign Up



In order to register new users must enter the following information:

Name as they would like it to appear in their profile

Username which is the name which will appear on all ratings and reviews and be used for login. If a username is already taken, the system will alert the user and they must select a new username to finish registration.

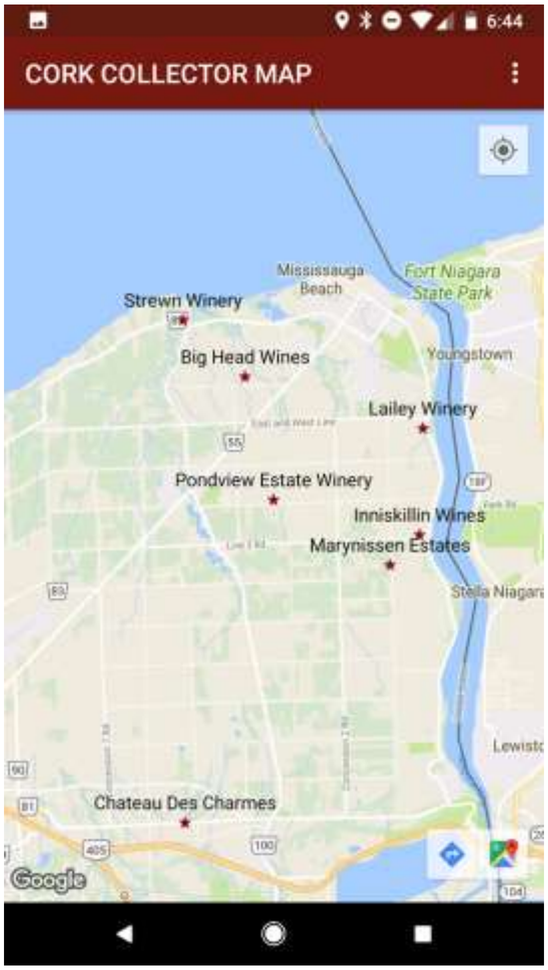
Email which they will use to receive promotions

Password which must be at least 6 characters long and is case sensitive

Re-enter Password which must match the first password entered. If the passwords do not match the system will alert the user. They must match in order for registration to complete.

Once information is entered the user clicks "Sign Up" in order to complete registration. They will then be returned to the "Login" screen to enter their information to sign in.

Map



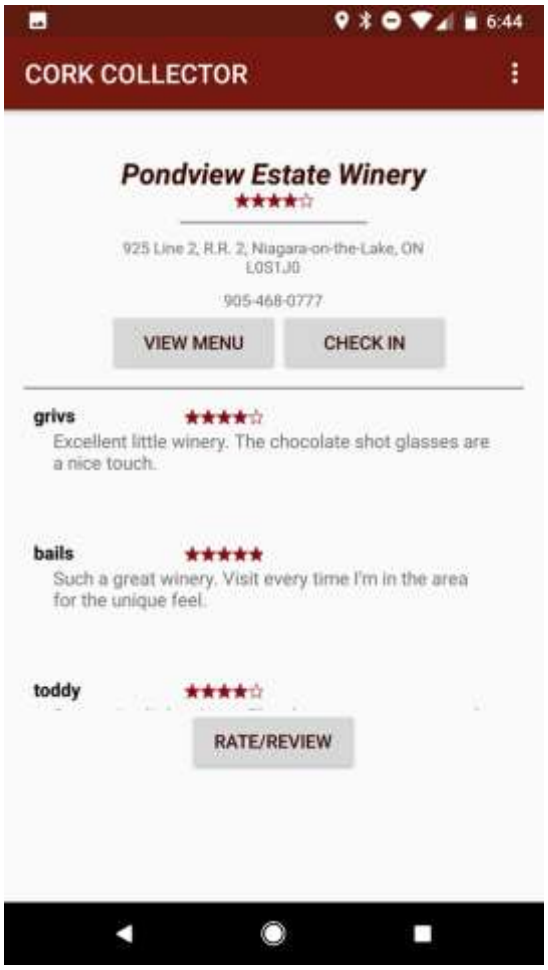
The first screen presented to a user is the map of the Niagara region.

On the map users may:

Zoom in/Zoom out via pinching the screen (same as a regular google maps screen)
Select Winery by clicking on the star/name of the winery
Get Directions to the last selected winery via the small blue arrow and maps icon in the lower right hand corner

The menu screen appears in the upper right hand corner and allows the user to navigate to the "Profile" screen or back to the "Map".

Winery



Pondview Estate Winery
★★★★☆

925 Line 2, R.R. 2, Niagara-on-the-Lake, ON
L0S1J0

905-468-0777

grivs ★★★★☆
Excellent little winery. The chocolate shot glasses are a nice touch.

bails ★★★★★
Such a great winery. Visit every time I'm in the area for the unique feel.

toddy ★★★★☆

Winery Information
The information displayed at the top of the screen lists the information about the currently selected winery

> **Winery Name**, in the example "Pondview Estate Winery"
> **Average Rating** shows the average rating from all user ratings and reviews
> **Address** shows the address of the winery
> **Phone Number** shows the contact number of the winery

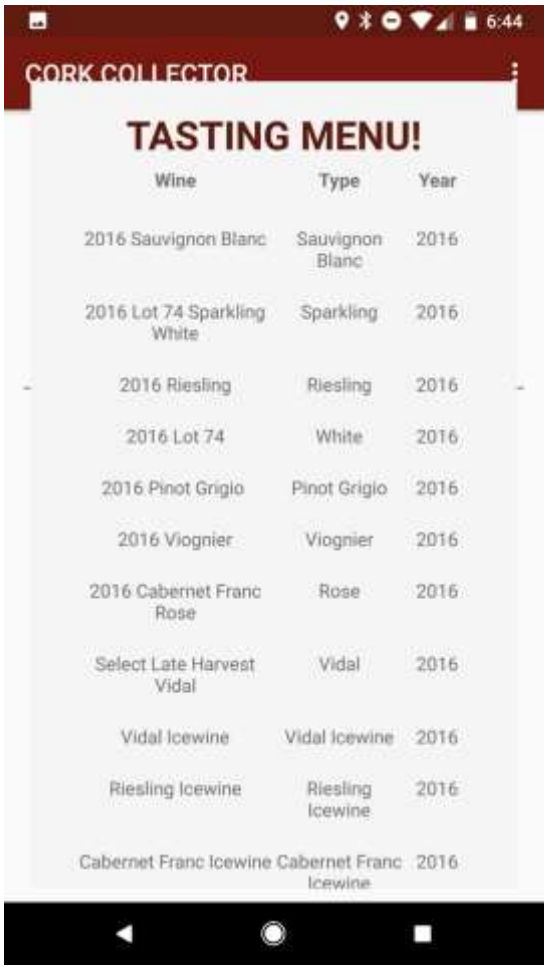
View Menu
In order to view the wines that are currently available for tasting at the specific winery. Once clicked the "Tasting Menu" screen will appear above the winery screen.

Check In
Users may click "Check In" in order to specify that they have visited this winery and it will be added to their list of Visited Wineries which can be viewed from the "Profile" page.

Rate/Reviews
The scrollable section which contains all ratings/reviews from users who have visited and commented on the winery.

View Rating/Review the review text may be selected which will bring up the "Edit Rating/Review" page which allows the user who made the review to edit it or delete it in the future.
Rate/Review can be clicked which will then load the "Rate/Review" page which will allow the user to post their own comments and rating.

Tasting Menu



TASTING MENU!

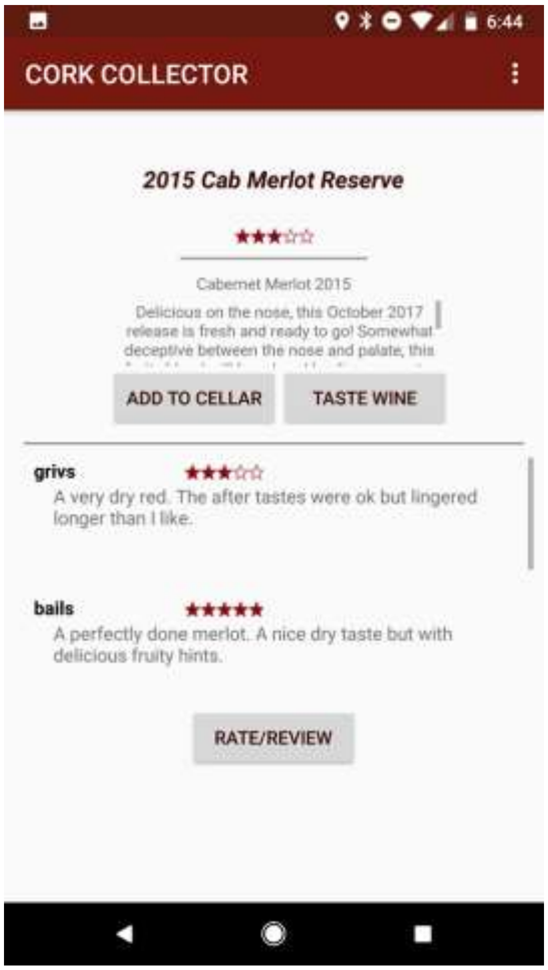
Wine	Type	Year
2016 Sauvignon Blanc	Sauvignon Blanc	2016
2016 Lot 74 Sparkling White	Sparkling	2016
2016 Riesling	Riesling	2016
2016 Lot 74	White	2016
2016 Pinot Grigio	Pinot Grigio	2016
2016 Viognier	Viognier	2016
2016 Cabernet Franc Rose	Rose	2016
Select Late Harvest Vidal	Vidal	2016
Vidal Icewine	Vidal Icewine	2016
Riesling Icewine	Riesling Icewine	2016
Cabernet Franc Icewine	Cabernet Franc Icewine	2016

The tasting menu shows all of the wines currently available for tasting at that winery. Each one may be clicked which will bring up the wine page of that specific wine.

The wine information includes:

Wine Name as specified by the winery. It may also include information about specific nature. This is a clickable link which will bring the user to the “Wine” page.
Type which specifies the type of wine it is classified as.
Year which specifies the year in which the wine was bottled.

Wine



2015 Cab Merlot Reserve

Cabernet Merlot 2015

Delicious on the nose, this October 2017 release is fresh and ready to go! Somewhat deceptive between the nose and palate, this

grivs
A very dry red. The after tastes were ok but lingered longer than I like.

bails
A perfectly done merlot. A nice dry taste but with delicious fruity hints.

Wine Information
The information displayed at the top of the screen lists the information about the currently selected wine.

> **Wine Name** shows the name given by the winery to that particular wine. In the example, "2015 Cab Merlot Reserve"
> **Average Rating** shows the average rating from all user ratings and reviews
> **Type/Year** shows the type of wine and the year it was bottled. Year is optional.
> **Description** shows a scrollable description that the winery has given to the wine. This is optional.

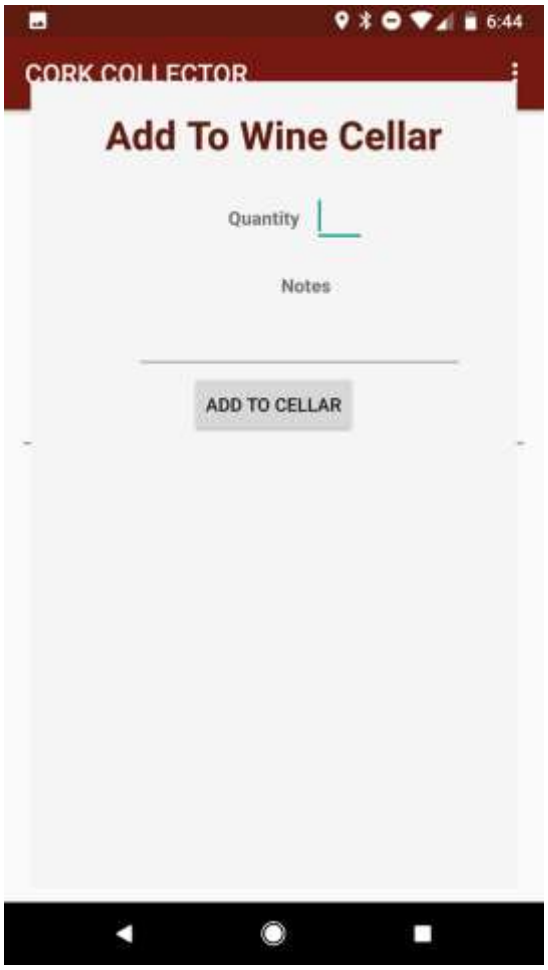
Add to Cellar
If a user is purchasing a bottle of wine to bring home and drink in the future they may select to add the bottle of wine to their cellar. Once you click "Add to Cellar" the "Add to Cellar" page will appear for users to enter further information.

Taste Wine
Users may click “Taste Wine” in order to specify that they have tasted this wine and it will be added to their list of Tasted Wines which can be viewed from the “Profile” page.

Rate/Reviews
The scrollable section which contains all ratings/reviews from users who have visited and commented on the wine.

> **View Rating/Review** the review text may be selected which will bring up the “Edit Rating/Review” page which allows the user who made the review to edit it or delete it in the future.
>
> **Rate/Review** can be clicked which will then load the “Rate/Review” page which will allow the user to post their own comments and rating.

Add to Cellar



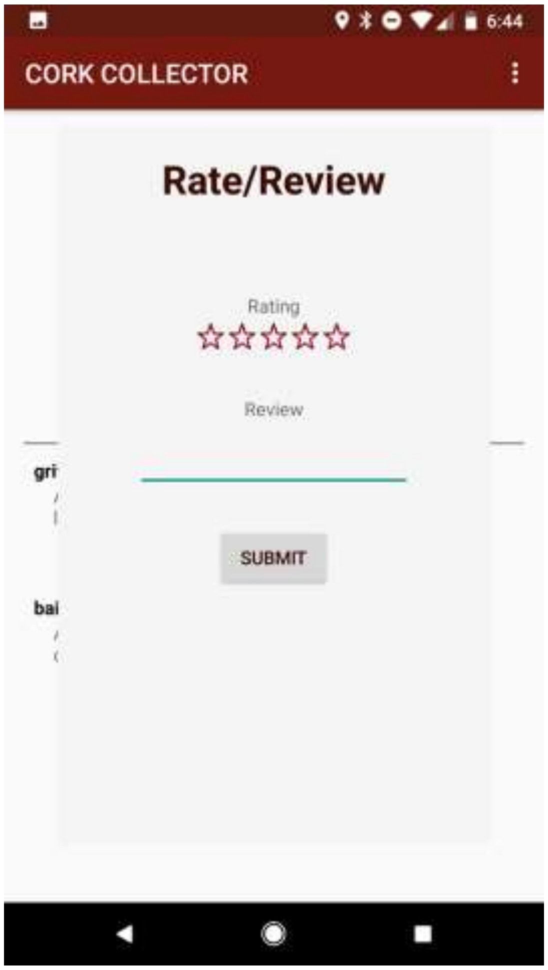
In order to add wine to the users profile they must enter information:

Enter a **quantity** to specify how many bottles they are purchasing

Enter **notes** to leave some personal comments on the wine for themselves which they do not want to share as a review.

Click on **Add to Cellar** which will add this information to their Wine Cellar which can be viewed from the user's "Profile"

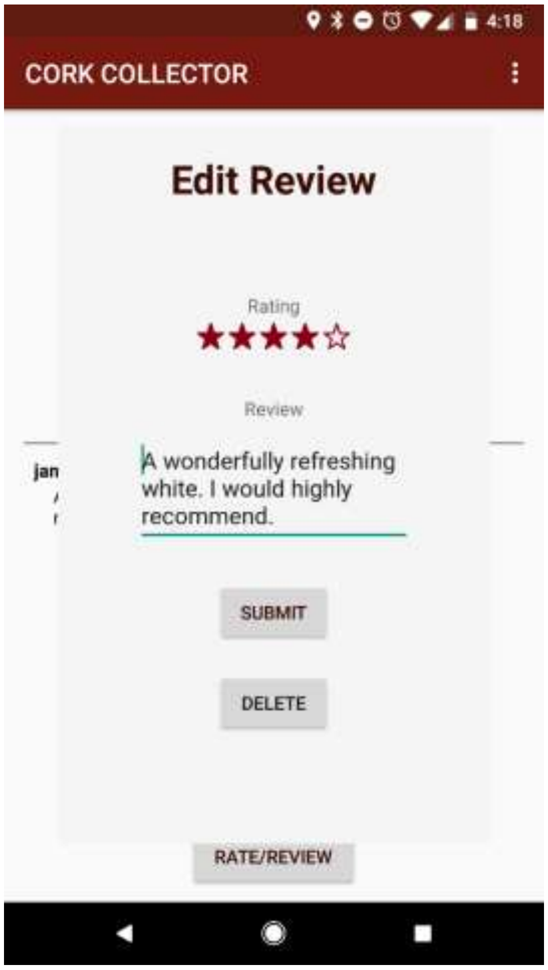
Rating/Review



In order to leave a rating/review on a wine or a winery the user must:

Entering a **rating** based on a 5-star scale.
Entering a **review** which can be as long as the user wishes.
Click **submit** which will post the rating/review and bring the user back to the winery or wine page and refresh to show the new rating/review.

Edit Rating/Review

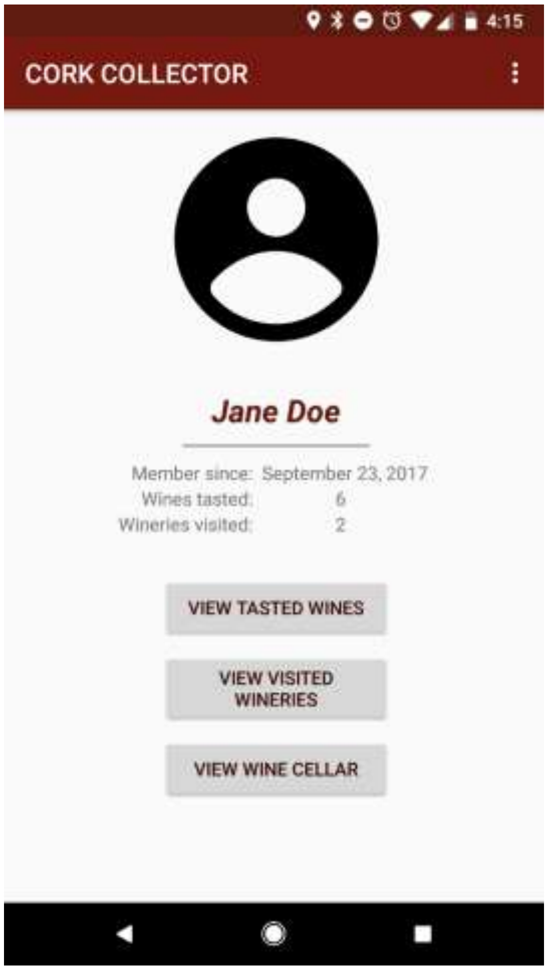


A wonderfully refreshing white. I would highly recommend.

This screen will only appear once a review is clicked on that was originally written by the current user.
In order to edit a rating/review on a wine or a winery the user may:

Enter a new **rating** based on a 5-star scale.
Enter a new **review** which can be as long as the user wishes.
Click **submit** which will repost the rating/review and bring the user back to the winery or wine page and refresh to show the new rating/review.

Profile



Member since:	September 23, 2017
Wines tasted:	6
Wineries visited:	2

User Information
Shows the information about that specific user including:

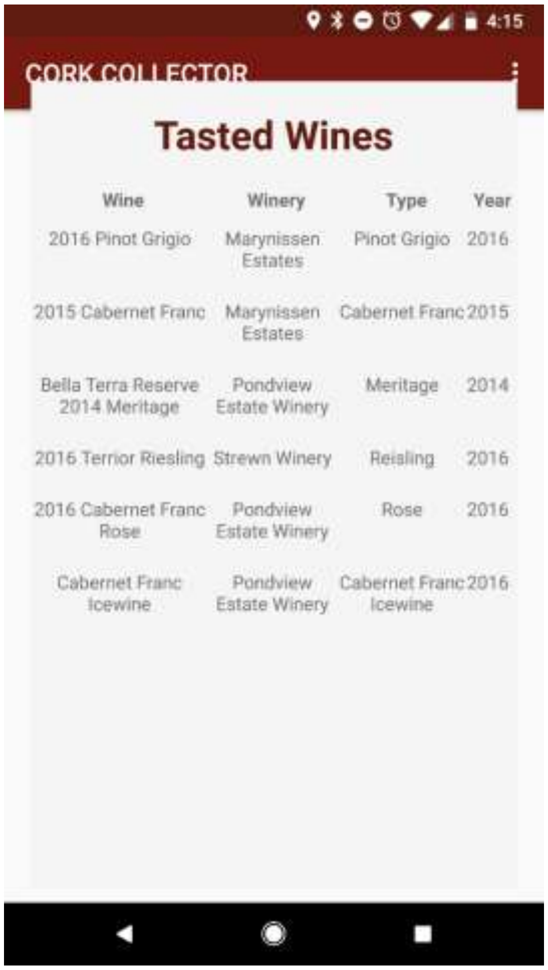
Name which is the name that the user entered during sign up
Member since information which states the date the user signed up
Wines Tasted is a count of the wines that user has tasted
Wineries Visited is a count of the wineries that the user has visited

View Tasted Wines
Users may click on “View Tasted Wines” which will show the “Tasted Wines” screen.

View Visited Wineries
Users may click on “View Visited Wineries” which will show the “Visited Wineries” screen.

View Wine Cellar
Users may click on “View Wine Cellar” which will show the “Wine Cellar” screen.

Tasted Wines



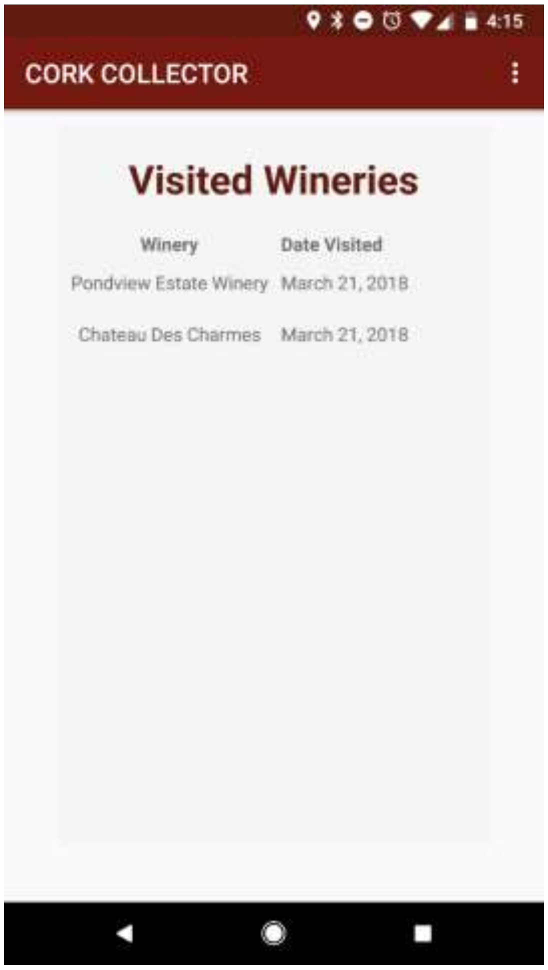
Tasted Wines

Wine	Winery	Type	Year
2016 Pinot Grigio	Marynissen Estates	Pinot Grigio	2016
2015 Cabernet Franc	Marynissen Estates	Cabernet Franc	2015
Bella Terra Reserve 2014 Meritage	Pondview Estate Winery	Meritage	2014
2016 Terrior Riesling	Strewn Winery	Reisling	2016
2016 Cabernet Franc Rose	Pondview Estate Winery	Rose	2016
Cabernet Franc Icewine	Pondview Estate Winery	Cabernet Franc Icewine	2016

A user’s tasting menu will display all of the wines that they have tasted. It displays the following:

Wine displays the winery given name of the wine. It is a clickable link to the wine page.
Winery displays the name of the winery that made that wine.
Type displays the type of wine.
Year displays the year the grapes were harvested.

Visited Wineries



Visited Wineries

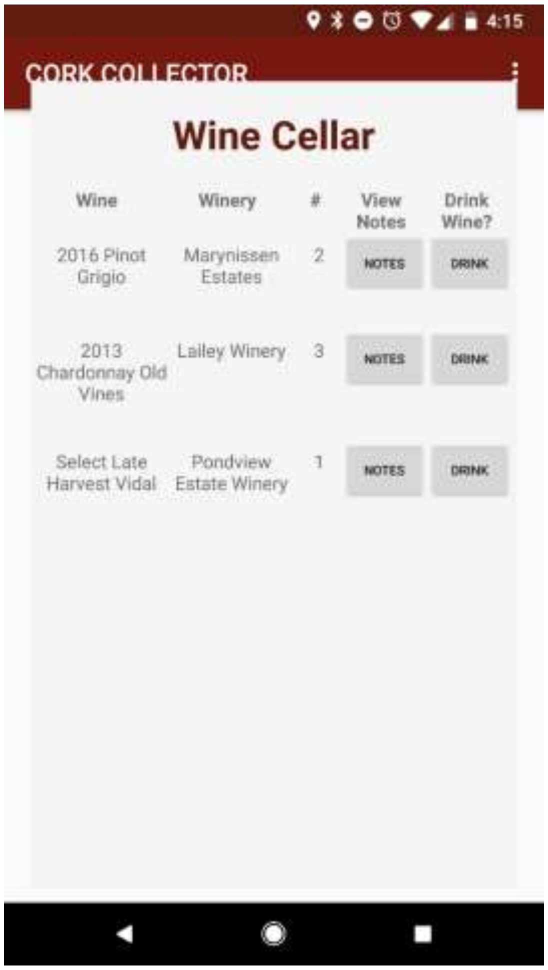
Winery	Date Visited
Pondview Estate Winery	March 21, 2018
Chateau Des Charmes	March 21, 2018

A user’s visited wineries will display all of the wineries that they have checked into. It displays the following:

Winery displays the name of the winery that made that wine. It is a clickable link to the winery page.

Date Visited displays the date that the user visited and checked into that winery.

Wine Cellar



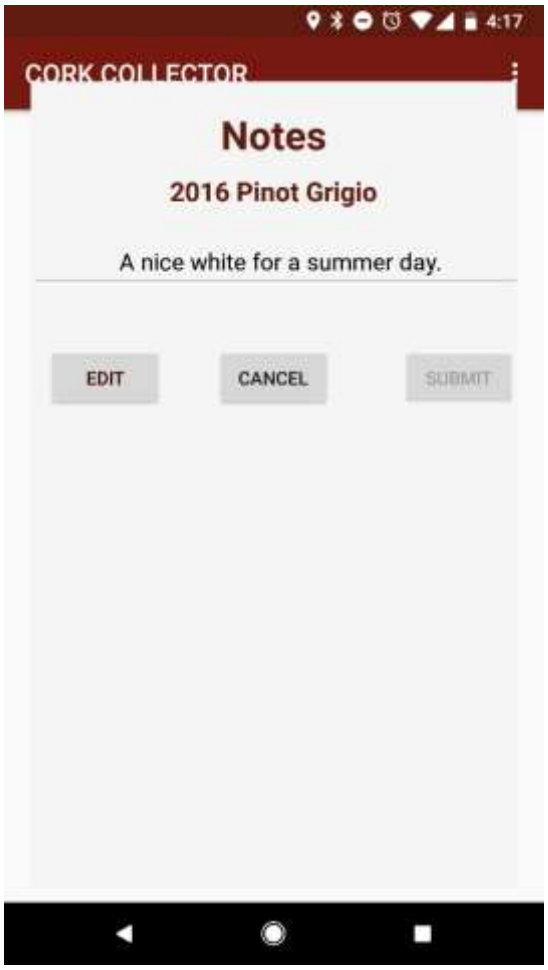
Wine Cellar

Wine	Winery	#	View Notes	Drink Wine?
2016 Pinot Grigio	Marynissen Estates	2	NOTES	DRINK
2013 Chardonnay Old Vines	Lailey Winery	3	NOTES	DRINK
Select Late Harvest Vidal	Pondview Estate Winery	1	NOTES	DRINK

A user's wine cellar will display all of the wines that they have purchased and added to their "cellar" to drink later. It displays the following:

Wine displays the winery given name of the wine. It is a clickable link to the wine page.
Winery displays the name of the winery that made that wine.
# displays the number of bottles that the user still has to drink.
View Notes is a button which, when clicked, will display the "Notes" page
Drink Wine? Is a button which, when clicked, will decrease the number of bottles you have on hand.

Notes



A nice white for a summer day.

A user's note will be brought up when a user clicks to view their notes on a wine in their cellar from the "Wine Cellar" page. It displays:

Wine Name which is the wine for which the note is added.

Note is the note that the user entered on the "Add to Wine Cellar" page.

Edit Note

By clicked on "Edit" the text of the note will become editable. The user can then change their note. By clicking on "Submit" they will be able to save the new note. This will bring them back to the "Wine Cellar" page. "Cancel" will also bring the user back to the "Wine Cellar" page.

Troubleshooting

Location not appearing on the map

1. The map is restricted to the Niagara region and you must be in the region in order to appear on the Map.
2. Ensure that you have allowed the CorkCollector application to access your location. This can be done through the settings menu.

Check-in not allowing due to "Out of Range"

1. You need to be within one kilometer of the winery in order to check-in. Ensure that you are within this range in order to check-in.

Login information showing as incorrect

1. Please email the administrators at bhanna4@uwo.ca, rstirli2@uwo.ca, dlorence@uwo.ca and explain the issue. The administrators will then reset your password after information verification

Vitae

Please find a resume for each of the team members on the following pages.