Table of Contents

This page is intentionally left blank.

Requirements

1 INTRODUCTION

1.1 PROBLEM DEFINITION

First year students and students at WLU often grapple with finding information to their questions that are often difficult to find. With the recently reconstructed WLU site and the legacy domain, a lot of information that was already difficult to find has become even more difficult to find. As a result of this, a lot of the time students are forced to e-mail advisors, staff and use other avenues to find the information they need. For example, finding out how to compute your GPA properly is difficult to find if one is just simply browsing the website. In fact, the *Google Search Engine* often does a better job navigating than WLU does on their official site.

Due to this, students need a streamlined way of accessing information without spending up to an hour sifting through to find the information they need. Finding out a simple requirement to a course, asking a question about class, or getting directional help does not need to be difficult. Students have been accustomed to the way things are – but that does not mean there isn't a better way. Students might not realize it, but they need a better way to get the information they want at a faster pace in a technologically advanced world.

The major block for students in getting this information is due to poor communication channels and lack of timely feedback.

1.2 PURPOSE

The purpose of this document is to give detailed requirements for our application "Help Me! Laurier" to give a broad overall understanding of how our application will work before creating the application. It will show how the application will function as a whole with an overarching overview. This document will be used to evaluate the feasibility and allow outside clients and customers to audio the desirability of our application before proceeding to build it.

1.3 SCOPE

"Help Me! Laurier" is an internet connected application that runs on Android smartphones that connects students with knowledgeable peers to get them the answers they need. This include academics, financial information, extracurricular activities, and student recommendations (restaurants, entertainment, etc.). Students have the option of being anonymous to contribute their questions and knowledge while retaining the right to protect their privacy. "Help Me! Laurier" deploys a distributed approach for getting the answers students need, quickly.

1.4 Terms, Acronyms and Glossary

User, Client: A user interacting with our mobile application contributing questions and knowledge to the bank of data.

Administrator: A backend super user that manages the bank of questions, monitors for spam and collects metrics about our system.

Front-End: A set of views and interface that a user will interact with, in this case the mobile application.

WiFi: A wireless signal protocol used to connect to the internet

Push Notification Server: A service provided by Google to send information to a user's device as it happens, similar to text message notifications, e-mail notifications, or notifications about your social media.

Push Oriented: A term used to describe data which is sent to users when required rather than on demand

Pull Oriented: A term used to describe data which is sent to a user when they request it.

OAuth: A process used to authenticate with open providers such as Google. Used in the application to login using student credentials.

ApplicationServer: This term is sometimes used to refer to the server component of the application, especially in use cases.

Curious User: A curious user is the specific actor involved in inquiring for content. They are still a user, but labelled as such to be specific.

Experienced User: An experience user is the specific actor involved in answering questions. They are still a user, but labelled as such to be specific.

Note on user types: For the most part, Curious Users and Experienced Users are the same. The difference lies in the contextual differences they interact within. For example, both can have profiles and follow the authentication rules. A user can be both but the actor type will change dependent on the context. For example, when an Experienced User answers a question it is pushed to the Curious User. Note that they are both users but they are labelled as such to make it clear who the targets are.

2 Overall Description

2.1 Perspective

"Help Me! Laurier" is a networked application, so the system will consist of a mobile application and a backend server that the mobile application will connect to. The actual front-end application is where the users will spend all of their time – the server simply exists to facilitate communication between the students. Users will use this front-end to view the questions of other students, receive notifications about topics they may be interested in and submit questions for other students.

The only thing required by the mobile application is a strong wireless data signal (or "WiFi") in which the application can use to periodically use to communicate with the server to retrieve question information and submit question content. Occasionally, users will receive updates from the push notification server regarding recent application activity. This functionality is all contained within the single application, so there is no need for the user to do anything outside of it.

The server side application is responsible for storing and categorizing all the content in a performant way. The server will be an intelligent program capable of crunching information, creating related question links, and sending out notifications. It is responsible for delegating all content to the appropriate channels.

On the server side, we realize our product is very content-heavy and requires a strong back-end as we are very push oriented versus pull (although, this is supported, too). This means our back end will utilize a strong server capable of storing a lot of text data in an efficient manner. Data will be indexed, time stamped and stored for easy archiving and storage. The mobile application will not interact with this database, only though the server.

For a more detailed description of each piece of the program, you can refer to the *Design* documentation which contains detailed and descriptive information on what is composed of what pieces of software and how they interact with the application as a totality. This document is meant to be brief and non-technical.

2.2 Constraints

Lack of network connection must be handled. Specifically, we need to handle the cases where every request that gets sent cannot afford to be lost. For example, when a push notification is sent to a specific user and they are not available via network connection, this push should be deferred until they are available once again. Thus, as a functional requirement one must assume that every piece of critical information sent to the user might not have an available endpoint at the time of request.

Intelligence about pushing. The selling point of the application is that it can intelligently display information to people who are interested and get the users in front of the right people. To do this without causing annoyance, the application must be smart and accurate. The application must be able to detect inactivity and find the right people for the job without fail.

2.3 Specific Functional Requirements

2.3.1 Response Time

The application must be able to send out notifications quickly and process fast. Long running algorithms to decide where content will be passed around are not acceptable. When a question is submitted, it must be determined who can potentially answer within **one minute.**

2.3.2 Anonymity

This does not affect the user experience but it must be noted that while anonymity is kept where possible, users that could be identified as a threat to the system must be able to be identified. *This means the server should keep track of who posted and answered what, in case of court order or authority.*

2.3.3 Bandwidth Mindfulness

This application runs on mobile devices such bandwidth consumption must be reduced to a minimum in order to achieve this. This means reducing the amount of headers and data sent on each request and reducing the amount of data that must be sent. This is partially done by using the pull model rather than pull, but is also achieved by keeping our data send between endpoints as slim as possible.

3 Use Cases of the Application

3.1 Scenario: Asking a question

Entry Condition: The user has tapped "Ask a question" and thus made the request. The user must have proper authentication at this point in time.

Description: The user has thought of a question they want to ask the student peer advisors and wants to use the application to ask a question. They have loaded the application and have arrived at the main menu screen.

Actor(s): CuriousUser, ApplicationServer

Flow:

1. Tap the button on the screen that indicates "ask a question"
2. User is prompted for a question type consisting of: academic, school related and social categories.
3. Depending on the selection, the user will be presented with more sub types
   a. Selecting "Academic" presents a list of courses and the option for a user to filter and select one.
   b. Selecting "School Related" presents a list of possible subcategories such as "Financial" or "Organizations"
   c. Selecting "Social" presents a list of school clubs and other recommendations.
4. Once the student has entered a sub category it prompts them to ask a question. While typing, they will be presented with some potential similar questions.
   a. If they select one of these similar questions, the use case will be aborted and use case **"Check an answer"** will be initiated.
   b. Otherwise, the user will continue to type their question and flow will continue.
5. After the user has entered their question, they may select "Anonymous" which will hide their identity from the question before selecting **submit**.
   a. Even if a user selects anonymous, user information will be recorded for safety and legal reasons.
6. At this point, the user will be notified that their question has been submitted successfully. They will be returned to the main menu.
7. The **ApplicationServer** actor will be notified of this submission and the use case "**Distribute a question"** will be activated.

Extensions:

1. At any time, if the user has decided they do not want to ask a question anymore, they may simply use the cancel utility provided to do so.

Exit Condition: The user has successfully asked their question and it has been posted **or** the user cancels the process by utilizing the built in cancel function.

3.2 Scenario: Answer a question

Entry Condition: The user has tapped “answer a question” and thus made the request. The user must have proper authentication at this point in time.

Description: A user is currently viewing a question in the application and has decided to answer it. They will be answering within the application.

Actor(s): User, ApplicationServer

Flow:

1. If a student chooses to respond to a question, they may select “Answer”
2. The user will be presented with a textbook and some minor editing utilities, like the ability to attach a picture.
3. After entering their response, the user will optionally select “Answer Anonymously” and then touch “Submit”
    a. Even if a user selects anonymous, user information will be recorded for safety and legal reasons.
4. At this point, the user will be notified that their answer has been submitted successfully and they will be returned to the question.
5. The **ApplicationServer** actor will be notified of this submission and then the use case “**Push an answer”** will be activated.

Extensions:

1. At any time during the use case execution, the executing actor may choose to use the cancel utility provided to terminate execution and thus not post an answer.

Exit Condition: The use case will successfully end when the user has posted an answer successfully and the server has acknowledged it **or** the user has used the cancel extension to terminate answering the question.

3.3 Scenario: Check a question

Entry Condition: The user has selected an action that would require viewing a question and thus made the request. The user must have proper authentication at this point in time.

Description: A user has initiated an action to view a question – either from a notification of some other part of the application that has invoked this use case.

Actor(s): User, ApplicationServer

Flow:

1. The user is presented with the question title, description and a list of answered, in the order they have arrived by default; then sorted by satisfaction
    a. Each question has two buttons: "**Satisfactory"** or "**Unsatisfactory"**
        i. If the student presses satisfactory a push is sent to database to move the answer up.
        ii. If the student presses unsatisfactory a push is sent to database to move the answer down.
2. The user may be prompted with other options for the question, such as "report" or other administrative utilities.

Exit Condition: If the user UI with all the question interface has been successfully loaded, the use case will exit and the user will be able to interface with the UI to perform additional use cases.

3.4 Scenario: Login and Setup

Entry Condition: The user has entered the application for the first time – this use case runs immediately.

Description: A user has just downloaded and installed the application and wants to get started. This use case is invoked the first time the user launches the application and has not yet completed the entire use case flow.

Actor(s): User, ApplicationServer

Flow:

1. The user will be presented with a “Welcome Screen” that will display some brief information about the application and how to get started.
2. After dismissing the above welcome, the user will be prompted for a **Google Account Login.**
3. The user will select their ”MyLaurier” e-mail and authenticate using **Google OAuth**
    a. If the user selects and authenticates with a non-Laurier e-mail, they will be asked to repeat this step.
    b. Similarly, invalid credentials will result in repeating this step until correct.
4. Upon successful authentication, the user will be moved to the main menu and the use case terminates.

Extensions:

1. If at any point the user cannot authenticate properly, they may say so using the “Cancel” function and return to Step 1 of the use case allowing them to try again with different user credentials / usernames.

Exit Condition: The user has successfully authenticated and logged in. The user will exit this use case and it will not flagged for invocation again.

3.5 Scenario: Check notifications

Entry Condition: The user has received a notification from a source (most likely their **notification tray)** and has selected it. They want to view their notifications.

Description: A user wishes to view their notifications to see what has changed since they last used the application.

Actor(s): User

Flow:

1. From the main menu, the user will select “Notifications” to be brought into the Notifications window.
2. From here, the user will be given a list of notifications they can view. They’re divided into several groups:
   a. “Answers for you” is a heading which will contain notifications regarding things you may have asked
   b. “Questions for you” heading containing notifications about questions that the user should be able to answer with their experience
3. A user may select either type of notification to be brought to the linked page – in either case selecting a notification will invoke the use case “**Check a question**” for the appropriate question page.

Exit Condition: The user will see a UI that represents the flow described above. They may interact with the UI to invoke other use cases.

3.6 Scenario: Distribute a question

Entry Condition: The server has received a question and needs to distribute it. This is invoked when the server has received and began processing.

Description: The server has received a question from a source and needs to distribute it to users who may be able to answer it.

Actor(s): ApplicationServer

Flow:

1. The server has received the question and is ready to distribute it.
2. If it has not yet recorded the question, it will connect to the database and record it.
3. A subset of users will be selected that are most likely able to answer this question based on a number of factors, including but not limited to: experience, answer history, satisfaction, and initial setup.
    a. If a subset cannot be found, the use case terminates immediately.
4. The selected subset will receive a push notification which when clicked will invoke the use case **"Check Notifications"**
5. A timer will be set for a later interval to re-execute this use case for the same question If an answer has not been received in the given interval. A different subset will be scheduled for this time to avoid double push notifications.

Extensions:

1. If a user is not available for some reason (such as network connectivity issues or the phone is off), the process should be suspended and start back at **1** when the user is available again. This is to satisfy the functional requirements.

Exit Condition: The server has sent the question successfully **or** the state has been suspended and queued again.

3.7 Scenario: Push an answer

Entry Condition: The server has been sent the answer and is ready to distribute this.

Description: The server has received an answer from a source and needs to distribute it the user who has asked.

Actor(s): ApplicationServer

Flow:

1. The server has received the answer and is ready to distribute it.
2. The question in which the answer belongs to will be looked up and then the answer attached and recorded in the database.
3. The user who asked the initial question will then be sent a push notification, alerting them their question has a new answer.

Extensions:

1. If a user is not available for some reason (such as network connectivity issues or the phone is off), the process should be suspended and start back at **1** when the user is available again. This is to satisfy the functional requirements.

Exit Condition: The server has sent the answer successfully **or** the state has been suspended and queued again.

3.8 Scenario: Update profile

Entry Condition: The user selected the "Profile" icon and thus made a request to the server to change some profile data.

Description: The user wishes to get more accurate answer candidates and has decided to update their profile to improve the accuracy of the application.

Actor(s): User**,** ApplicationServer

Flow:

1. The user will be presented with some options and various headings. Such things could be:
   a. **Academic Information:** This contains information regarding courses, your expertise level and comfort level, and enrolment status
   b. **Organization Affiliation:** This contains questions about your organizational affiliations
   c. **Extracurricular Affiliations**
2. The user will select one of these and then proceed to answer questions about their various associations. Exact layouts can be found in preceding documents.
3. When a user is satisfied with their changes, they can click "Submit" to have them persisted to the server and reflected in all stages of the application immediately.

Exit Condition: The user has decided they are content with their changes and have clicked "Submit" then this use case terminates.

3.9 Scenario: Reporting a question

Entry Condition: The user is viewing a question and has selected “Report” as they have decided they have been inappropriate material for the application.

Description: The user wishes to report a question for being outdated, old, or inappropriate (spam) or abuse.

Actor(s): User**,** ApplicationServer

Flow:

1. The user will click the “Report” bottom from the “Check a question” screen and see options for reporting the question and/or various answers.
2. When the user selects this option, they will be prompted with reasons as to why this question needs to be reported:
    a. Dealing with abusive material
    b. Spam
    c. Outdated information
3. After the user identifies the reason, they click submit and the administrators will be notified to take action accordingly.
4. The user returns to the question view.

Extensions:

1. If the user changes their mind at any time, they may use the cancel functionality to do so that is provided as a “Cancel” button.

Exit Condition: The user has successfully submitted a report abuse **or** the user has decided they no longer want to submit by making use of the *Cancel* ***extension.***

3.10 SCENARIO: CONNECTION DOWN

Entry Condition: The user has initiated a request to the serer but their connection is down. This use case interrupts them.

Description: A user is using the application when the network connection is lost or the **ApplicationServer** could not be reached. The user is then presented with a notification informing them of the connection status and that there attempt has failed

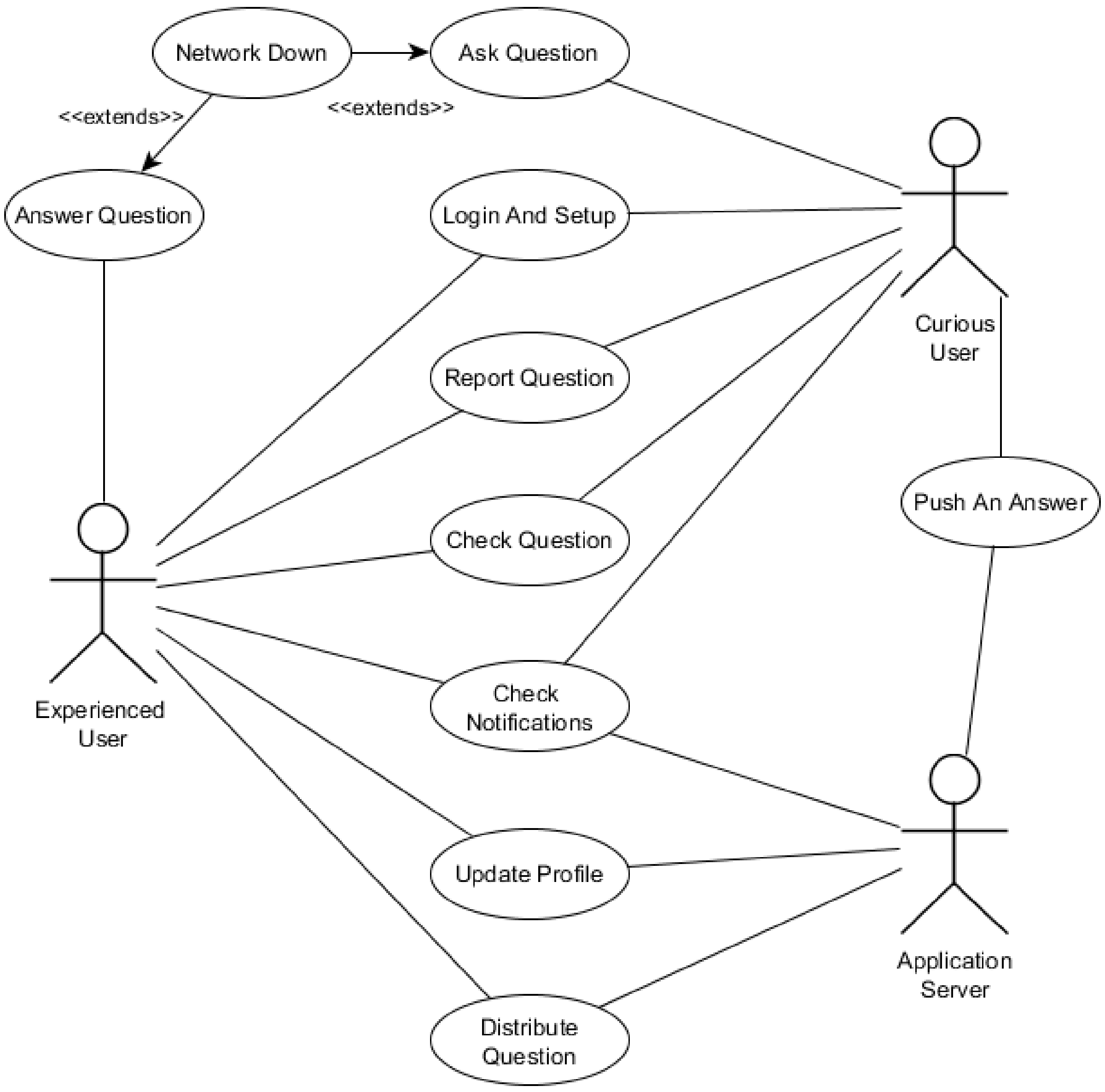
Actor(s): User, ApplicationServer

Flow:

1. The student attempts to retrieve information or notifications from the **ApplicationServer**.
2. The user is not connected to the network or cannot reach the server due to a connectivity issue.
3. The user will be presented with a toast/notification informing them that a connection could not be made and an error code describing the situation in more detail.

Exit Condition: The connection is restored at some point or the user aborts the request.

3.11 USE CASE DIAGRAM



4 Team Organization and Time Management

4.1 Development Organization

The proposed method for developing the software is a model that is similar to the popular *Waterfall Method.* We will be developing requirements, specifications, design and similar documents which will be signed off by a separate party, our research professor. However, due to the scope of the product the formal SQA team will be our *professor.* After each step, we will have the documents verified before proceeding to the next step to ensure our software is as stable and well-engineered as it can be. This will involve revision before proceeding to the next step, after each step.

Each step will be as a group to maximize the talent of each member as each of us is well rounded individuals with exposure to a lot of different subsets of technical skills. Maximizing them through a potential “phase leadership” system will be difficult, so the team will be using bi-weekly (read: twice a week) meetings to organize information and sync up accordingly. By doing this, the product will be well known by each developer while maximizing our productivity output.

For a more detailed documentation of everything completed for each team member, see the detailed *Revision History* in **Appendix B**. Each developer will have done work outside of the meetings which were coordinated together. For more specific information, refer to **Appendix C.**

4.3 Skill Matrix

The following matrix breaks down the skill levels and interest of our developers, to help organize our team better.

Task & Participants	Vaughan Hilts	Brandon Smith	Colin Gidzinksi
User Experience Design / UI (including some graphics design)	✓ Interested ✓ Secondary Experience	✓ Interested	
Database Schema Design & Support	✓ Interested ✓ Secondary Experience		✓ Secondary Experience
Configuration Management			
Client Side Frameworks & Implementations	✓ Primary Experience	✓ Interested	✓ Interested ✓ Primary Experience
Server Side Frameworks & Implementations	✓ Interested ✓ Secondary Experience		✓ Interested ✓ Primary Experience
Security, Authentication and Auditing			✓ Interested ✓ Secondary Experience
Software Testing	✓ Interested ✓ Primary Experience (formal training)	✓ Interested	
Documentation		✓ Interested	
Communications (reporting)	✓ Interested		
Software Architecture Design	✓ Interested		

Analysis

5 INTRODUCTION

In the proceeding sections we will flesh out the concrete entities, boundaries, controllers and application specifics in a manner that gives greater insight on the more granular operations of the application.

To do this, we must define all the objects we will be using at a high level and the entire spectrum of objects will be using throughout the application. Below, you will see a breakdown of the different pieces of the application, how they relate, and how they will interact with each other.

The document tries to provide a high level documentation of all the entities, boundaries, and control objects at an abstract level. The document will not try and be exhaustive but only provide a basic framework of everything required to get a deeper understanding of the application. This means all critical entities and control objects will be defined but not every single object required will be noted.

For a more detailed explanation of every object, attributes, methods and more technical documentation see the ***Design*** document.

6 Identifying Concrete Objects

6.1 A brief notice on actors

The various user actor variations explained: The requirements and some of the documentation will directly refer to the actors **Curious User and Experienced User.** These are both represented as a persistent **User** entity within the system application, but to make it clear what they represent, they have been separated in the use cases. A user can be both with one **User** entity but only one at a time.

6.2 Entities

Below, you will find a brief overview of every entity in the application without too much detail to save time in getting a general overview. **Note:** Each entity which is similar to another (read: inherits) is placed a subheading to group them together. For a very loose hierarchy of this, see the table of contents.

Note: Where applicable, a quick brief overview of potential attributes will be listed.

6.2.1 User

The persistent representation of our end user in the application is shown as an actor throughout the application and use cases. The user will typically submit **Questions**, **Answers**, update profile data and generate the content for the application. Users may also generate various types of reports through specific actions, as described below. Users are composed of names, descriptions, and important profile data. They also contain lists of questions and answers that belong to them.

User
+ UserId + QuestionsAsked + AnswersProvided + State + Score

6.2.2 Submission

A submission is a piece of content on the application. It is typically modelled as a **`Question`** or **`Answer`**, but it could be other things down the road such as comments. For those familiar with internet forums, it can be seen analogous with a forum post in many cases. They contain information, usually textual that describe useful information to be conveyed to a **`User`.**

Submission
+ Id + AuthourId + Body + Status + Score + Date

6.2.2.1 Question

A **`Question`** is a specific piece of content a **Curious User** actor will ask, which is tied to a **`User`** entity. For each question posted by the application by a single user, there will be a single entity of the type **`Question`** mapped to it. Questions can store bodies, scores and states.

Question : Submission
+ Title + CategoryId

6.2.2.2 Answer

An **`Answer`** is a specific piece of content (**`Submission`)** an **Experienced User** actor will usually submit. These are also tied to a **`User`** entity, similar to the previous question entity. Each answer contains a score, body and a state. In fact, the only distinguishing feature between a **`Question`** and **`Answer`** is the various states and business logic that can be applied to them.

Answer : Submission
+ ParentQuestionId

See *State Diagrams* to get a better idea of the various states the different entities can be split into.

6.2.3 Notification

A **`Notification`** is an entity that represents the state for a given possible event in the application. Typically, when a new **`Answer`** `or` **`Question`** has been pushed to the **`User`**, they will receive an updated entity that represents this particular entity. The **`Notification`** entity encompasses all the basic information a **`User`** needs to know about the event at a glance, without actually interacting with the said event.

Notification
+ Id
+ Type
+ Description
+ Title
+ Date
+ UserId
+ Status

6.2.4 ProfileQuestionEntry

A **ProfileQuestionEntry** represents and stores information that a **User** has voluntarily provided on their profile page in the application. This is persistent information the application will use to assess items such as submission suitability and what kind of **Notification** entities are generated for the user.

ProfileQuestionEntry
+ Id
+ Title
+ Description
+ Value

6.2.5 AbuseReport

Represents an abuse case raised against a **Question** or **Answer**, often generated by the use cases "**Reporting a Question"** and "**Reporting an Answer",** that is persistent and required to be handled by the application. Tied to a specific **User** and stored externally from the application to be handled. Actions performed due to an **AbuseReport** will affect both this entity and the **Question** or **Answer** it has been raised against.

AbuseReport
+ Id
+ Reason
+ Description
+ ReporterId
+ SubmissionId
+ Status

6.2.6 AuthenticationToken

The **`AuthorizationToken`** is a very specific type of entity that is only used for storing authorization requests and tokens for **OAuth** providers. We require keeping this around to be able to authenticate a **`User`** with their WLU account via **OAuth** providers like Google. For more information regarding tokens, check the official Google API:

https://developers.google.com/accounts/docs/OAuth2

AuthentcationToken
+ Id
+ OwnerId
+ Hash
+ ExpiryDate
+ IssueDate

6.3 BOUNDARIES

In our analysis, boundaries are considered elements that our actors will interact with. For the most part, this is simply the **Curious User** and **Experienced User.** Boundaries typically are things like frames, pages, buttons, and other elements a user will touch, slide, view and transition into.

Note: For brevity sake, the suffix "Boundary" has been omitted on most boundaries in the proceeding sections.

6.3.1 LoginPage

A generic page used to display information about logging in to the **`User`**, contains many sub boundary objects which may be important.

6.3.1.1 GoogleAuthenticationButton

A button that is used to initiate the authentication step of the **Login and Setup** use case. When pressed, users will be presented with authentication options.

6.3.2 WelcomePage

A page that displays introductory information regarding the application to a first time **`User`** to help them get accustomed. If this page is shown, it is because the use case “**Login and setup”** has just been invoked.

6.3.2.1 DismissButton

A button that is used to dismiss the dialog, WelcomePage, and let the “**Login and setup”** use case continue on in flow, leading to the **`LoginPage`.**

6.3.3 ParentFrame

A frame that encompasses various boundary controls, mostly pages. This boundary control is almost always visible during the application, except when then the **`WelcomePage`** and **`LoginPage`** are within view. It houses buttons such as…

Header

6.3.3.1 ViewNotificationsButton

When this button is touched, the use case “**Check notifications**” will be invoked and the **`User`** will be able to check their notifications.

6.3.3.2 HomeButton

When this button is touched, the active page in the application will switch to the **`HomePage`.** No particular use case will be invoked.

6.3.3.3 EditProfileButton

When this button is touched, the active page in the application will switch to the **`ProfilePage`.** Changes to specific **`User`** profile information can be changed here.

6.3.4 HomePage

A page that has some basic tools and utilities for viewing questions, asking them and sorting question content within the application is shown here. More interestingly are the controls housed on this page.

6.3.4.1 AskButton

When touched, the use case "**Ask a question"** is triggered and the user will be put through the flow of events.

6.3.4.2 ViewQuestionsList

When interacting with this control, the application will transition and allow viewing of previous questions that have been stored and archived.

6.3.5 NotificationPage

A page used to view and read notifications, when this page is shown the "**Check notifications**" has been invoked. Events on this page will flow according to that use case.

6.3.5.1 QuestionsForYouList

This is a list of **`Notification`** entities that have been sent to the client regarding questions they may have the proper expertise to answer. This boundary will be responsible for rendering the appropriate type of notifications that correspond to these and handling transition.

6.3.5.2 AnswersForYouList

This is a list of **`Notification`** entities that have been sent to the client regarding questions that have been answered by an **Experienced User** actor in another use case. This boundary will be responsible for rendering the notifications and handling transition.

6.3.6 SubmissionPage

This page is used to complete the "**Answer a question**" use case and "**Asking a question"** use case. Both require a form to enter information – and this page houses this form and the various controls. They are outlined below:

6.3.6.1 TitleTextBox

A simple textbox control that will map the **`Question`** *title* property with the user input.

6.3.6.2 DescriptionTextBox

A simple textbox control that will map the **`Question`** *body* property with the user input.

6.3.6.3 FormattingWidget

A widget that can control formatting options and mutate the *body* property accordingly; formatting details are not important at this stage.

6.3.6.4 SubmitButton

A button that when pressed can dismiss the **`SubmissionPage`** and return the results to the use case that required it.

6.3.7 ViewTopicsPage

Generally, a page that will simply list questions which are generated from **`Question`** entities for the end user to view and visit.

6.3.8 ViewQuestionPage

A page that is responsible for rendering a **`Question`** and by consequence, all of the associated **`Answer`** entities attached to it. When viewing this page, the "**Check a question**" use case has been invoked. These are done in the form of cards, described below:

6.3.8.1 SubmissionCard

Each contains information regarding the submission – body, date, score, and other attributes outlined on the according entity. Each card also contains a **Report** button which enables a **`User`** to generate an **`AbuseReport`** by invoking the "**Report abuse"** use case when touching it.

6.3.9 ReportQuestionPage

A page that is responsible for handling the "**Reporting a question**" use case. It contains many fields pertaining to things like: reason for closing, description of the issue, and priority.

6.3.10 ProfilePage

A page that is responsible for handling the "**Update a profile"** use case. This page will typically handle a stream of questions that allows updates to a **`User`** entity to be performed to have more pertinent **`Notification`** entities generated.

6.4 Control Objects

In the below section, we outline a brief description of each of the extracted controllers that will be required for the development of the application. Each has the responsibilities it will perform briefly outlined and then some of the entity objects it may perform on. Details are reserved for the “Design” section that can be found later in this document. You can use the table of contents found at the beginning of the document to navigate the controls and find their appropriate implementation details further in.

6.4.1 Server Control

A basic controller that is responsible for delegating and interfacing with the asynchronous application server link. Most boundaries and other controllers that require sending information and fetching it will usually pass through this controller at some point to allow delegation of effort. The server control often can answer requests from various other controllers to retrieve remote entities, such as **`Users`**, **`Question`** and **`Answer`**.

6.4.2 Authentication Control / OAuth Control

This controller has full responsibility for interfacing between the entities: **`User`** and **`AuthorizationToken`** and the **`LoginPage.`** Additionally**,** the controller provides services to communicate with the remote server that **Google Inc.** has provided to authenticate the WLU **Google Apps** domain.

6.5 AnswerControl

The answer control is responsible for tracking and modifying **`Answer`** entities and delegating them around the application for various boundaries and controllers to consume. While it will typically interact with local **Answer** entities for the local side, it can also retrieve remote entities by interfacing with the **`ServerControl`**.

6.6 QuestionControl

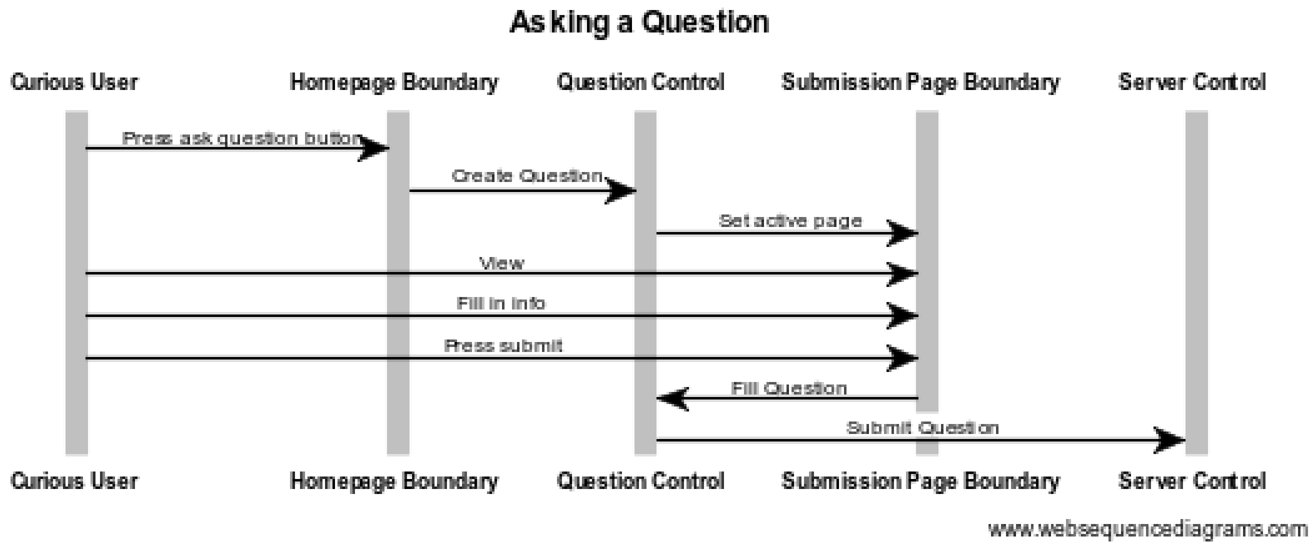
The answer control is responsible for tracking and modifying **`Question`** entities and delegating them around the application for various boundaries and controllers to consume. While it will typically interact with local **Question** entities for the local side, it can also retrieve remote entities by interfacing with the **`ServerControl`**.

6.7 NotificationControl

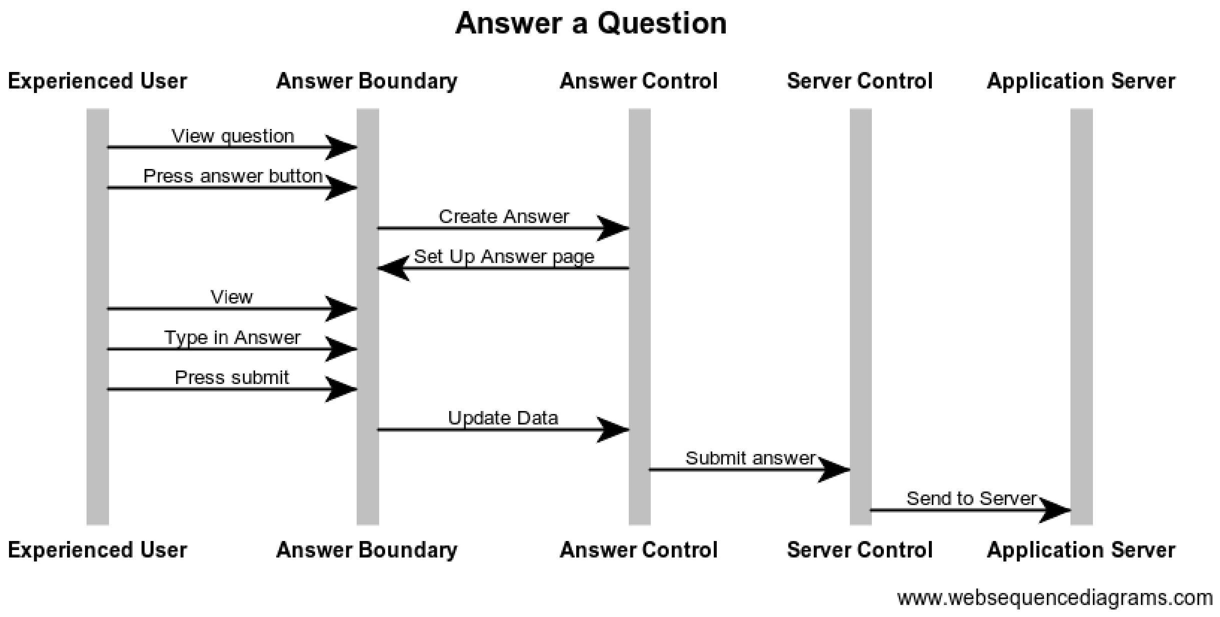
This controller is used to delegate the processing of notifications between the applications. Two specific versions of this exist: the client and server version. Both of these versions have similar responsible. They handle the operations of **`Notification`** entities and tracking throughout the services. The client version is used to track notification on the screen and pushing to the user's notification tray. On the server side, the notification control is responsible for generating and sending the **`Notification`** entities to the various clients that they belong to.

7 Use Case Sequencing Diagrams

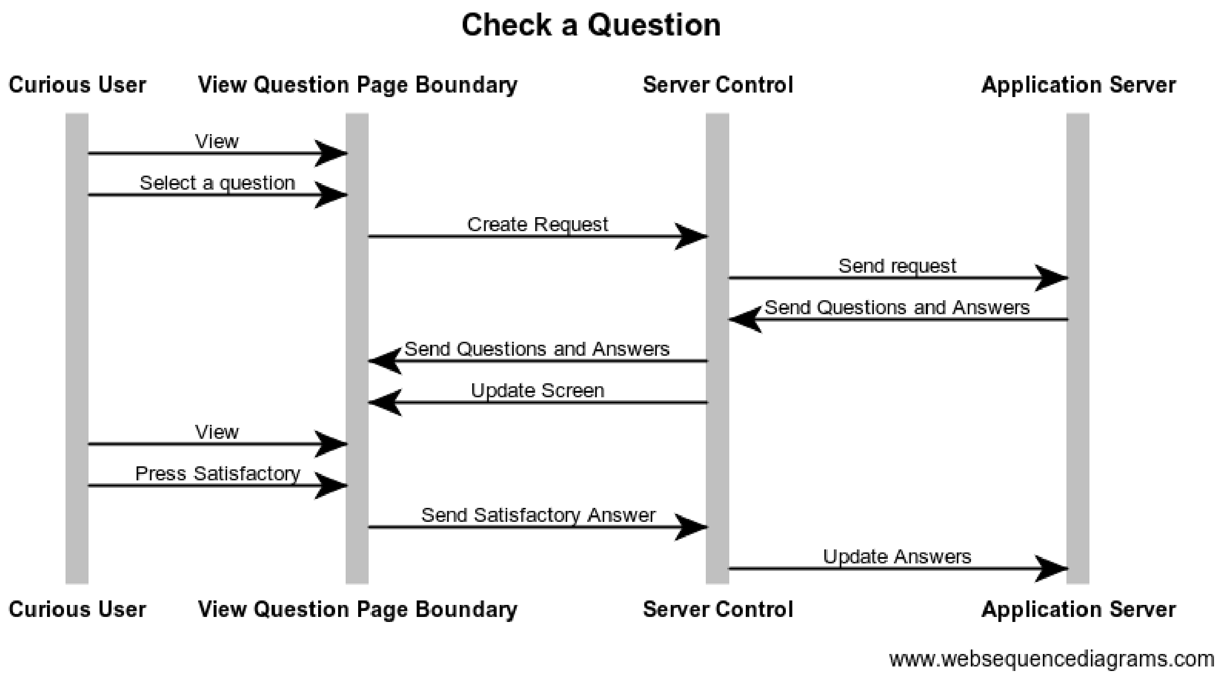
7.1 Use Case Sequence: Asking a question



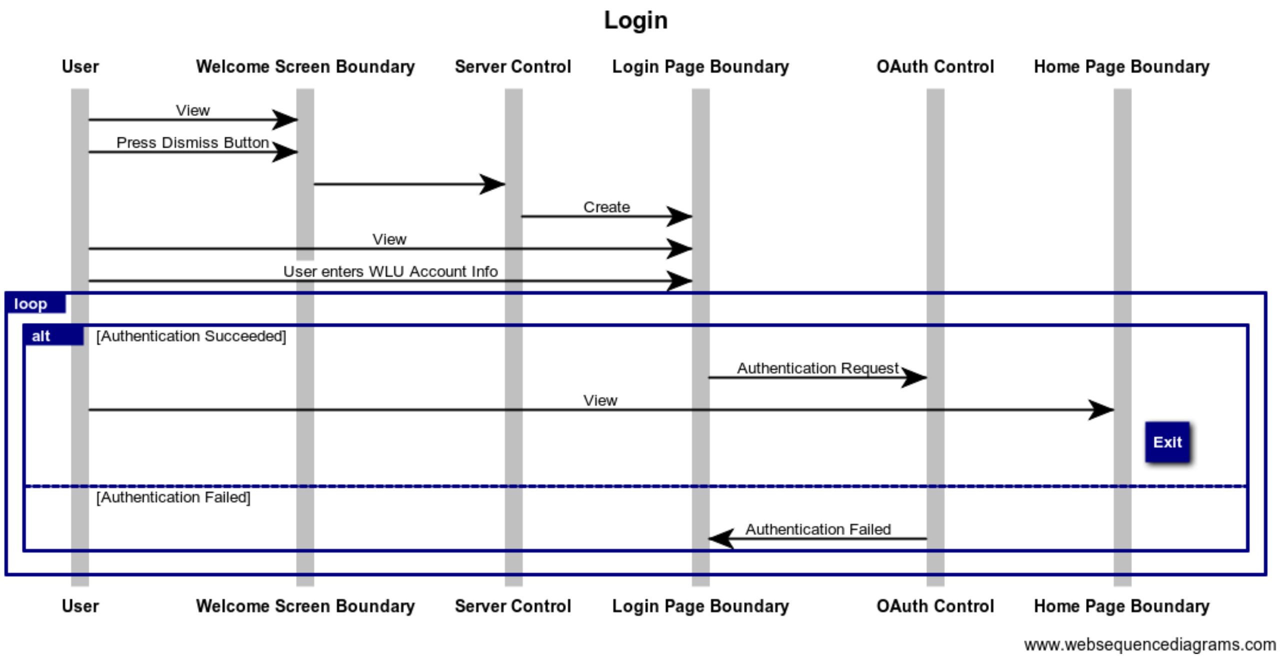
7.2 Use Case Sequence: Answer a question



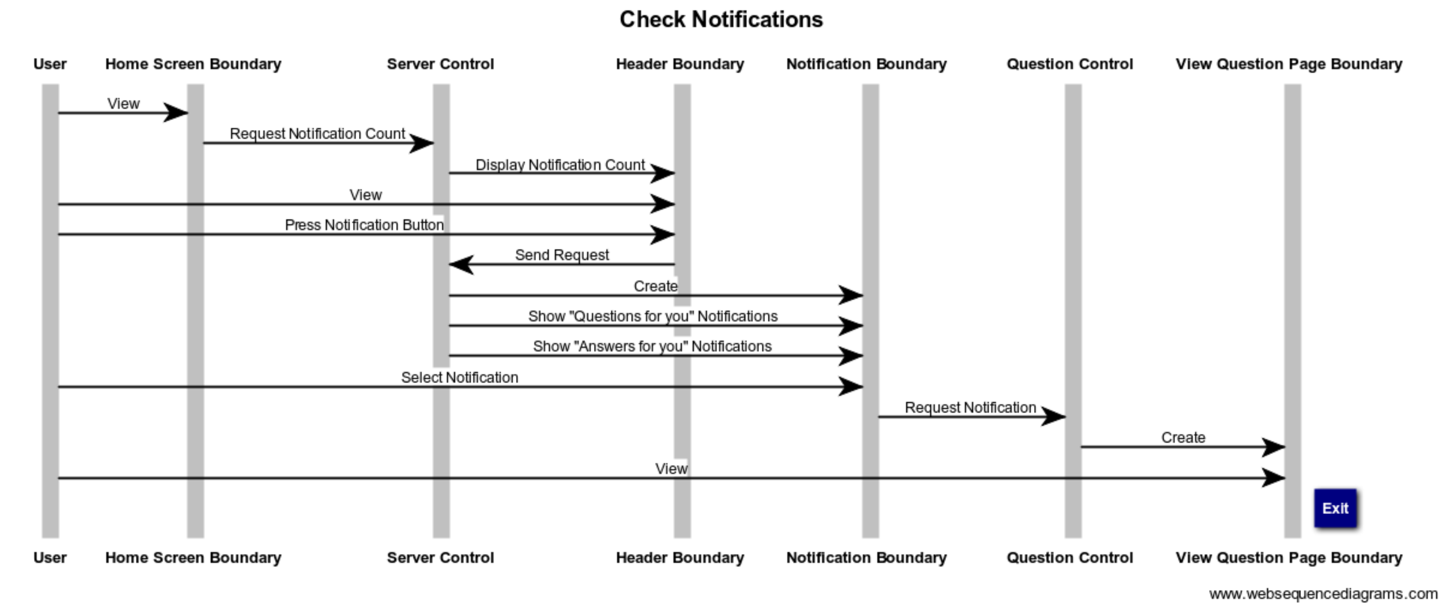
7.3 Use Case Sequence: Check a question



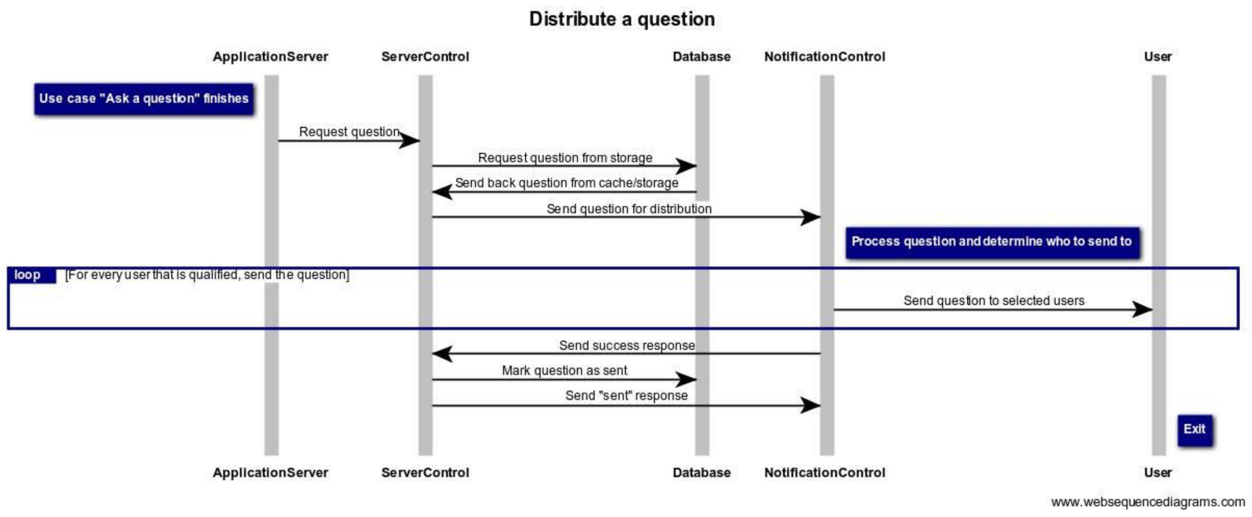
7.4 Use Case Sequence: Login and setup



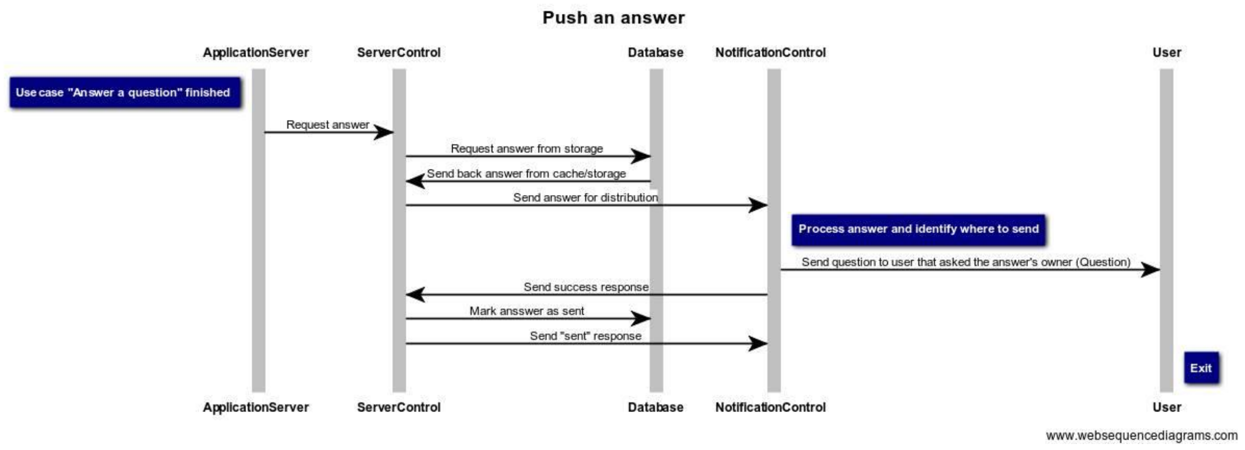
7.5 USE CASE SEQUENCE: CHECK NOTIFICATIONS


Check Notifications
User
Home Screen Boundary
Server Control
Header Boundary
Notification Boundary
Question Control
View Question Page Boundary
View
Request Notification Count
Display Notification Count
View
Press Notification Button
Send Request
Create
Show "Questions for you" Notifications
Show "Answers for you" Notifications
Select Notification
Request Notification
Create
View
Exit
www.websequencediagrams.com

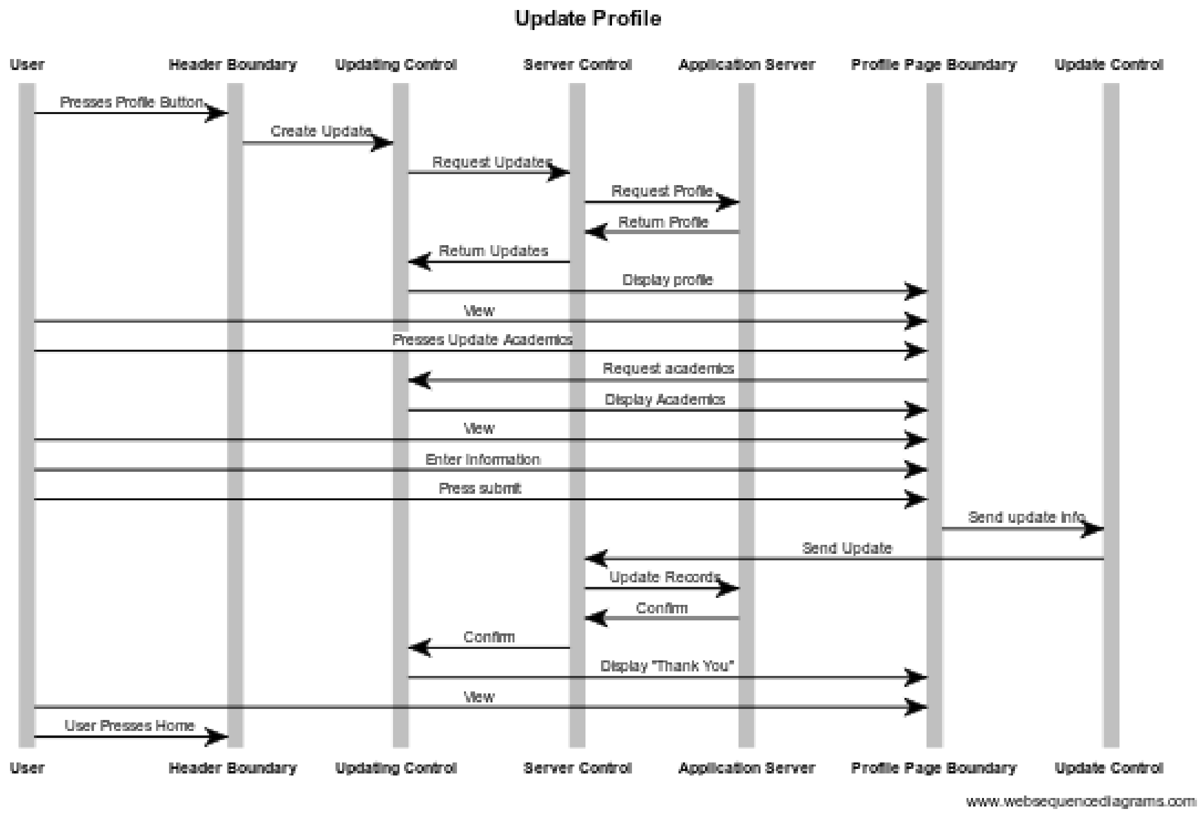
7.6 USE CASE SEQUENCE: DISTRIBUTE A QUESTION


Distribute a question
ApplicationServer
ServerControl
Database
NotificationControl
User
Use case "Ask a question" finishes
Request question
Request question from storage
Send back question from cache/storage
Send question for distribution
Process question and determine who to send to
loop [For every user that is qualified, send the question]
Send question to selected users
Send success response
Mark question as sent
Send "sent" response
Exit
www.websequencediagrams.com

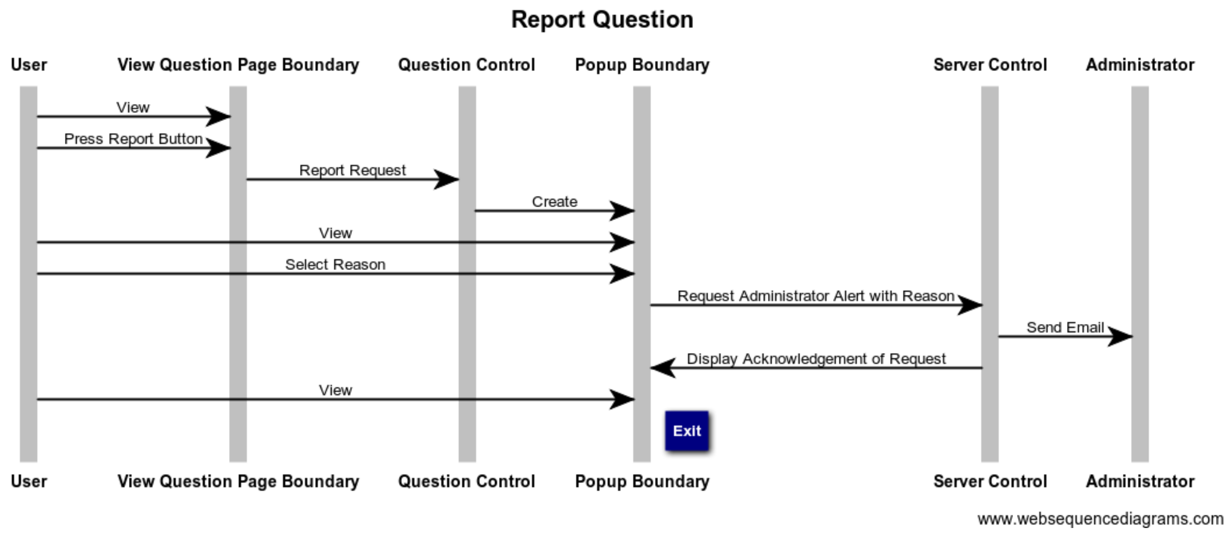
7.7 USE CASE SEQUENCE: PUSH AN ANSWER


Push an answer
ApplicationServer
ServerControl
Database
NotificationControl
User
Use case "Answer a question" finished
Request answer
Request answer from storage
Send back answer from cache/storage
Send answer for distribution
Process answer and identify where to send
Send question to user that asked the answer's owner (Question)
Send success response
Mark answser as sent
Send "sent" response
Exit
www.websequencediagrams.com

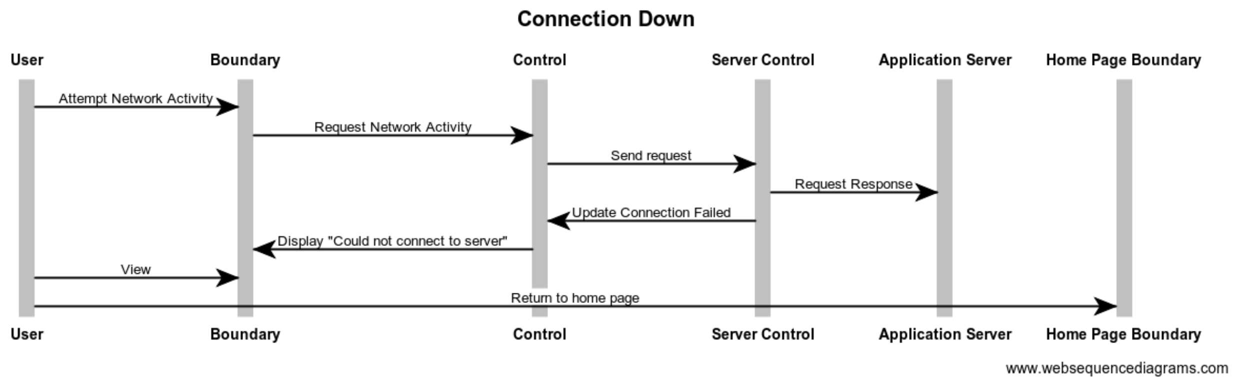
7.8 USE CASE SEQUENCE: UPDATE PROFILE



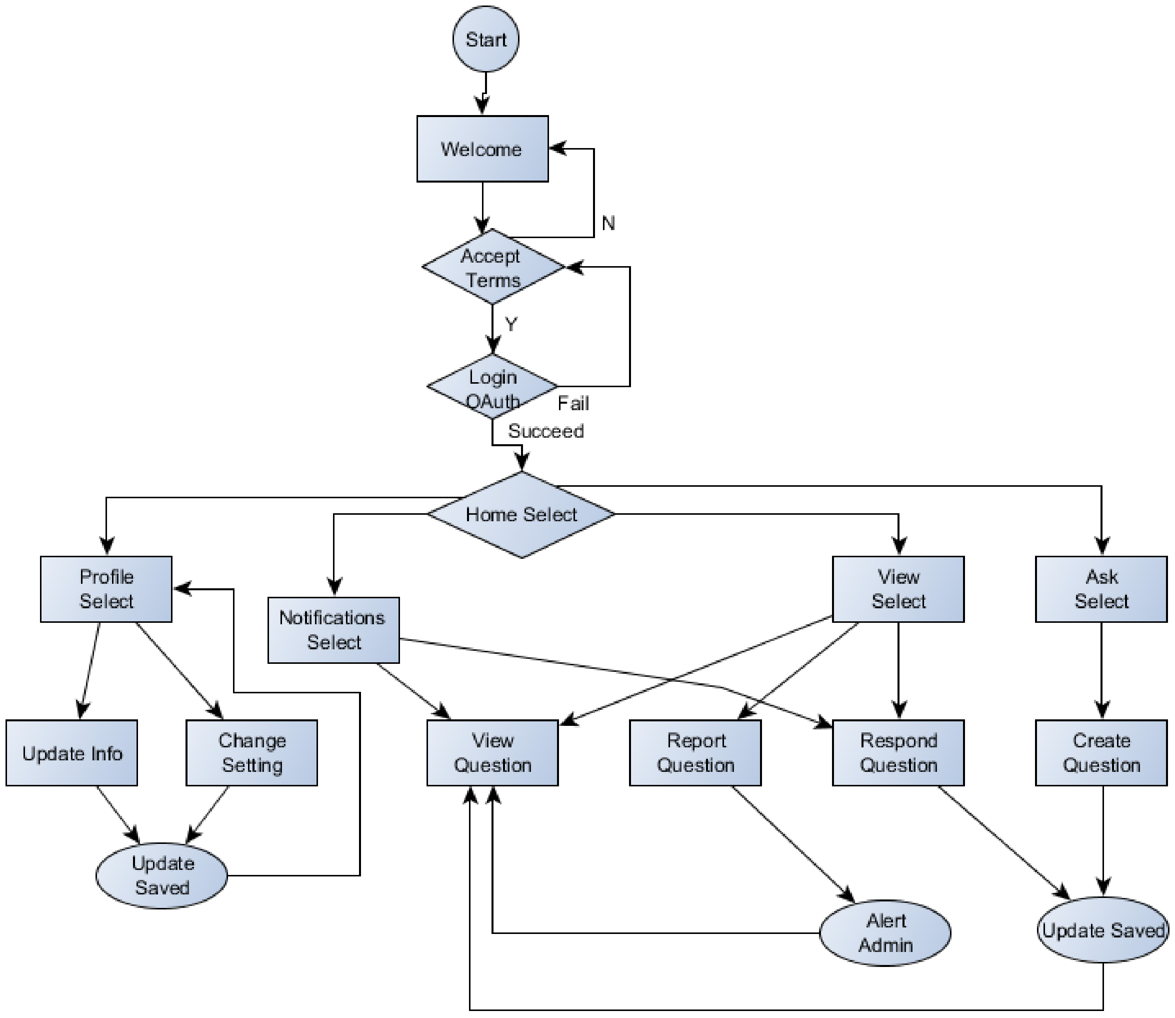
7.9 USE CASE SEQUENCE: REPORTING A QUESTION



7.10 USE CASE SEQUENCE: CONNECTION DOWN



8 ACTIVITY DIAGRAM



9 STATE DIAGRAMS

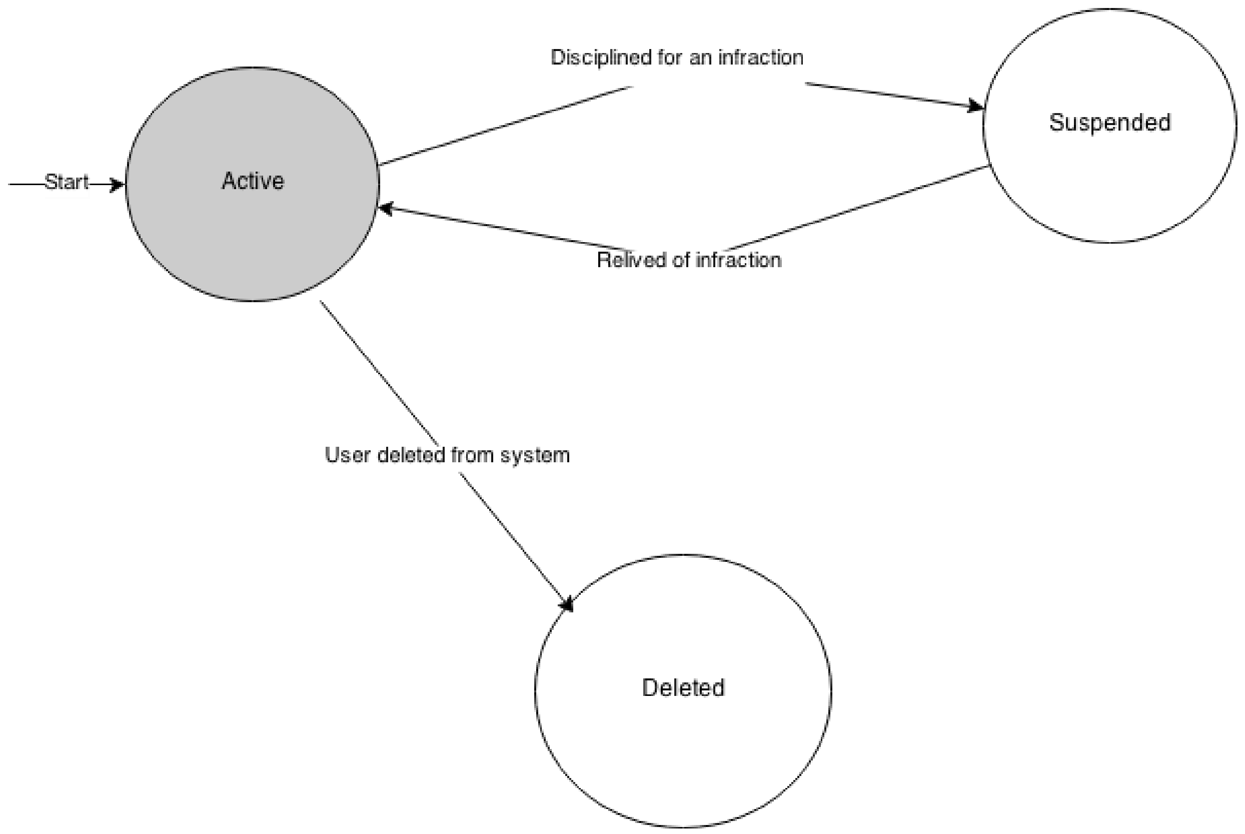
For many entities, a state diagram that summarizes potential states and transitions are described below. This list is as exhausted as possible, use the table of contents to navigate accordingly.

Note: The initial state for all entities is denoted by a solid gray state bubble on the below diagrams unless otherwise noted by a use case.

For source files of the state diagrams, please check out the accompanied files in the complete report.

9.1 USER

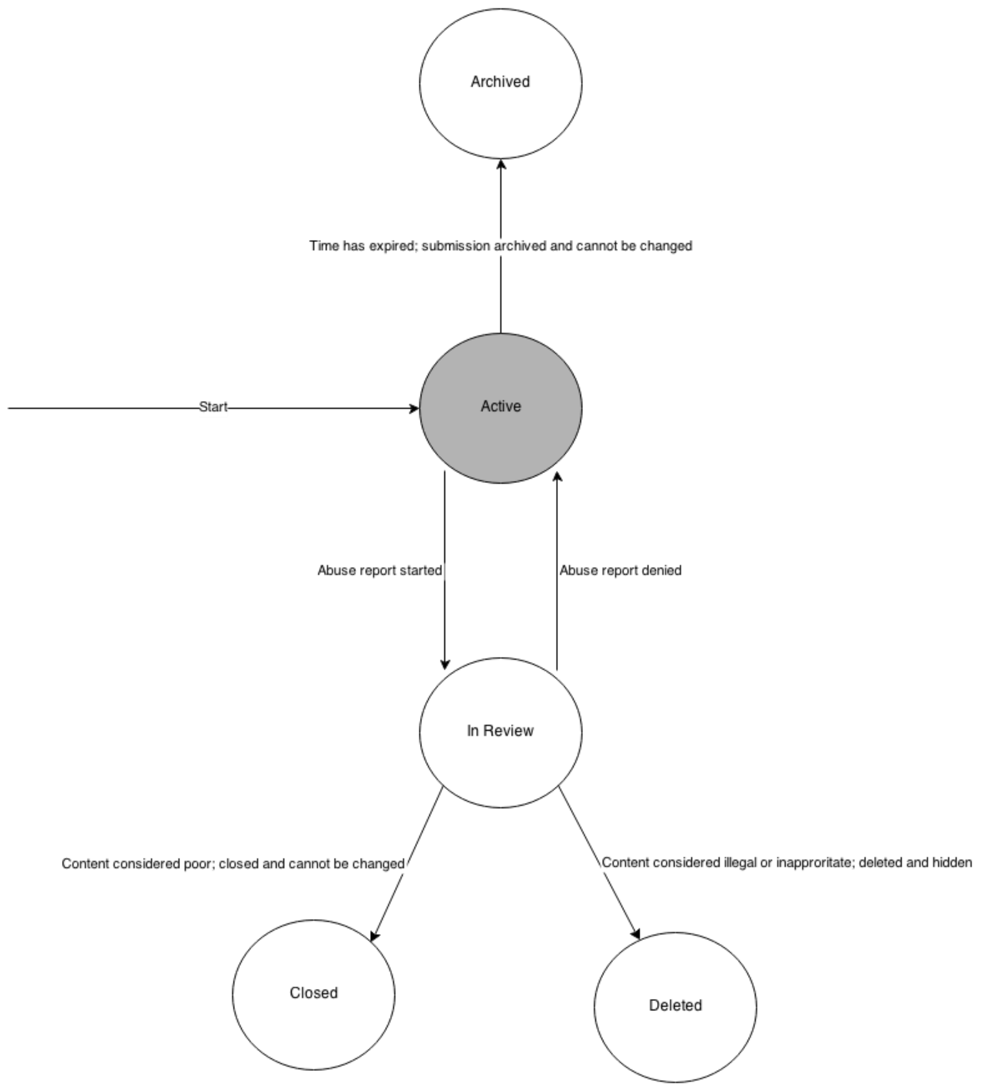
A **User** has a few states throughout the application that they can take on. Quickly, they are outlined here:



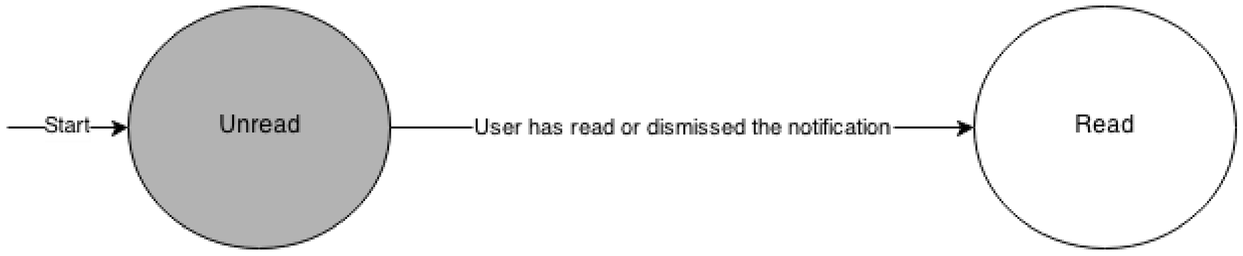
In this case, it is shown they can be **Suspended** or **Deleted** through administrative utilities.

9.2 SUBMISSION

When a **`Submission`** is created, it is generated as **Active.** After some time, they can become archived. When invoking the "**Reporting a question**" use case, a **`Question`** entity may have their status changed to "**In Review**".


Archived
Time has expired; submission archived and cannot be changed
Start
Active
Abuse report started
Abuse report denied
In Review
Content considered poor; closed and cannot be changed
Content considered illegal or inapproritate; deleted and hidden
Closed
Deleted

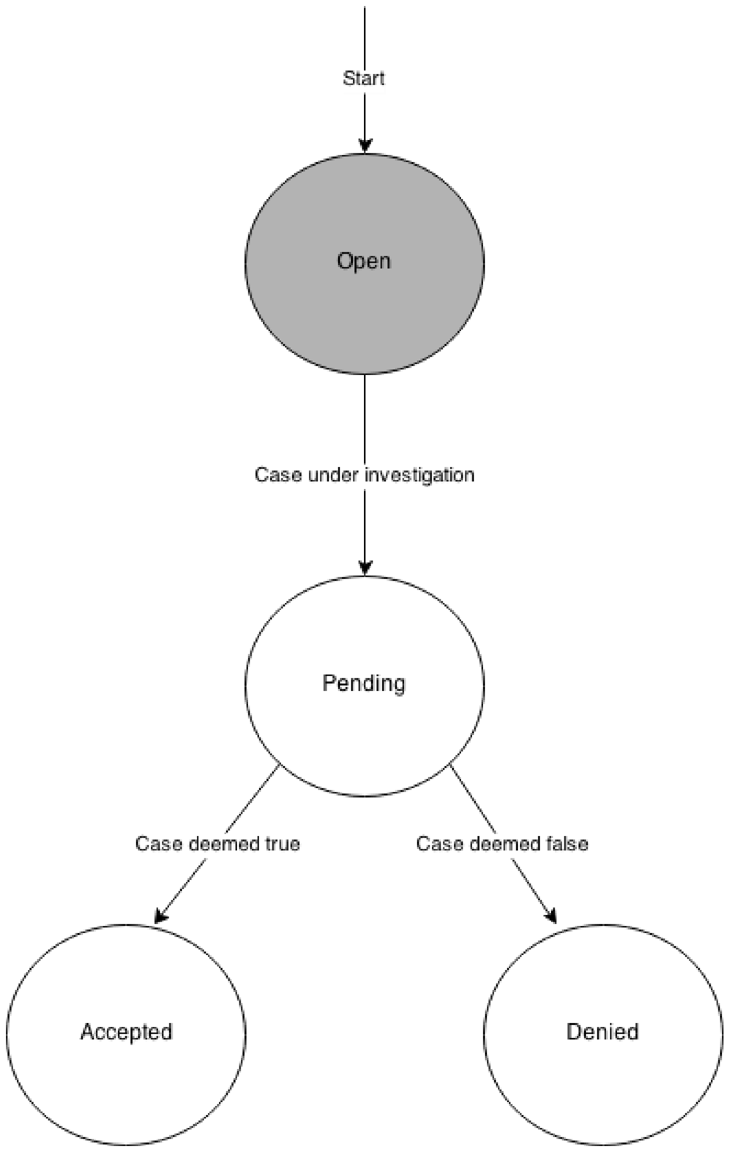
9.3 NOTIFICATION



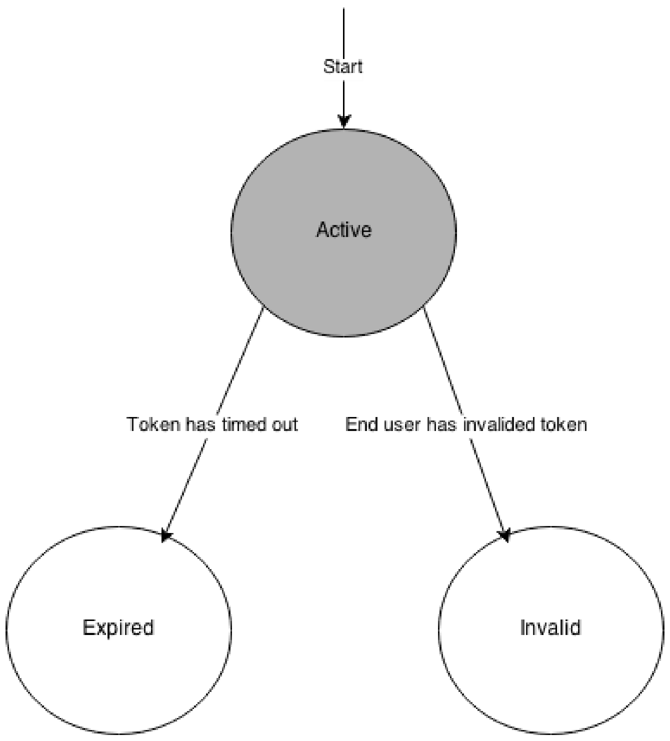
The flow for this is very basic, however in the interest of being exhaustive it has been provided.

9.4 ABUSEREPORT

When a **`AbuseReport`** is generated, it is set to **Open.** As the report is handled, the entity is moved to **Pending** and then to another state when a decision has been made.



9.5 AuthenticationToken



Design

10 Introduction

10.1 Purpose

The following document is intended to give the developers insight as to how the software is to be developed from a technical perspective. The architecture, system design, component design, and all of the technical aspects of the “Help Me! Laurier” android application will be discussed. The documentation in this section will strive to give a complete overview and ensure every developer knows the structure of the application during the construction of “Help Me! Laurier”.

It should be noted that because of this, the documentation will be **very technical** from this point on**.** If you are only interested in a broad requirements or introduction to the application, please consider returning to the *Requirements* or *Analysis* sections first.

10.2 Overview

To get a general overview of this section, see the table of contents provided at the beginning of the document. For a more in-depth look at what each subsection will contain, see the descriptions listed below:

10.2.1 System Overview

The system overview subsection gives a general description of the functionality of the application. This includes information such as: client-end choices, back-end choices, technology stack descriptions, database technology choices and a general overview of them. This subsection is designed so that a developer entering the project can read it and learn what technical skill set they will need in order to successfully contribute to the project.

While this segment does not outline every library and piece of software used throughout the system, it describes all the major components. For example, if a specific server-sided library is used for a small task such as time zone calculations, it is not mentioned. Critical components such as **AngularJS** are discussed as they are mandatory to understanding the bulk of the application.

A brief description of why certain technologies were chosen is also included along with other technologies that were considered. This allows other developers to gain insight on how the previous developers thought and why they had made the choices they did.

10.2.2 System Architecture

The system architecture provides a more detailed look at all the different subsystems and components of the application. Here, you will find decompositions of how each subsystem will interact, function and provide services to the other systems in the application. This gives a high level technical overview of the different systems in the application. If you are interested in how everything works from an overview perspective, this is where you should start.

In this subsection, you will also find some rationale for the choices that were made. This allows a developer to properly document alternatives. If a particular architectural decision does not function properly, it is important for the developers to be able to reflect on why it does not and how to correct the problems.

10.2.3 Data Design

In this subsection, we discuss the overview of all the data that flows through the application. You will find a listing of all the data the applications needs to manage, how it is managed, where it is stored, how it is processed and how it is organized. If you want to know the specifics on a how any piece of data interacts with the application, look in this subsection.

10.2.4 Object Breakdown Design

In this subsection, objects are looked at it on an individual level. The information from the system architecture overview is analyzed in a more granular and specific way than the previous subsections of this document. Check this subsection out for brief descriptions on programming UMLs, member function descriptions, and descriptive code analysis.

10.2.5 Interface Design

In this subsection, the user interface is laid out in the form of mockups and brief descriptions. This helps give a more general idea of how the application should function from a user experience standpoint. This is important to the front end developers, so that they can properly capture the essence and vision of the UX team. This subsection outlines each screen briefly and labels what each button should do, mapping it to use a case or action.

10.3 Further Reading / Technology Glossary

Before reading this document, it may be useful to list a few technical terms and information on where to read more about them as they are referred to throughout the documentation.

Apache Cordova: Apache Cordova is a set of tools designed to allow developers to package their web apps into native, easy to use packages to be distributed on various phone and tablet devices easily. (Read more: http://cordova.apache.org/)

AngularJS: AngularJS is a client side framework that makes writing **MVVW** like applications easy with **JavaScript**. (Read more: https://angularjs.org/)

Ionic Framework: The Ionic Framework is a high performance framework based on ***AngularJS*** and ***Cordova*** to create like-native applications using the greatest web application. (Read more: http://ionicframework.com/)

SASS: SASS is an extension of the **CSS** standard which describes some added in functionality, such as constants, expressions, and fallbacks for newer specifications. (Read more: http://sass-lang.com/)

jQuery: jQuery is a powerful DOM manipulation framework that allows developers to quickly work with the DOM to achieve powerful and interactive websites. (Read more: http://jquery.com/)

jQuery Mobile: jQuery Mobile is a mobile version of **jQuery** that allows the rapid development of mobile **jQuery** based websites. (Read more: http://jquerymobile.com/)

Bower: A client sided package manager for libraries and tools. It used throughout the application to manage dependencies and ensure all developers are up to date. (Read more: http://bower.io/)

npm: The **Node Package Manager** (**npm**) is a tool for importing libraries and module into a **Node.js** project. It is similar to **Bowe**r but is mainly used to manage the back-end of the software, rather than the client side. (Read more: https://www.npmjs.org/)

V8: V8 is Google's implementation of the **ECMAScript JavaScript** specification. It is a fast, optimized version of the specification designed with performance in mind. (Read more: https://developers.google.com/v8/)

Node.js: Node.js is a sever side implementation of **JavaScript** using the **V8 JavaScript** engine to interpret. (Read more: http://nodejs.org/)

MongoDB: MongoDB is a document-oriented database that allows storage of data via blobs and "documents", unlike traditional relational database management systems. It is a popular choice among many **Node.js** developers. (Read more: http://www.mongodb.org/)

nodemon: A build monitoring tool for ***Node.js*** that assists in rapid builds. (Read more: http://nodemon.io/)

Gulp: Gulp is a streaming build tool used to simplify the process of developing **JavaScript** applications. Similar to *make* in **C** and other build tools, *Gulp* can manage compiling, minifying, packaging, compressing images, building native applications with ***Cordova*** and more. (Read more: http://gulpjs.com/)

REST API: A REST API is a specific **API** that provides **CRUD** operations at **HTTP** endpoints.

CRUD: A simple acronym for "Create, Read, Update, Delete" which helps describe the entity manipulations of a typical business application.

Sails.js, Express: A structured web application server provided that makes creating **REST APIs** a snap. (Read more: http://sailsjs.org

11 System Overview

"Help Me! Laurier" is an Android application that runs on the newest bleeding edge technology to make use of the latest and greatest software available, to make the best applications possible. To do this, powerful frameworks like **AngularJS** and **Ionic Framework** are used on the client side. Powered by powerful **HTML5**, they allow the rapid development and flexibility required to build our application. For brains on the server side, we have chosen things like the upcoming **V8** powered, **Node.js** and **npm.** With these technologies, we aim to utilize all the tools available to the full potential. Below, we describe the rationale, other choices considered and how we intend to use most of the major technologies.

11.1 Client Overview

On the client side, choosing **AngularJS**, **Ionic Framework** and **Cordova** enables us to work with some of the best technology from the best companies. Google develops, and therefore has stake, in **AngularJS** which our team has experience with. Thus, for the construction of this specific product we will make use of **Angular**. **Angular** provides data binding, routing, view management, resource management, controller management, and scaffolding to help in the developing process. It makes the most sense for our specific application.

Ionic Framework is another layer on top of **Angular**, specifically targeted towards making great looking, easy to develop mobile applications built with **Angular**. As our team also has experience with this and **Ionic Framework** is built top down to be integrated 100% with **Angular**, it was our logical choice. The set of controls and power provided to map UI to a data contract model in a few lines of code made it ideal for creating an application fast.

Cordova, **Bower**, and **Gulp** are used as part of the build process to ensure a smooth development process. **Gulp** is one of the few **JavaScript** build tools available, with the alternative being **Grunt** or **Yeoman**. **Ionic** uses **Gulp**, so it was the most logical choice. **Bower** is the only reasonable option for client-side dependency management, so it is also a clear choice. **Cordova** is described in more detail below.

Other options have been considered before settling. However, all of them have pitfalls that were answered by **Ionic** and **Angular**. These are listed below with a brief description of why they were not used:

- **Pure JavaScript**
  - Development with no frameworks is cumbersome and eats up time for no reason. In the software industry today, when writing simple **CRUD** applications like ours – it is only logical to leverage all existing support ecosystems.
- **jQuery Mobile**
  - **jQuery Mobile** is great and provides an excellent set of controls and intuitive building blocks to great powerful applications. However, **jQuery** encourages mixing in view and model code with the selector pattern it has imposed for many years. **Ionic** has all the great things about **jQuery Mobile** – packaged as an **Angular** friendly wrapper.
- **Bootstrap**
  - This is a great **CSS** library we considered but it simply has no bindings available for quick prototyping and our team has little experience with it. The learning curve would exceed the scope of the project.
- **Sencha Touch**
  - **Sencha** is incredibly popular, similarly to **Cordova**. All team members are familiar with **Cordova**, however. This makes **Cordova** the better choice as it meets all the requirements for this project.
- **Kendo UI**
  - **Kendo** has restrictive licensing which would make it difficult to open-source our project, a potential fate for this project in the future. In order to future proof ourselves; this option had to be ruled out.
- **Grunt**
  - An excellent package manager that was considered – but **Ionic** integrates perfectly with **Gulp**, so it makes sense to use what is compatible with our toolset rather than work against it.

Other technologies may have also been considered but were decided to not be of high enough relevance to be list above.

11.2 Server Overview

When picking technology for our server stack, we decided there are a few main requirements:

- Every developer had to have knowledge in the language it was going to be written in
- It had to have the ability to provide a **REST API** easily
- It had to run on **UNIX** systems (this ruled out **C#** / **.NET**)
- Having a low entry barrier for new developers was a must

With these requirements in mind, an analysis was performed and the **JavaScript** powered **Node.js** was selected for a variety of reasons. First and foremost, it uses **JavaScript**. As our client end is written in **JavaScript**, every developer will need to know it. This will make every developer able to work on both code bases with some degree of skill. **Node.js** runs on almost every platform and has frameworks like **Express** to provide **REST APIs**. Best of all, **Node.js** is a rising popular platform: support is everywhere and is only growing.

For storage, we had a few options. **MongoDB** is by far the most popular choice when developing with **Node.js** which is ultimately why it was chosen for this project. The document oriented hierarchy allows us to store massive amounts of question and answer information without a traditional relational database. However, these were considered as well. **MySQL** and traditional databases are unfortunately second class citizens in the context of the **Node.js** community. For this reason, they were quickly discarded.

Express and **Sails.js** are being used to give a structured feeling throughout the entire application lifecycle and prevents writing a lot of boilerplate code. They are industry hardened and have been used by big name companies like Facebook. The alternative was writing all this boilerplate from scratch in **JavaScript**, which is a major problem as it violates one of the major rationale we used for selecting libraries previously.

Node.js and the powerful **npm** were selected, but we also considered a few other possibilities based on their merits. Ultimately, we considered but rejected:

- **C# & ASP.NET**
    - **C#** is a powerful language that produces a great **REST API** with little effort, great **ORMs** (such as **Entity Framework**) and a wide tool service. Unfortunately, they are not multi-platform and our team is not well versed with it.
- **Ruby on Rails**
    - Nobody on the team has any experience with **Ruby**, even though the technology looked great and similar products such as **Disqus** has been developed under it.
- **Java / Tomcat**
    - This is usually reserved for enterprise companies and has a lot of boilerplate required to get started. The barrier of entry is too high here to consider using it for our current development team and expected team.
- **MySQL**
    - For reasons outlined above in detail, **MySQL** did not fit well within the paradigm we were aiming to use. For this reason, it was culled immediately without much second thought.
- **Sphere.**
    - This is an enterprise level framework that was considered but quickly discarded.

With everything we selected, we made sure it added significant value to our project. Each framework, tool, and library has been selected with extreme care as the architecture will be designed around them. Care will be taken to abstract the details around them, however.

11.3 Conventions and Readability

This section will talk briefly about conventions and guidelines that will be used throughout the project. Since readability is such a major concern when writing complex software, it is worth outline the guidelines required to keep a consistent and uniform code base. This section will outline briefly some documentation and style guidelines that every developer reading this document should be familiar with.

It is placed here as every developer must have read and understood this document before construction can begin – thus, to keep developers familiar with the choices here we overview it. This information is also placed here as it has some bearing on the design and how future revisions will be completed. Please keep these guidelines in mind when editing this document.

11.3.1 Documentation: JSDoc

The standard documentation standard used throughout the project will be **JSDoc** which allows a clean standard of **JavaScript** guidelines to enable everyone to be on the same page. At the very minimum, functions and methods should have a clear description and the return values of each function needs to be commented.

At the very minimum, developers should expect to make use of:

- `@param`
- `@class`
- `@returns`
- `@description`

These are the bare minimum tags that are required to give a good function or class definition, so good documentation can be provided. For example:

```
/*
  * Given the name of a person and the tab, loads and changes to the page required.
  * @param {string} person: The name of the person we will be switching to
  * @param {string} tab:    The tab of this persons profile we will be swapping to
*/
```

By using method headers like the one above, we can maintain a clear and uniform codebase that everyone can understand accordingly. Documenting return values and their type are of extreme value and importance as **JavaScript** is not a concrete typed language.

11.3.2 Conventions

Generally, these conventions have been borrowed from Douglas Crockford, from his book *JavaScript: The Good Parts*[1] but they are briefly summarized here so all developers are up to speed.

- camelCase is the choice of naming for both variables and functions
- Line length should be below 80 characters
- Indentation should be limited to 4 spaces
- **JavaScript** will be stored within **JavaScript** files and not embedded within **HTML** pages, as our application is a single page application
- All variables should be declared before usage
- *eval* is forbidden throughout the application

For things that require privacy, the convention with a prefixed _ will be used. This includes both the client and the server sided component. The exception is a closure, in which case privacy is already guaranteed within the scope. In this case, there is no need to use a convention privacy modifier.

In under no circumstances, shall one invoke a private function from outside the intended scope. For example, if you see the following code:

```
someObject._doStuff()
```

Then, the code is incorrect according to our style guide and should be fixed.

All **AngularJS** controllers must be contained within modules to avoid pollution. More on this in the next section…

[1] http://javascript.crockford.com/code.html

11.3.3 AngularJS and Ionic Framework Guidelines

AngularJS and **Ionic Framework** are both flexible and allow for extreme flexibility in both cases. Generally, here are some guidelines:

Do…

- Separate all sections of the markup into separate template files and link them together
- Create directives for repeating chunks of code
- Make use of **AngularJS** modules to ensure code is kept modularized
- Create a controller per view, and a model per view
- Use the **Ionic Framework** router for all movement within the application.
- Create services to abstract away network calls from the controller modules

Don't…

- Make extensive use of **jQuery** to manipulate the **DOM** in ways that **AngularJS** may not be aware of
- Don't manipulate the **DOM** directly
- Do not manually direct around the application
- Couple services to controllers

AngularJS controllers should be grouped in a way that is logical and easy to find. *Refer to Developer Construction Documentation for more details on exact specifics.*

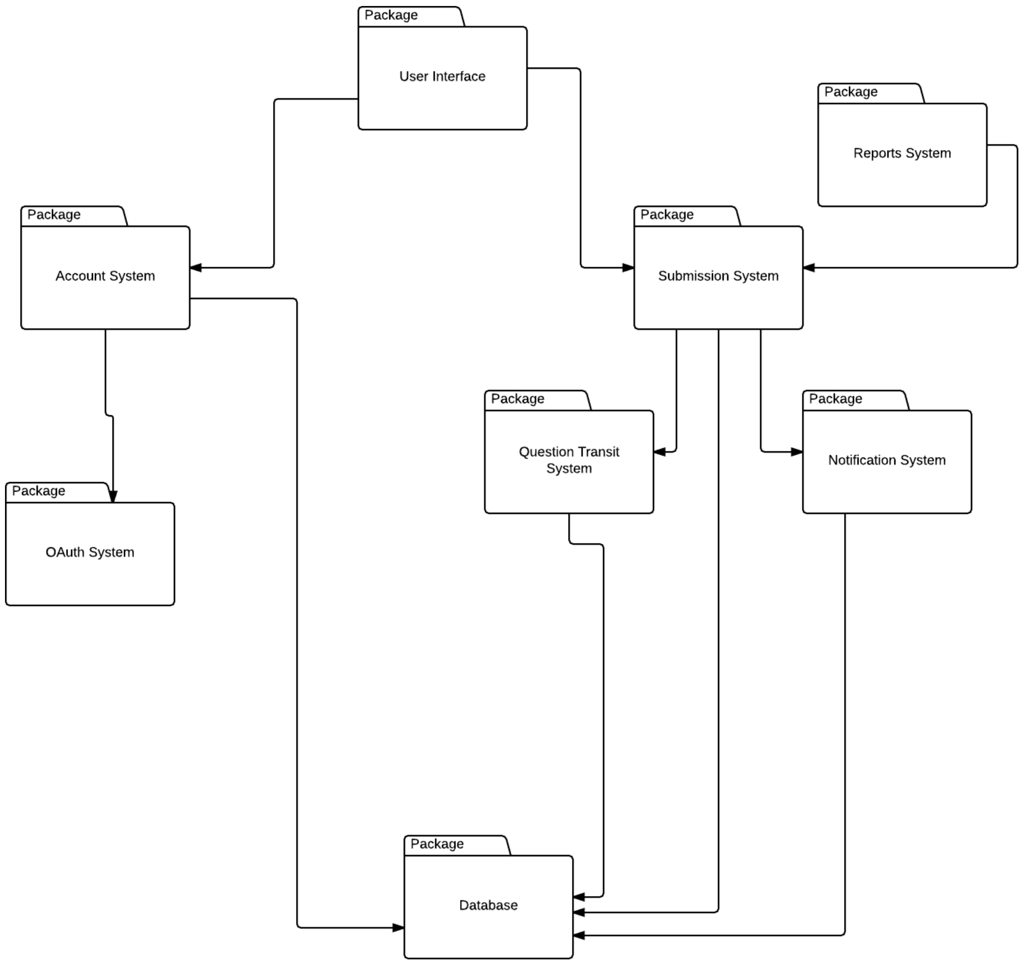
12 System Architecture

12.1 High Level Overview

Below, we will describe the system architecture by listing the relationships between the major subsystems involved in the application. We decompose them into a few major systems:

1. **User Interface**
2. **Account System**
3. **OAuth System**
4. **Submission System**
5. **Question Transit System**
6. **Notification System**
7. **Reports System**
8. **Database System (I/O)**

These major subsystems make up the bulk of the work for the application. To get a better look at how they interact, please refer to the below system architecture diagram that illustrates dependencies:



The diagram should stand by itself, but a brief description is given below for completeness sake.

The eight major subsystems in use are represented by the different packages above. At the root of everything is the **User Interface.** This interacts with all the subsystems indirectly as the user will need to interact before any of the other systems can react. Most of the systems that do actual work will have a relationship with the database. The **Submission System** will contain core services regarding questions and answers, while the **Account System** will describe authentication related tasks. Each subsystem below is separated out services that are lent to these two major systems. Their details are important but will be described in much more detail in the next few sections.

12.2 Layout Decision Rationale

Our system is mostly layered in that most of the subsystems try to interact with ones that that are directly above them and between each other and not try to skip layers. There is an exception to this rule and it's the data access layer (**Database System)** which generally obeys this rule but has a minor exception or two. This approach was taken as it found to be a more plausible approach than portioning the system in pieces into various partitions. Here's the reasoning:

AngularJS works with services and controllers that work together in a way similar that an operating system does. That is, services provide services to other portions via dependency injection to pass around various utilities. By nature, this design usually leads itself into the **layering** approach, rather than the partition approach. In fact, *Google* recolonized that when developing the framework and has included namespace services for layering and module organization. In fact, trying to partition the application in isolation would be working against the spirit of our technology stack.

When working with a framework, one must learn to think in terms of it rather than against it. For example, one can manipulate the DOM directly in **AngularJS** similar to **jQuery** to directly do things that were not immediately intended. However, by working against the system you are losing out on a lot of functionality and potentially opening one's self open to a lot of hazards. **AngularJS** discourages behavior that would operate like this as it prefers you to think within its mindset.

Layering also has the advantage of creating perfect isolation between most subsystems which allows one to carefully consider what things are attached to. Keeping isolation like this encourages loose coupling and a strong sense of cohesion. By doing this, it encourages good programming practices and testability. Testability is a huge mantra that **AngularJS** tries to enforce.

12.3 Cohesion and Coupling Analysis

12.3.1 Cohesion

First, we will describe the cohesion as it is the easiest to describe. Cohesion is meant to keep the subsystems uniform and keep their tasks separated so they perform singular tasks. The layout above is believed to be a good compromise on keeping tasks separate and yet well put together. Ideally, every system would be completely decoupled and have perfect cohesion. However, the time constraints, budget constraints and technical constraints make this tough to achieve. So, where compromises have been made in design they have been outlined. Reasons will be provided for rationale for tradeoffs. An analysis on each sub system and the justification for why it is highly cohesive is found below.

User Interface. The user interface subsystem is responsible for all UI aspects that can occur throughout the application. Since it only handles UI rendering and responsive hooks, this subsystem is completely cohesive in that it's separated from all logic. There is some minor cohesive violates at this subsystem will also handle some degree of user input and rendering, but this is generally an accepted type of cohesion mixing that is acceptable when it comes to *User Interface* design.

Account System. This system is responsible for authentication generally and nothing else. It communicates with the primary user database and allows retrieval of these details and verification of their credentials. It also handles the validation of authentication tokens and their expiry. However, it has no knowledge of specific systems and instead delegates those to the specific subsystems that have specialized knowledge in these topics. The **Account System** is quite cohesive as it specifically deals with one thing – authentication. The communication with the user database is not done through the actual account system, but delegated to the database subsystem to further reduce the amount of responsibility for a given subsystem.

OAuth System. This system is responsible for authentication **Google Authentication**. In order to keep the **Account System** free of any specific implementation details, this subsystem is included as a dependency to communicate at a lower level while allowing the account system to deal with higher level details. It is cohesive as it does a single job: it communicates with **OAuth** providers supported, in this context, Google and **MyLaurier**, and provided a service to it. It does nothing else like storing the info or even validating it.

Database System. This system is strictly used for storing information into a database system. It does not process any data or actually transform any of it from an application level, it merely acts as a bus between different systems to read and write data. Due to this, it has a clearly defined purpose which can be said to be cohesive. It has no reliance on specific data types either thanks to the **MongoDB backing,** which will be described later on in the coupling analysis. Since it has just one specific job, it can be said to be very cohesive.

Submission System. This system has a couple jobs – which makes it a bit less cohesive but the drawbacks were thought through and have been declared as being worth it. The submission system is responsible for dealing with submissions – that is retrieving them, saving them, modifying them, and synchronization with the server system. Since the subsystem does some synchronization, it is not completely cohesive as the purpose is slightly blurred. In order to alleviate this, we have added another service called the *Synchronization Service* that will be implemented as an **AngularJS** service allowing us to delegate most of that work away. With this, we will still have some vague stubs and tramp data being passed around, but the intent will be clearer and the level of abstraction more sufficiently divided as there will be only a single connection. There's more on this in the coupling analysis.

Reports System. The report system is responsible for managing reports – checking them, queueing them, setting their status, and the like. The job is very clearly and it collaborates together with the **Submission System** in order to find reports attached to certain questions and answer submissions that might require moderation. By doing this, it can clearly divide out the work it is required to do. It delegates the task of finding a **`Submission`** out and focuses on simply interacting with a working set of data. It has a level of abstraction set – it works with working sets of data without questioning where they came from or how they were delivered.

Question Transit System. This is a special kind of system that is responsible for managing outgoing traffic regarding **`Submissions`**. This is primarily used by the **Submission System** to buffer out messages that might need to be transmitted and delivered. It does processing and works out where and how things need to be delivered. Note: It is only doing computations but actually has no idea HOW these will be sent. Due to this, the job is very well defined and it has a single task. It can be said to be cohesive. This leads to…

Notification System. This is a system with the sole task of taking in data from the **Question Transit System** and ensuring it reaches the places that the packages of data have been labelled to be sent. A good analogy is your post office: Someone prepares a shipping label, packs all the goods into a box – but the person who actually delivers it is UPS. **Question Transit** would be your supplier and the **Notification System** would be more like *UPS* or *Canada Post.*

12.3.2 Coupling

Below, we describe the tradeoffs that were encountered in coupling some systems together and why it was done. Generally, the goal is to reduce the amount of connections between subsystems and ensure they can be disconnected with fair ease to ensure a decoupled architecture. In a perfect world, everything would be perfectly decoupled and have little connections. However, in reality this is an incredibly difficult feat to achieve and in some cases downright impossible. Rationale for all choices will be provided in the following points.

User Interface. A completely recoupled user interface is tough to create – a UI that cannot broadcast events for logic to take care of would be nearly useless. Due to this, the **User Interface** has a small degree of coupling on the **Authentication System** and **Submission System.** Mainly, it uses services from both to populate the interface with some data. In the case of the **Authentication System,** it is authorization rules such as: are you logged in, are you allowed to use the application, and are you suspended? In the case of the **Submission System,** it is something more like: I need these questions to display, can you increase the score of this **`Submission`?** These minor types of coupling are tough problems to occur and given budget constraints, it has been decided that this minor coupling is an acceptable way to complete things while still maintaining a fair amount of abstraction.

Account System. Aside from the documented coupling on the **User Interface,** there are no other strict dependencies on the **Account System** that causes major coupling. There is an inherent dependency on the **Database System** which is shared on many different systems. This is also another type of dependency that is difficult to hide – so we simply try to hide it as far down the chain as possible to reduce the amount of systems that can interface with it at once. More on this during the analysis of the actual **Database System.**

OAuth System. The **OAuth System** has a hard dependency on the **Account System** which makes logical sense. The only task it has is to serve it – so the coupling here makes sense. Even if it were disconnected somehow from the **Account System,** it would need to transfer information eventually back to it and no other subsystems requires a similar bus, so the tradeoff has been made to simply make them dependent.

Database System. Before moving onto other systems, it is worth describing the **Database System.** It is used and depended on by many different systems, such as the **Account System, Submission System, Notification System,** and **Question Transit System.** All of these systems require the storing of some sort of information that will be persisted through the application, so they require a hard dependency on the system. At first, some discussion was held that wondered about separating the database access further, but it was decided that access through a simple layer was all that was needed at this level of design. The tradeoff was a conscious one and all team members are aware of it.

Submission System. The **Submission System** relies on the other two systems directly below it in the diagram outlined above. This decision was made as the systems below it cannot operate without them, so it is simplest to simply pass the data through a single weakly linked pipe. Their coupling will be low and can be decreased with dependency injection provided by **AngularJS,** so anything with a similar interface (one or two transfer methods) will be able to substitute with minimal effort. This makes the coupling a more trivial problem that if there were many connections. By reducing the connection count, we create what is known as a **weak coupling.**

Reports System. The coupling is quite small on this system, it has a small amount of coupling on the report data types but other than that is fairly loose and free. Most of the retrieval data is already handled in the **Submission System** so the **Reports System** has minimal dependencies. In fact, the only entity that this system should really have to deal with is that of **`AbuseReport.`**

Transit System. The transit system has a hard dependency on the **Notification System** as per the reasoning of the analogy given in the *Cohesion Overview.* The simplest and cleanest way without combining the systems involves a variant of dependency injection to separate the sending mechanism from the bundling and processing, which is what has been done.

Notification System. The Notification System has no major coupling dependencies, similar to the **Reports System.** Instead, it is primarily focused on delivering packages of **`Notification`**`s`. This reduces the amount of entity information that it needs to be aware of, thus reducing total coupling.

12.4 Detailed Subsystem Analysis

12.4.1 User Interface System

12.4.1.1 Responsibilities

The **User Interface** subsystem contains responsibilities for responding to user input. This subsystem will be composed mostly of the **AngularJS** services which work together to interface with the other subsystems to manipulate and update data throughout the application. If the user can physically see it on their phone screen, this subsystem will take care of it.

12.4.1.2 Class Breakdown

Boundary	Entity	Control
`LoginPage`		`InterfaceControl`
`WelcomePage`		
`ParentFrame`		
`HomePage`		
`NotificationPage`		
`SubmissionPage`		
`ViewTopicsPage`		
`EulaPage`		
`ReportQuestionPage`		
`ProfilePage`		
`ViewQuestionPage`		
`SubmissionCard`		
`FooterPage`		
`HeaderPage`		
`HelpPage`		
`LandingPage`		
`Page`		
`FormattingWidget`		
`Listbox`		
`Textbox`		
`Button`		

12.4.2 Account System

12.4.2.1 Responsibilities

The **Account System** subsystem contains responsibilities for managing authentication and keeping users signed into the service. This subsystem is mainly used at the beginning of the application lifecycle to verify identity but is also used for services like verifying identity when making profile changes on the **Profile Page** and other points in the application where identity operations are required. The **Account System** does not do **OAuth** authorization directly by itself – it relies on the **OAuth System** for this so it can effectively abstract away the service provider aspect and focus purely on the identity verification at the application level.

12.4.2.2 Class Breakdown

Boundary	Entity	Control
`LoginPage`	`AuthenticationToken`	`ServerControl`
`WelcomePage`	`User`	`AuthenticationControl`
`LoginForm`		

12.4.3 OAuth System

12.4.3.1 Responsibilities

The **OAuth System** subsystem contains responsibilities for managing authentication and keeping users signed into the **OAuth** providers. At the time of implementation, the only actual provider that this subsystem needs to provide is **Google Authentcaton** as all Laurier students are required to have a school e-mail – powered by **Google Apps**.

12.4.3.2 Class Breakdown

Boundary	Entity	Control
	`AuthenticationToken`	`ServerControl`
	`User`	`AuthenticationControl`

12.4.4 Submission System

12.4.4.1 Responsibilities

The **Submission System** subsystem contains responsibilities for **`Submissions`** throughout the application. It controls most of the data flow regarding **`Submissions`,** specifically **`Question`** and **`Answer`** per the request of **`User`.**

12.4.4.2 Class Breakdown

Boundary	Entity	Control
`QuestionPage`	`AuthenticationToken`	`AnswerControl`
`SubmissionPage`	`User`	`QuestionControl`
`ViewTopicsPage`	`Submission`	
	`Question`	
	`Answer`	

12.4.5 Question Transit System

12.4.5.1 Responsibilities

The **Question Transit System** subsystem contains responsibilities for questions to be computed and directed to the right users to answer. This subsystem will contain the bulk of logic for analyzing users and question content to find the perfect match.

12.4.5.2 Class Breakdown

Boundary	Entity	Control
	`User`	`ServerControl`
	`Submission`	`QuestionControl`
	`Question`	
	`Answer`	

12.4.6 Notification System

12.4.6.1 Responsibilities

The **Notification System** subsystem contains responsibilities for **`Notification`** delivery to the various users throughout the application. If a system needs to push out notifications to others, it should contact this system.

12.4.6.2 Class Breakdown

Boundary	Entity	Control
	`User`	**`ServerControl`**
	`Submission`	**`QuestionControl`**
	`Question`	**`NotificationControl`**
	`Answer`	
	`Notification`	

12.4.7 Report System

12.4.7.1 Responsibilities

The **Report System** subsystem contains responsibilities for managing abuse reports submitted by users through the submission system. This system will be responsible for marking, queuing, and performing operations on the various **`AbuseReport`** objects**.**

12.4.7.2 Class Breakdown

Boundary	Entity	Control
	`User`	**`ServerControl`**
	`Submission`	**`QuestionControl`**
	`Question`	**`AnswerControl`**
	`Answer`	
	`AbuseReport`	

12.4.8 Database System

12.4.8.1 Responsibilities

The **Database System** subsystem contains responsibilities interacting with the storage backing of the entire application. When a service or system wants to store information, it will need to contact this system to do so.

12.4.8.2 Class Breakdown

Boundary	Entity	Control
	`User`	`ServerControl`
	`Submission`	`QuestionControl`
	`Question`	`NotificationControl`
	`Answer`	
	`AbuseReport`	
	`Notification`	
	`ProfileQuestionEntry`	
	`AuthenticationToken`	

13 Data Design

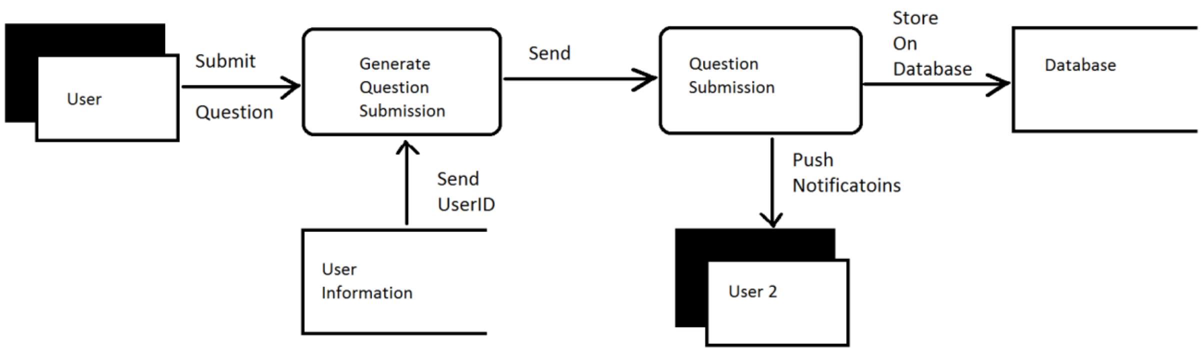
In the next few sections, different objects and how their data can be interacted with and manipulated are discussed. For each object, a brief description of the process and a diagram is provided. At the end of this section, a dictionary is provided with all elements and various attributes that describe everything in one easy to use index.

13.1 Data Flow with Objects

13.1.1 QuestionSubmission

A question takes text and categories from the user and creates a **`QuestionSubmission`** object. The user enters the title for their and their questions body then presses submit. The system requests categories for the question such as course information through an alert. After the user confirms there categories a variety of information is passed to the backend of the system.

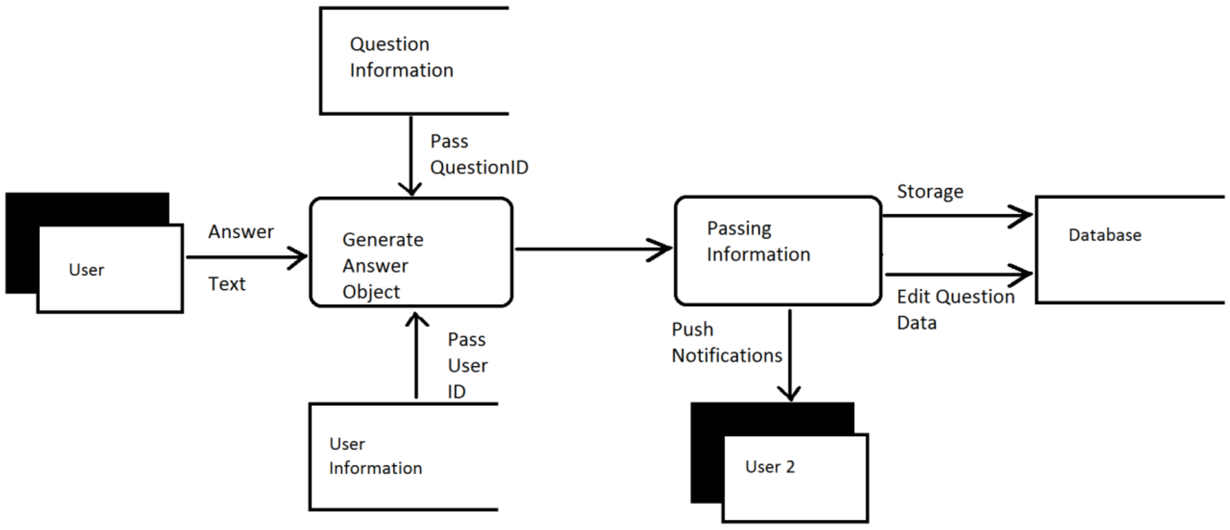
The system creates a **`QuestionSubmission`** object by setting **`Title,`** a string equal to the title textbox contents, setting **`CategoryID,`** an integer, to the selected category, initializing **QuestionAnswer**s, an array, for **`AnswerID`**s as an empty array, and generating the remainder of a **`Submission`** object which it inherits from. This involves setting a **Body**, a string, equal to the question textbox data, setting an integer **`AuthorID`** equal to **`UserID`**, initializing **`Score`**, an integer, to zero and its **`Status`** to Active, setting **`Date`**, a **JavaScript** date object**,** equal to the current date and generating an **`ID`**, an integer, for itself. The information is then placed in the database and sent to other users based on the **`CategoryID`**. The following dataflow diagram illustrates this process:



13.1.2 AnswerSubmission

An answer takes text from the user and creates an **AnswerSubmission** object. The user enters their response to a given question and presses submit. The **ParentQuestionID**, **UserID** and answer text are then sent to the back end of the system. The backend takes this information and generates an **AnswerSubmission** object.

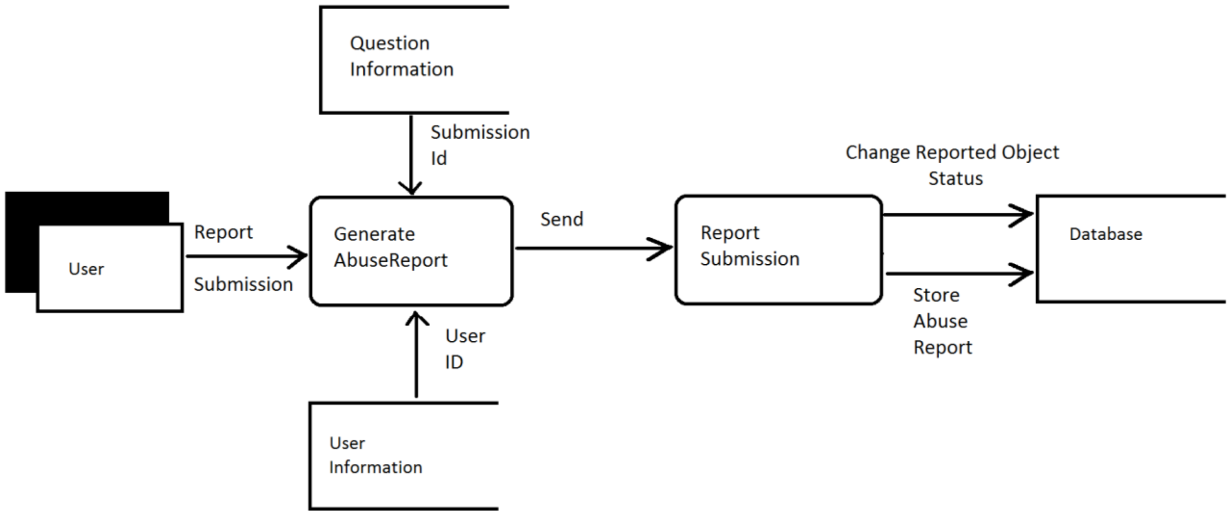
The **AnswerSubmission** object is created by setting **ParentQuestionID**, an integer, equal to the **AuthorID** of the question the user is answering and generating the remainder of a **Submission** object which **AnswerSubmission** objects inherit from. This involves setting **Body,** a string, equal to the answer text, setting **AuthorID**, an integer, equal to **UserID**, initializing **Score**, an integer, to zero and initializing **Status** to Active, setting **Date**, a **JavaScript** Date object, equal to the current date, and generating an **ID**, an integer, for itself. The backend then updates the **QuestionSubmission** object in the database by placing its own **ID** in the **QuestionAnswers** array found using the **ParentQuestionID**. Finally the backend pushes a notification to the user who asked the question by referencing the **AuthorId** on the **QuestionSubmission** object. The following dataflow diagram illustrates this process:



13.1.3 AbuseReport

A report takes a report type and a brief description of why they are reporting a **`Submission`** to make an **`AbuseReport`**. The user selects either spam or language for the reason of the report. The backend then creates an **`AbuseReport`** object.

The backend generates **`AbuseReportID`**, **a**n integer, for the object, setting **`Reason`**, a string, to either spam or language, setting **`Description`**, a string, to the description entered by the user, setting **`SubmissionID`,** an integer, to the reported objects submission id, setting **`Date`**, a **JavaScript** Date object, equal to the current date, setting its **`Status`** to active and setting **`ReporterID`**, an integer, to the reporters **`UserID`**. Once the report has been generated the **`Status`** of the flagged item is set to under review. The **`AbuseReport`** object is then place in storage to be reviewed by an admin later. The following dataflow diagram illustrates this process:



13.2 Data Dictionary

Below, all data elements and their respective members are listed exhaustively. This can be used as a quick reference of all data flowing throughout the application.

Name of Data Element	Description	Narrative
AbuseReport	Record comprising fields: **AbuseReportID** **Reason** **Description** **SubmissionID** **Status** **ReporterID** **Date**	The fields contain all details of a report
AbuseReportId	An integer **AbuseReport**	A unique number generated by **generate_reportID**
AnswerID	An integer **AnswerSubmission**	A unique number generated by **generate_submissionID**
AnswerSubmission	Record Comprising fields: **AuthorID** **Body** **Date** **ParentQuestionID** **ID** **Score** **Status**	The fields contain all details of an **AnswerSubmission**
AuthorID	An integer **Submission**	The **UserID** from the user who generates a submission
Body	A string **Submission**	contains the bulk of a **Submission** gathered from the description textbox in the UI when a user submits a **QuestionSubmission** or **AnswerSubmission**
CategoryID	An integer **QuestionSubmission**	The **CategoryID** generated when the user selects one of several pre-set categories
Date	A JS date Object **Submission** **AbuseReport**	The current date generated using built in **JavaScript** functions
Description	A string **AbuseReport**	contains a description of an **AbuseReport**
generate_reportID	Procedure Output parameter **ReportId**	Increment a 32 bit integer saved within the backend by one. Return the saved integer.

generate_submissionID	Procedure Output parameter **SubmissionId**	Increment a 32 bit integer saved within the backend by one. Return the saved integer.
ID	An integer **Submission**	An **ID** for a **Submission** generated by **generate_submissionID**
ParentQuestionID	An integer **AnswerSubmission**	The **ID** of a **QuestionSubmission** object used by an **AnswerSubmission** object to find the **QuestionSubmission** Object it relates to.
QuestionAnswers	An array **QuestionSubmission**	An array initialized to empty to contain the **ID**s of answers to the **QuestionSubmission**
QuestionSubmission	A field containing: **Category ID** **Body** **Title** **Date** **QuestionAnswers** **ID** **Status** **AuthorID** **Score**	The fields contain all details of a **QuestionSubmission**
ReporterId	An integer **AbuseReport**	The **UserID** of the user submitting a **AbuseReport** object
Score	An integer **Submission**	An integer attached to a **Submission**. The **Score** is altered by users rating questions and answers and is used to decide what is displayed to the user first
Status	A status enum **Submission** **AbuseReport**	A **Status** Enum with 3 state Active UnderReview Deleted
Submission	A field containing: **Body** **Date** **ID** **Status** **AuthorID** **Date** **Score**	The fields contain all details of a **Submission**
SubmissionID	An integer **AbuseReport**	The **ID** of a reported **Submission** gathered by a

		report
Title	A string **QuestionSubmission**	A brief description of a **QuestionSubmission** gathered from the title textbox in the User interface
UserID	An integer **User**	The **ID** of a user generated upon profile creation

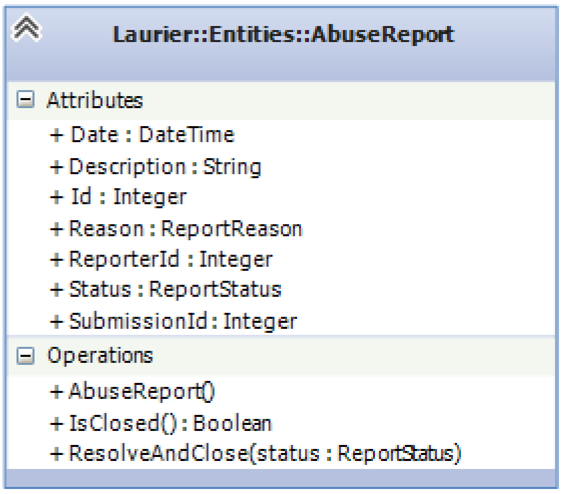
14 Object Breakdown

In the below sections, we will decompose and describe all the member functions and fields of the various objects used throughout the application. In all cases, a UML is provided to give a quick overview of attributes and operations.

In most cases, the UML diagrams show common data types in common with most languages since **JavaScript** has no typing system in place. This helps organize thoughts and maintain the uniformity of data typing, which is inevitable in **JavaScript** even without a formal typing system.

Boundary objects and smaller objects with no immediate purpose within the application has been excluded from this listing as they minor and too few to list. To list every boundary object would pollute the list and cause a lot of minorities to be listed. To reduce this, the major and medium usage classes have been defined and listed.

14.1 AbuseReport



14.1.1 Attributes

`Id`

An integer ID generated by **`generate_reportID`**

`Reason`

User input gathered directly from the User interface. When the submit button is pressed and an **`AbuseReport`** object is generated one of the parameters passed in from the UI is the selected reason.

`Description`

User input gathered directly from the user interface. When the submit button is pressed and an **`AbuseReport`** is generated one of the parameters passed in from the UI is the text from one of the selections. This is the Description of an **`AbuseReport`**.

`SubmissionId`

When the user selects the report button it is associated with a specific question. This integer is passed in from the front end however is unseen by the users and is used as a parameter to generate an **`AbuseReport`** object.

`Status`

When an **`AbuseReport`** is first initialized its **`Status`** is set to open. This does not require any parameters to create. When the report has been looked at this is changed to pending and when the review has been completed the **`Status`** is either set to accepted or denied.

`ReporterID`

This is the **`UserID`** of the user who is submitting the **`AbuseReport`** It is used to keep track of which user reported the **`Submission.`**

`Date`

This is a Date Object generated by the **JavaScript** date function. Its purpose is to keep track of when the **`AbuseReport`** is submitted which should help with giving faster feedback to the reports.

14.1.2 Methods

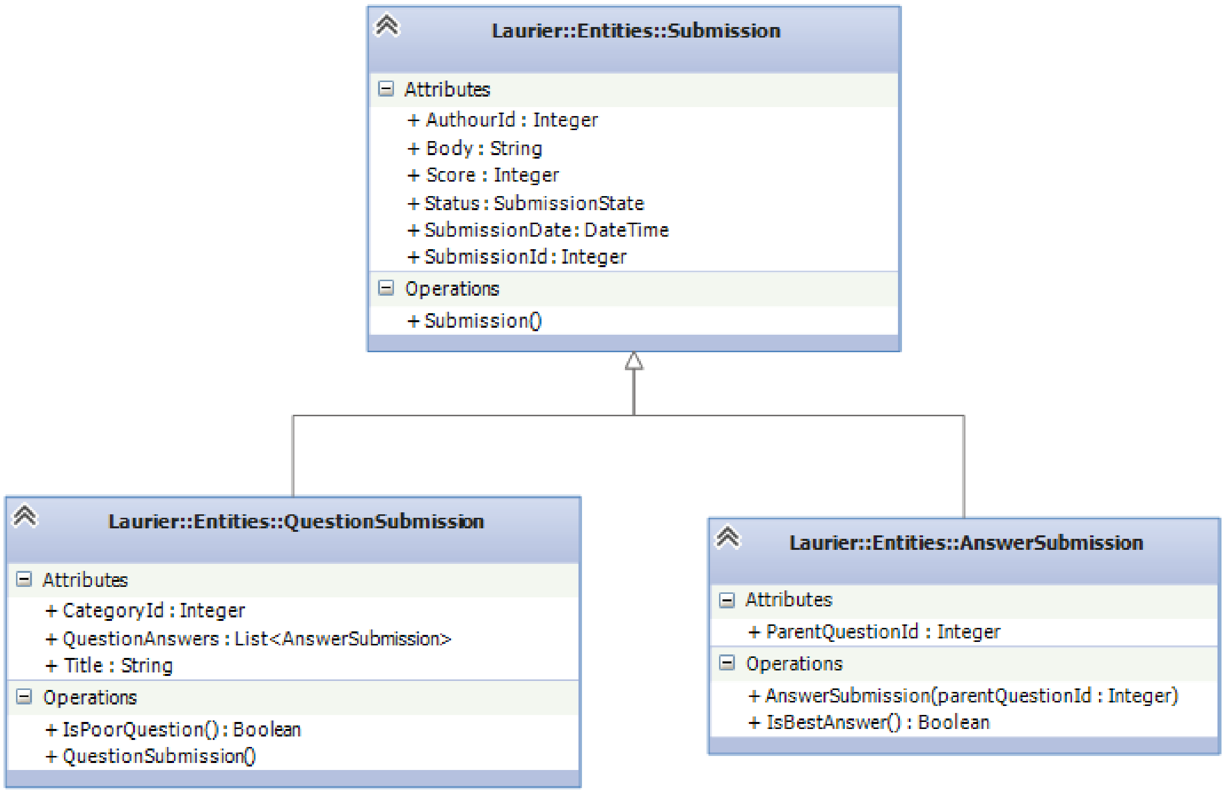
`IsClosed() : Boolean`

This determines whether or not this particular report has been closed yet (reviewed). When something Is closed, this will return **true.** Otherwise, this will return **false.**

`ResolveAndClose(status: ReportStatus)`

This allows the closing or setting of status for this particular report.

14.2 Submission


Laurier::Entities::Submission
Attributes
+ AuthourId : Integer
+ Body : String
+ Score : Integer
+ Status : SubmissionState
+ SubmissionDate : DateTime
+ SubmissionId : Integer
Operations
+ Submission()
Laurier::Entities::QuestionSubmission
Attributes
+ CategoryId : Integer
+ QuestionAnswers : List<AnswerSubmission>
+ Title : String
Operations
+ IsPoorQuestion() : Boolean
+ QuestionSubmission()
Laurier::Entities::AnswerSubmission
Attributes
+ ParentQuestionId : Integer
Operations
+ AnswerSubmission(parentQuestionId : Integer)
+ IsBestAnswer() : Boolean

14.2.1 Attributes

Id

All submissions have an integer ID generated when they are created in order to identify them. For a submission this is called its ID.

AuthorId

The AuthorID is the User ID associated with the User that submitted this Submission.

Body

All Submissions have a Body associated with them. For a question and answer, this will be the entirety of the text contents portion.

Status

This contains three states. Upon creation it is initialized to Active. *See State Diagrams for full transitions.*

Score

This is the score that is associated with the submission. It is an integer value and changed as the students go.

Date

This is a **JavaScript** date object and used to identify the time a submission was generated.

14.3 QuestionSubmission

14.3.1 Attributes

`Category ID`

This is the id based on the category selected in the user Interface. It is passed in as a parameter and used to allow users to view more applicable questions

`Title`

This is a brief description of the question. It is passed in by the title textbox in the user and used as a parameter in the creation of a **`QuestionSubmission`** Object.

`QuestionAnswers`

This is an array of **`AnswerSubmission`** object **`ID`**s its job is to keep track of the answers to a question so that they can be recovered when a user wants to view the question.

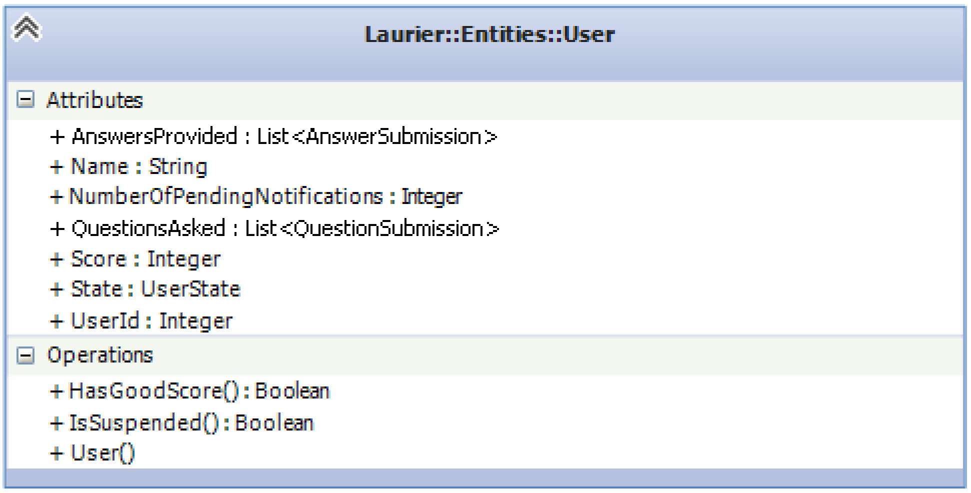
14.4 AnswerSubmission

14.4.1 Attributes

`ParentQuestionID`

This is an integer passed in upon creation of a question Submission Object. It is the ID of the question that the user has selected when they choose to answer a question.

14.5 User



+ AnswersProvided : List<AnswerSubmission>
+ Name : String
+ NumberOfPendingNotifications : Integer
+ QuestionsAsked : List<QuestionSubmission>
+ Score : Integer
+ State : UserState
+ UserId : Integer

+ HasGoodScore() : Boolean
+ IsSuspended() : Boolean
+ User()

14.5.1 Attributes

Name

This is a unique integer value assigned when a user first signs up for the application using a myLaurier email address and is used to identify specific

NumberOfPendingNotifications

This is a unique integer value assigned when a user first signs up for the application using a myLaurier email address and is used to identify specific

UserId

This is a unique integer value assigned when a user first signs up for the application using a myLaurier email address and is used to identify specific

QuestionsAsked

This is a list of submission ids associated with questions asked by a given User account

AnswersProvided

This is a list of submission ids associated with answers given through a User account

`State`

This is the state associated with a given user account and consists of the default of active, suspended which occurs as a discipline for an infraction, and deleted which is used when a user must be removed from the system.

`Score`

This is an integer associated with a user account that keeps track of the total score of their questions and answers.

14.5.2 Methods

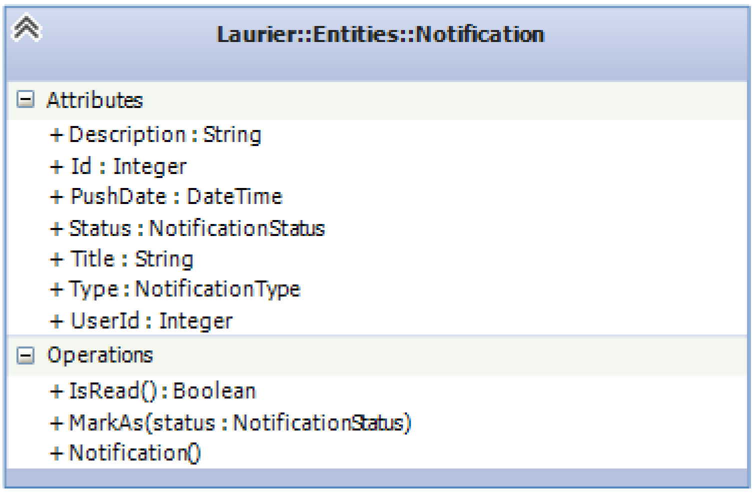
`HasGoodScore() : Boolean`

This function returns **true,** if a User is in good standing score wise. Otherwise, it returns **false.**

`IsSuspended() : Boolean`

This function returns **true** if a User is suspended, **false** otherwise.

14.6 Notification



14.6.1 Attributes

Id

This is a unique integer value created to unique identify each notification.

Type

This specifies what kind of notification it is. This will specify if the notification is an answer or a question, so the user interface system can determine how to render the view accordingly.

Description

This is a portion of the text within a submission object that relates to this notification.

Title

If the notification is a question, this contains the title of that question. If it is an answer, it contains the title of the question that the answer is a child of.

PushDate

A **JavaScript** date object used to show when a notification was generated.

`UserId`

A user ID associated with the User who generated the submission that the question goes with.

`Status`

This is the state associated with a notification object and consists of the default of unread and after a user check there notification it is changed to read.

14.6.2 Methods

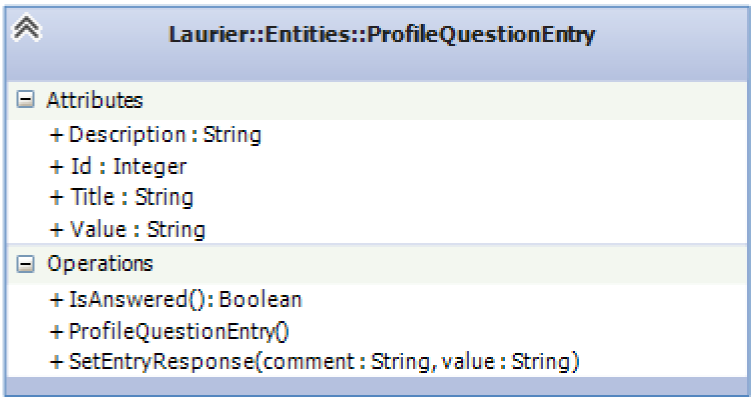
`IsRead() : Boolean`

Returns whether or not the notification has been read or not.

`MarkAs(status : NotificationStatus)`

Sets the notification status to the parameter *status* passed in.

14.7 ProfileQuestionEntry


Laurier::Entities::ProfileQuestionEntry

Attributes
+ Description : String
+ Id : Integer
+ Title : String
+ Value : String

Operations
+ IsAnswered() : Boolean
+ ProfileQuestionEntry()
+ SetEntryResponse(comment : String, value : String)

14.7.1 Attributes

`Id`

This is an ID used by a given question

`Title`

This is the question that the **`ProfileQuestionEntry`** asks and is stored as a string.

`Description`

This is the possible ways to answer the question in the **`ProfileQuestionEntry`**

`Value`

This is a string containing the answer they chose from the description.

14.7.2 Methods

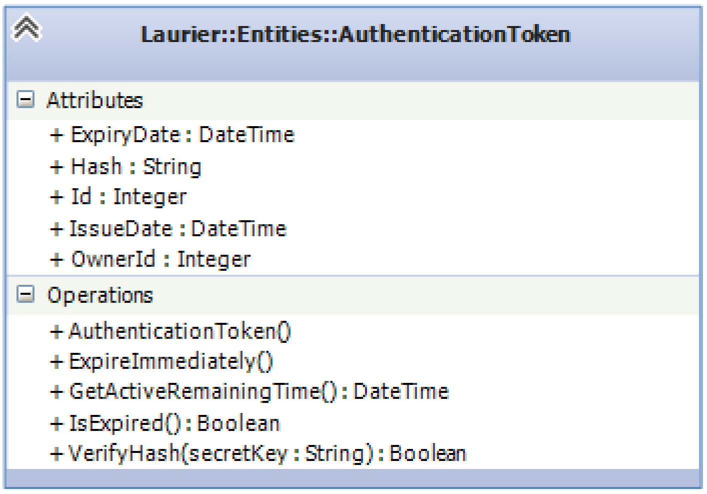
`IsAnswered() : Boolean`

Returns whether or not the profile entry question has been answered or not as of yet.

`SetEntryResponse(comment : String, value : String)`

Sets the value of the entry and optionally records a response.

14.8 AuthenticationToken


Laurier::Entities::AuthenticationToken

Attributes

```
+ ExpiryDate : DateTime
+ Hash : String
+ Id : Integer
+ IssueDate : DateTime
+ OwnerId : Integer
```

Operations

```
+ AuthenticationToken()
+ ExpireImmediately()
+ GetActiveRemainingTime() : DateTime
+ IsExpired() : Boolean
+ VerifyHash(secretKey : String) : Boolean
```

14.8.1 Attributes

`Id`

The unique ID associated with an **`AuthenticationToken`**

`OwnerId`

This is the ID of the user who caused the **`AuthenticationToken`** to be spawned.

`Hash`

This is a crypto-secure hashed string of a secret key provided by an **OAuth** provider

`ExpiryDate`

A **JavaScript** date object stating when the token will expire this is a fixed time after the **`IssueDate`**

`IssueDate`

A **JavaScript** date object stating when the token was created.

14.8.2 Methods

`ExpireImmediately()`

Given the current token, causes expiration of the token immediately.

`GetActiveRemainingTime() : DateTime`

Using the stored times, computes the period left in which the token will remain valid.

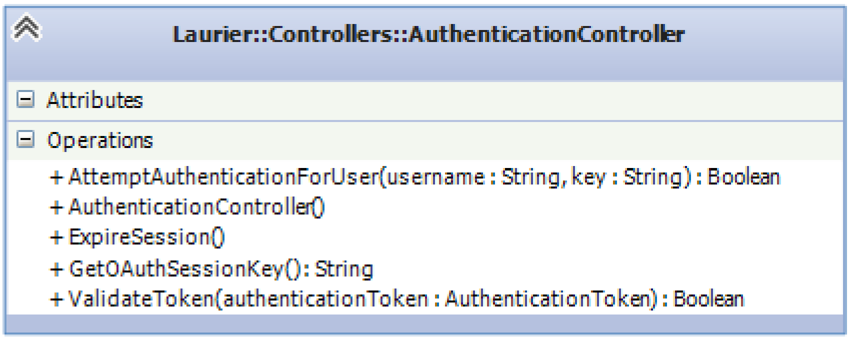
`IsExpired() : Boolean`

Using the stored times, computes whether the token is still valid or not.

`VerifyHash(secretKey : string) : Boolean`

Using the secret key, determines whether the hash is valid.

14.9 AUTHENTICATIONCONTROLLER



Laurier::Controllers::AuthenticationController

Attributes

Operations

+AttemptAuthenticationForUser(username : String, key : String) : Boolean
+AuthenticationController()
+ExpireSession()
+GetOAuthSessionKey() : String
+ValidateToken(authenticationToken : AuthenticationToken) : Boolean

14.9.1 Methods

`AttemptAuthenticationForUser(username :string, key :String): Boolean`

Takes in a Username and a string and returns whether or not the username has a valid hash key associated with it and is used to log in.

`AuthenticationController()`

This is the constructor for the authentication controller

`ExpireSession()`

This expires an authentication key that is currently stored.

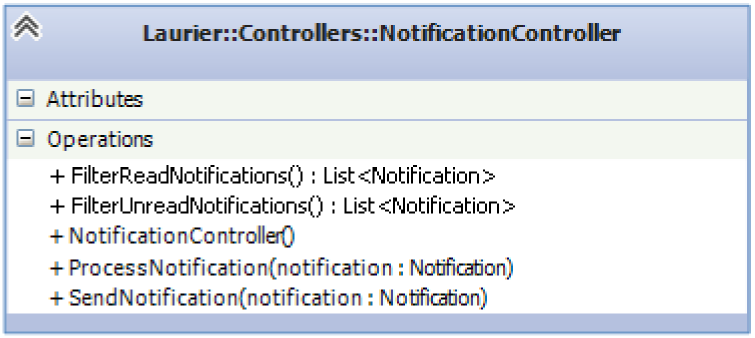
`GetOAuthSessionKey(): String`

This gets a session key from the **OAuth** provider and returns it as a string.

`ValidateToken(authenticationToken: AuthenticationToken): Boolean`

This tells you whether a token is valid.

14.10 NotificationController


Laurier::Controllers::NotificationController

Attributes

Operations

+ FilterReadNotifications() : List<Notification>
+ FilterUnreadNotifications() : List<Notification>
+ NotificationController()
+ ProcessNotification(notification : Notification)
+ SendNotification(notification : Notification)

14.10.1 Methods

`FilterReadNotifications()` : `List <Notification>`

This looks at all the notifications the user has cached and read and returns a list of them.

`FilterUnreadNotifications(): List<Notification>`

This looks at all the notifications the user has cached and not read and returns a list of them.

`NotificationController()`

This is a constructor.

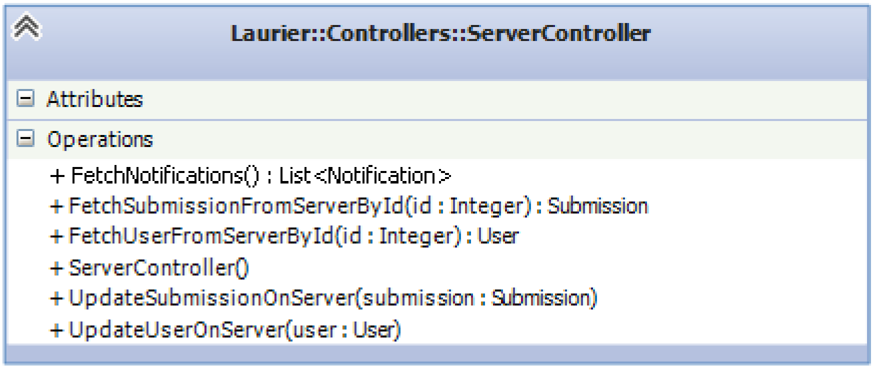
`ProcessNotification(notification: Notification)`

This is used to process a notification and decide what users to send it to.

`SendNotification(notification: Notification)`

This is used to send notifications to specific users.

14.11 ServerController



```
+ FetchNotifications() : List<Notification>
+ FetchSubmissionFromServerById(id : Integer) : Submission
+ FetchUserFromServerById(id : Integer) : User
+ ServerController()
+ UpdateSubmissionOnServer(submission : Submission)
+ UpdateUserOnServer(user : User)
```

14.11.1 Methods

`FetchNotifications(): List<Notification>`

Gathers notifications for use by the server.

`FetchSubmissionFromServerById(id:Integer): Submission`

Grabs a specific notification from the server using a question id to locate it

`FetchUserFromServerById(id: Integer): User`

Fetches a user account from the server by using the users id

`ServerController()`

A constructor for the server controller.

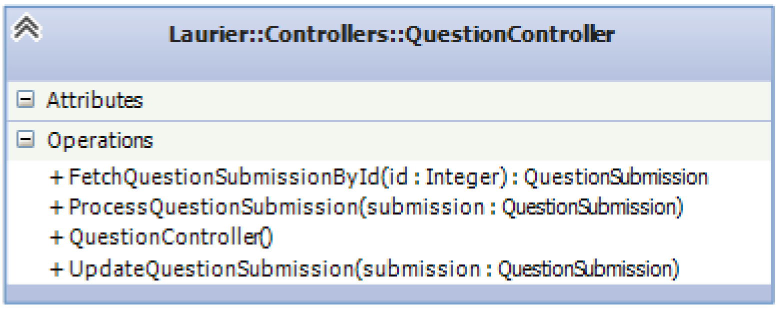
`UpdateSubmissionOnServer(submission: Submission)`

Takes a submission Object and uses its id to find and then overwrite the Submission data that was on the server with the **`Submission`** Object passed in to this function.

`UpdateUserOnServer(user:User)`

Takes a **`User`** entity object and uses its ID to find and then overwrite the **`User`** entity object that was on the server with the **`User`** entity object passed in to this function.

14.12 QuestionController



Laurier::Controllers::QuestionController

+FetchQuestionSubmissionById(id : Integer) : QuestionSubmission
+ProcessQuestionSubmission(submission : QuestionSubmission)
+QuestionController()
+UpdateQuestionSubmission(submission : QuestionSubmission)

14.12.1 Methods

`FetchQuestionSubmissionById(id : Integer): QuestionSubmission`

Given an integer ID, will fetch and return a **`QuestionSubmission`** from the remote server with the given ID.

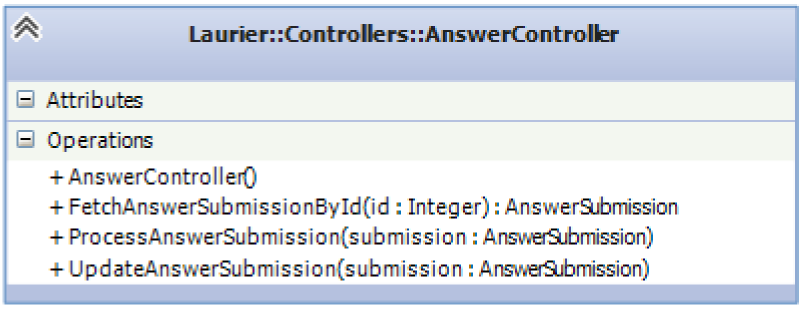
`ProcessQuestionSubmission(submission: QuestionSubmission)`

Given a submission object, will process and setup the submission for client usage.

`UpdateQuestionSubmission(submission :QuestionSubmission)`

Given the specified submission object, the remote will be updated.

14.13 AnswerControl


Laurier::Controllers::AnswerController

Attributes

Operations

+AnswerController()
+FetchAnswerSubmissionById(id : Integer) : AnswerSubmission
+ProcessAnswerSubmission(submission : AnswerSubmission)
+UpdateAnswerSubmission(submission : AnswerSubmission)

14.13.1 Methods

`FetchAnswerSubmissionById(id : Integer): AnswerSubmission`

Given an integer ID, will fetch and return a **`AnswerSubmission`** from the remote server with the given ID.

`ProcessAnswerSubmission(submission: AnswerSubmission)`

Given a submission object, will process and setup the submission for client usage.

`UpdateAnswerSubmission(submission : AnswerSubmission)`

Given the specified *submission* object, the remote will be updated.

15 Interface Design

In the following sections, we will describe some aspects of the interface. The below sections are mockups are do not have to be followed completely but are generally what has been decided as the UX experience.

15.1 Brief Overview of User Interactions

As previous described, the application is a mobile application that runs on the Android smartphone platform. Naturally, the main source of input throughout the application will be the touchscreen. Using the touchscreen, the user will interact with various widgets on screen to navigate the mobile application. The user will have the functionality of interacting with a basic forum like interface with the focus on being able to pull in question and answer data and push it out quickly.

In most of the screens, the user will see a **Header.** This **Header** contains a few small utility buttons: the **home button**, **profile button**, and **notification button**. These are the main three ways of switching flow throughout the application. When clicking one of these, you will be brought to a screen with a new flow. It could be said that these buttons break the application up into three miniature applications. For example, once the user has clicked the **profile button** and been brought to the **profile page**, the scope of the application has been limited to profile like actions until the user decides to click another **header** button.

The **header** is made up of:

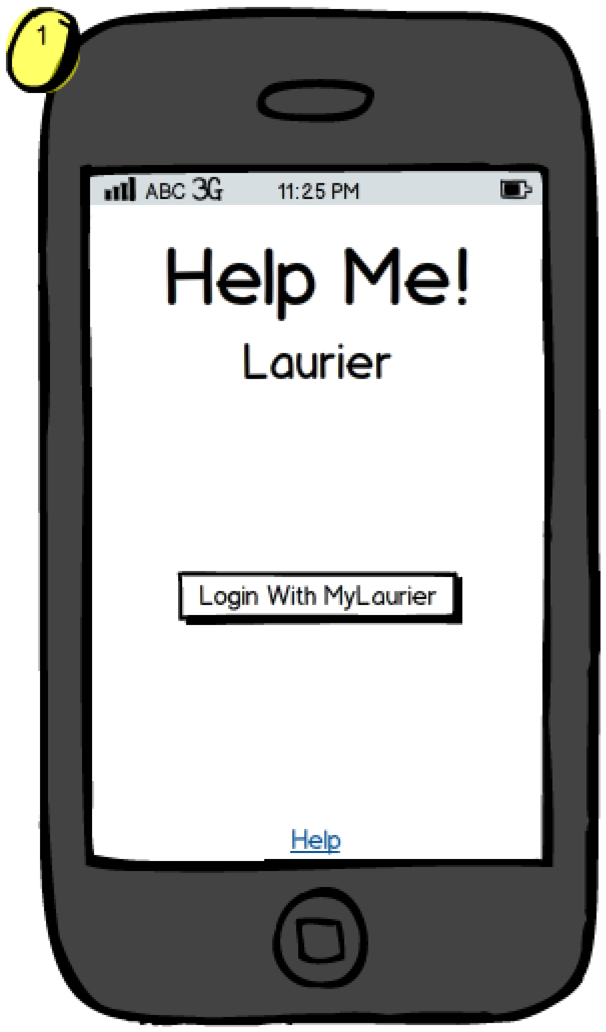
- **Home button**-A button that takes the user back to the **Home Page** (3)
- **Alert button** - A button that displays the current amount of notifications the user has and will redirect them to the **Alerts Page (4)** when clicked
- **Profile button** - A button that takes users to the **Profile Page** (5)

In all screens, a **footer** is also provided. Each page with this **footer** will include a help link at the bottom which will redirect the user to legal agreements, usage agreements and **Usage Instructions** (11).
This also breaks into its own context except leaving the screen pops back into the previous context. More information is available on page (11).

In the below documentation, sometimes screen and page are used synonymously. For the sake of this document, you can treat them as the same. That is, a page is the same as a screen. A page represents an entire state that a user sees at one given time.

15.2 Screen Images and Description

15.2.1 Landing Page (1)



The **Landing Screen** is the first page that the user will encounter when opening Help Me! Laurier. On this page the user will touch the **`LoginButton`** and be sent to **the Login Page** (2).

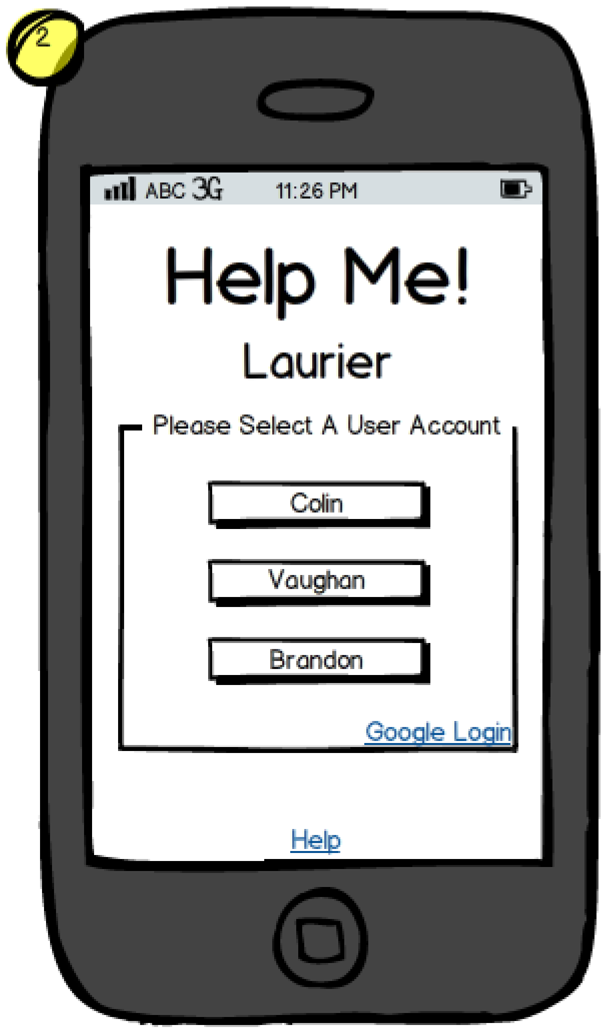
This screen will generally provide introductory information not shown that can be shown again when clicking the “Help” hyperlink.

Notes:

- Generally, this screen will only be shown once per user. Once the user is logged in, this screen will not be displayed again.

- This screen is the entry point of the application when the user is not logged in. When the user is logged in, the user will be shown the **Home Page** (3)
- The logo is pending and may be changed depending on the resources provided at time of application construction.

15.2.2 Login Page (2)

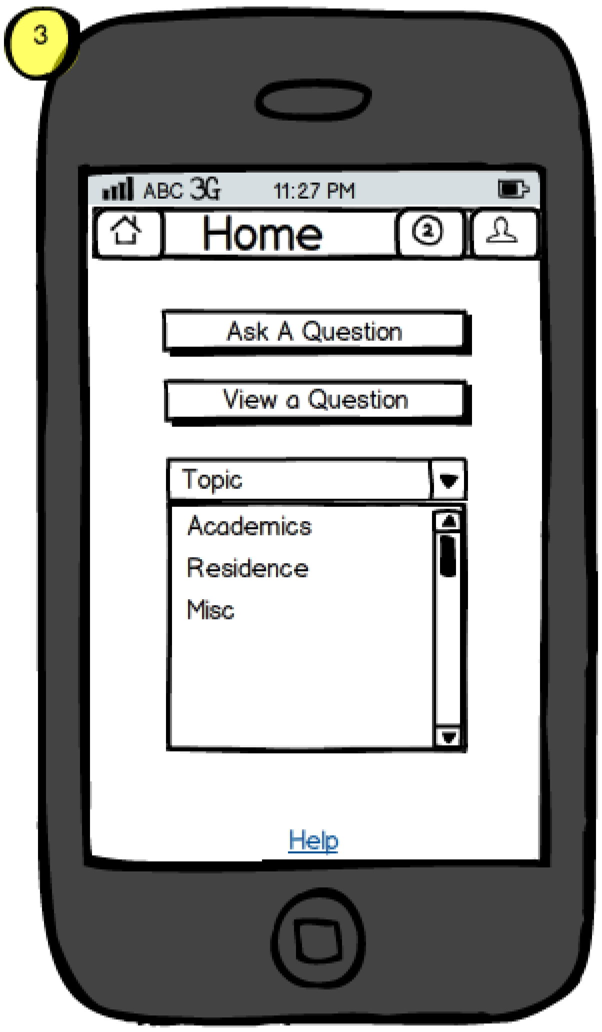


The **Login Page** will allow the user to enter their Laurier username and password if they are not currently logged into their smartphone. Otherwise, a popup will appear asking the user to select the account they wish to use if multiple accounts are logged into the Android smartphone. After selecting an account, the user will be redirected to the **Home Page (3)**. Otherwise, the user will see a prompt and stay on this screen.

Notes:

- Google will provide a second modal dialog that is not shown on this screen. It is not possible to show the exact dialog as it will vary from device to device and the version of the page that Google has chosen to display.
- The link "**Google Login**" redirects to a 3rd party.

15.2.3 Home Page (3)

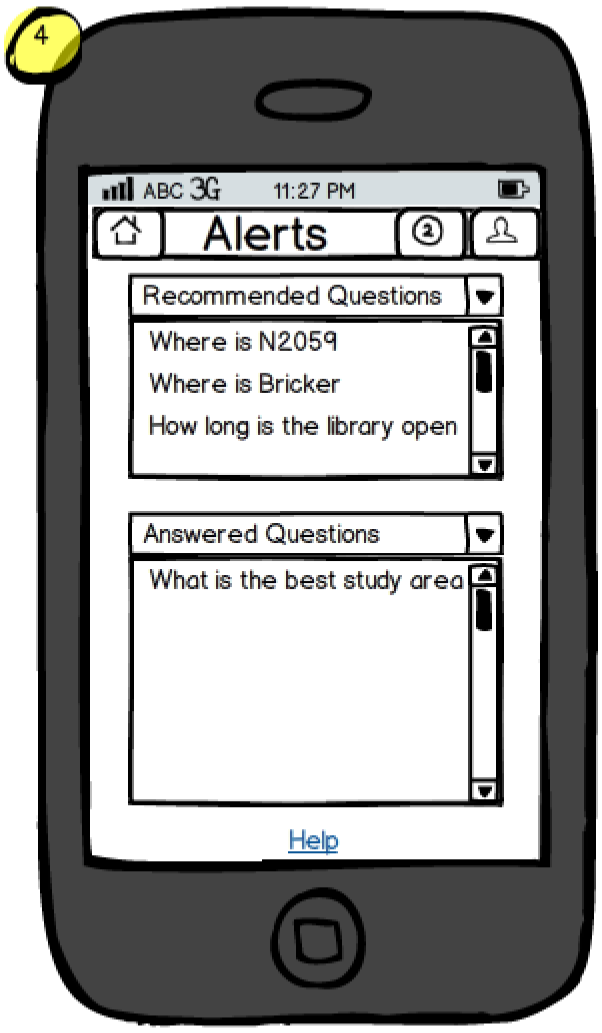


The **Home Page** contains a scrolling dropdown box which will allow the user to select a category of interest. Following this selecting the user can choose to “**Ask a question**” (9) or “**View questions**” (6) for that category. The user must select an item from the list first and then click the appropriate buttons. The buttons will be grayed and disabled without first selecting a category from the scrolling list.

Notes:

- The categories above aren’t the exact categories that will necessarily be used in the application. The information here is provided only for illustrative purposes.

15.2.4 Alerts Page (4)

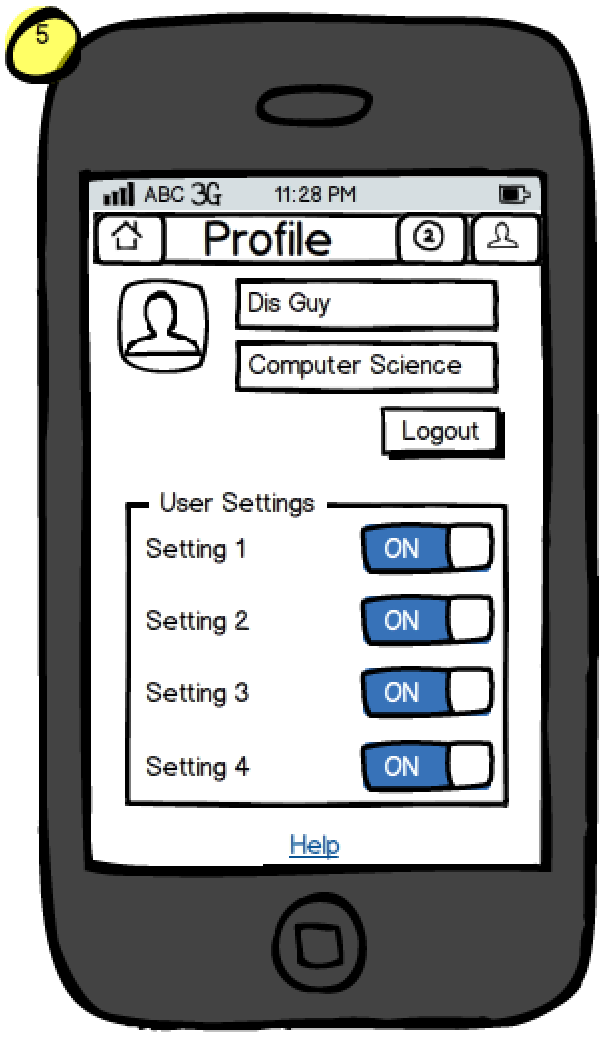


The **Alerts Page** contains 2 scrollable lists, one containing questions that are recommended for the user to answer and the other containing answers regarding questions the user inquired about. The user can select an item from either list and will be redirected to view the **Question Page** (7) for the according submission.

Notes:

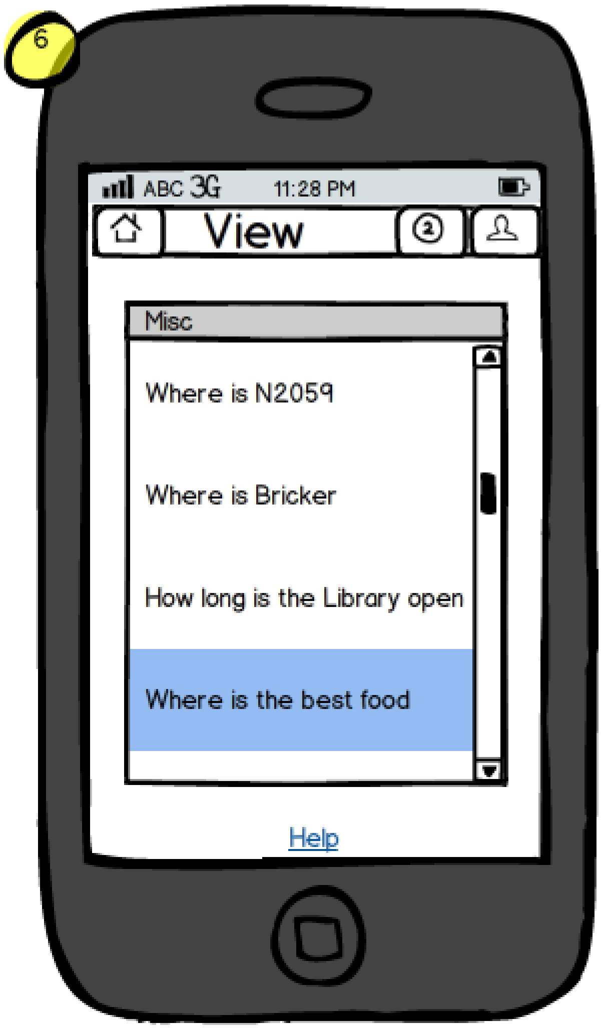
- A header bar will not be displayed if there are no alerts / notifications of that type to be shown. The list box and header will be hidden.

15.2.5 Profile Page (5)



The **Profile Page** contains information that the user can fill out to better accommodate the user experience. This page will also contain settings, a user image, and other various options. The user cannot transit off this screen until selecting a different button from the **Header,** as described in the introduction.

15.2.6 View Questions (6)

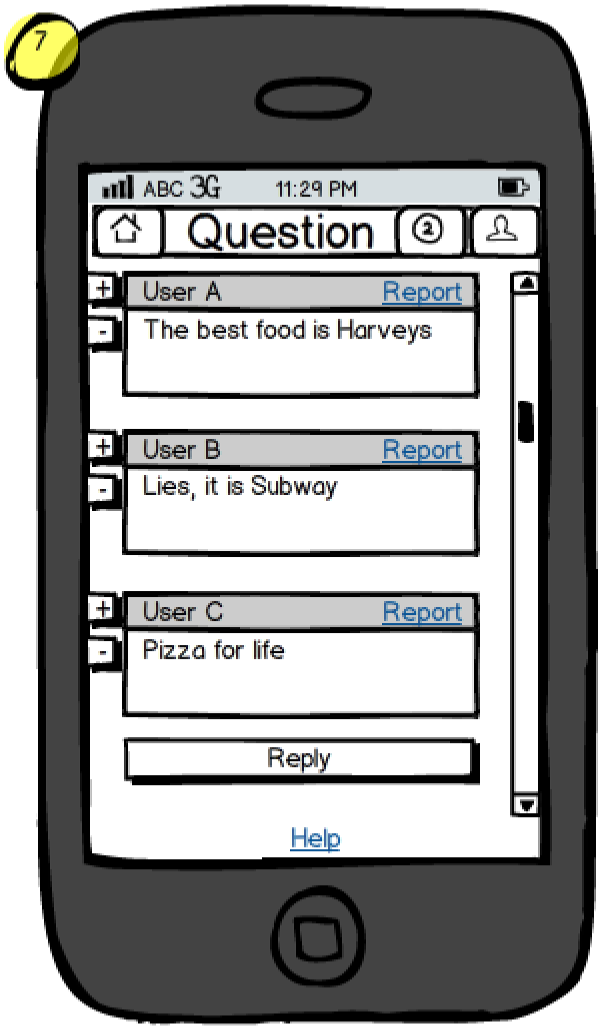


The **View Questions Page** will contain a list of questions pertaining to the users chosen category. When the user selects a topic they are redirected to the view **Question Page** (7).

Special Notes:

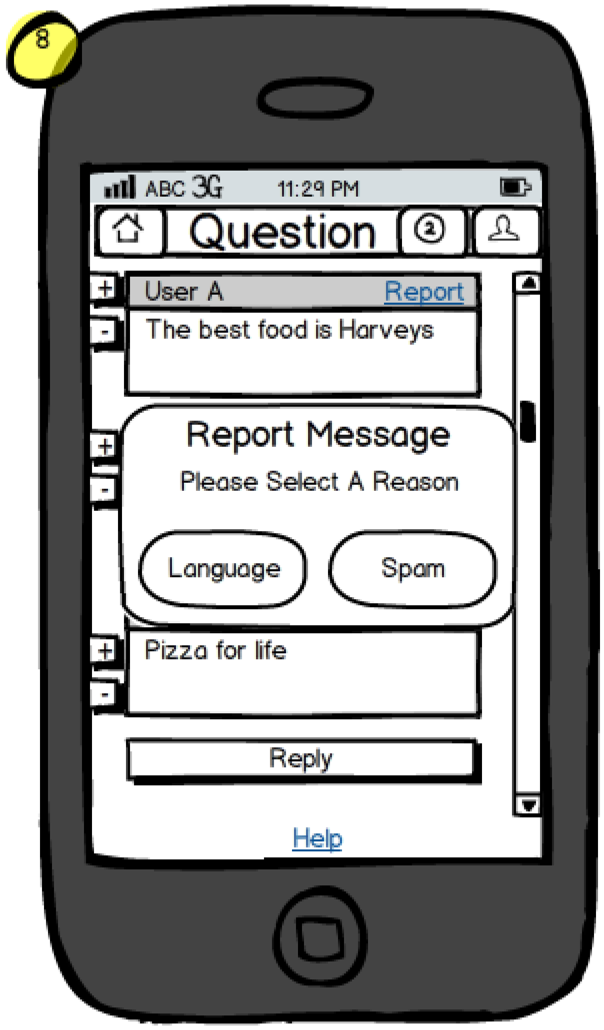
- As noted above, the category and questions provided are only used for illustrative purposes.

15.2.7 View Question (7)



The **View Question** page contains the asked question at the top followed by answers suggested by users. For each of the individual cards the user can either up vote, down vote or report the question/answer. At the bottom of the page will be a reply button allowing the user to post a reply given that the question is not locked. This will redirect them to the **Reply Page** (10)

15.2.8 Report Question (8)

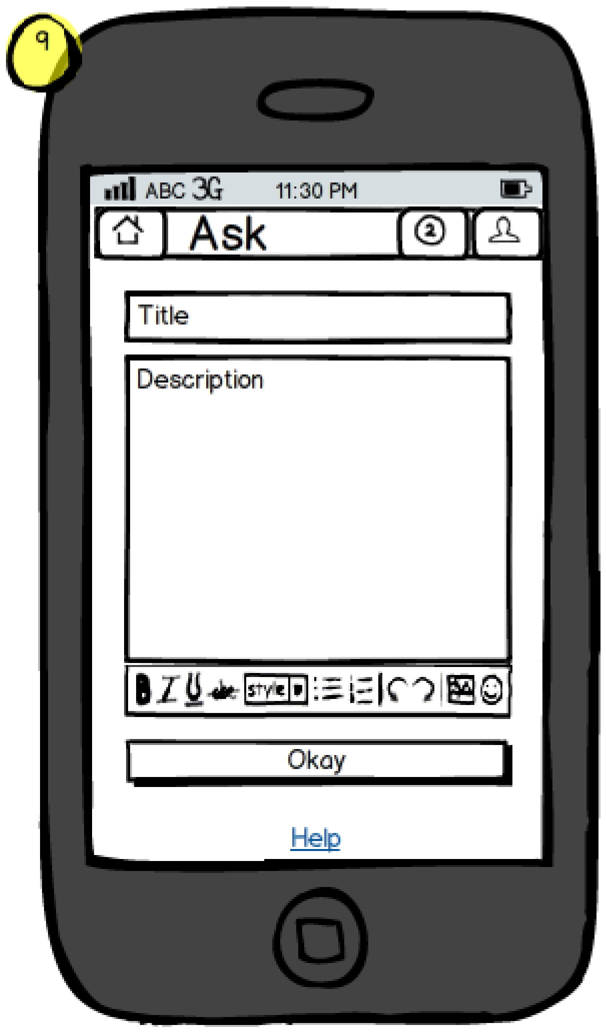


If the user see's that the content is unsuitable for a viewing audience the user can select the report button. Once the report button has been selected the user will be able to select a reason for reporting such as language or spam. This is a modal dialog, so flow exists back to the **View Question** (8) page when completed.

Notes:

- The reasons listed for reporting a question may not exhaustive or illustrative of the exact options. Please refer to the according documentation before implementing as shown.

15.2.9 Ask Question (9)

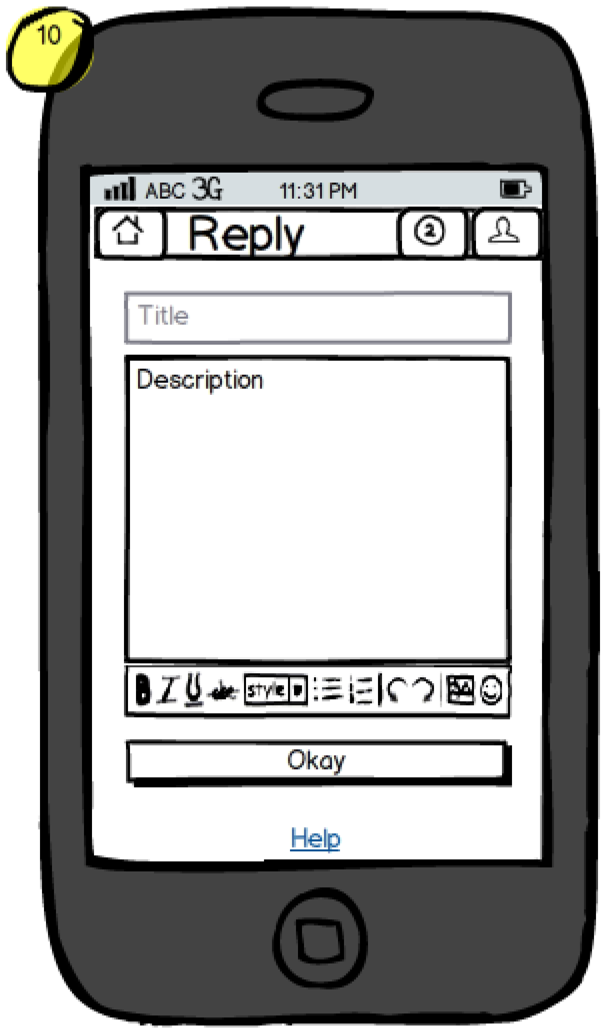


The **Ask Question** page will allow a user to submit a new question by filling in a title and a description of their issue. After the user leaves the description box, *Help Me! Laurier* will try to suggest questions that are similar to ensure repetitive questions are not being asked. If none of the suggestions are relevant, the user can continue and select the "**Okay**" button to submit the question. The user will then be redirected back to the **View Question** page (7).

Notes:

- The editing controls above are not illustrative of the exact controls that will be provided. These will be determined by the rich editor controls that are selected to be used within the application.

15.2.10 Reply Question (10)

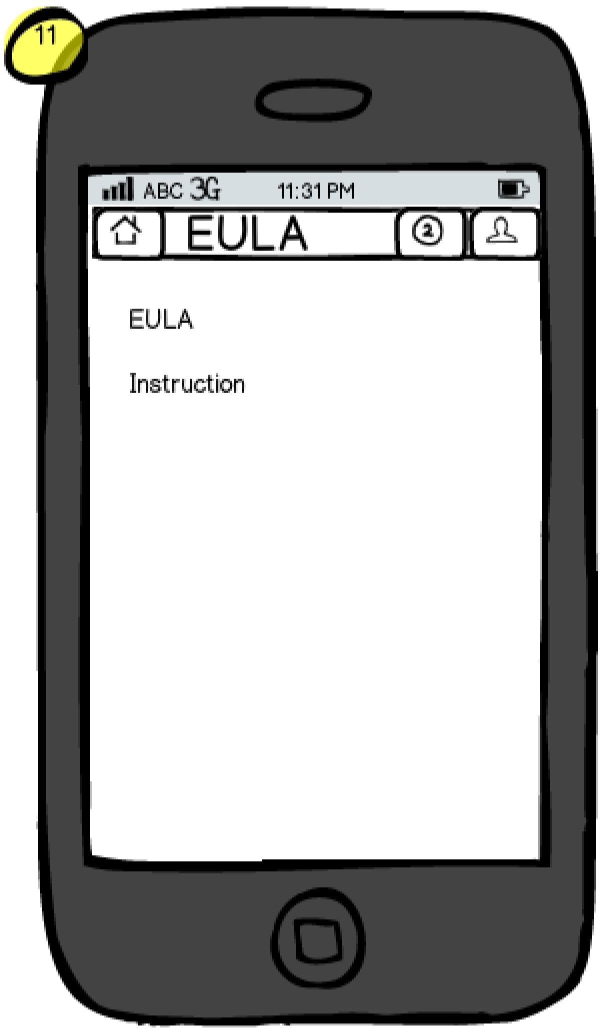


The **Reply Question** is a page that allows a user to reply to a given question by filling out the description textbox and selecting “**Okay**” which will submit the reply. The user will then be redirected back to the **View Question** page (7).

Notes:

- It should be noted that the “Title” textbox is grayed out here to illustrate it similar to that of the “Ask a question” page, without the title box. In practice, this box should be not visible to the user.

15.2.11 Usage Instructions (11)



The EULA and help screen will include information about general usage of the application and license agreements that users will need to know. This includes important information about how content in the application is stored and how users can expect their data to be used.

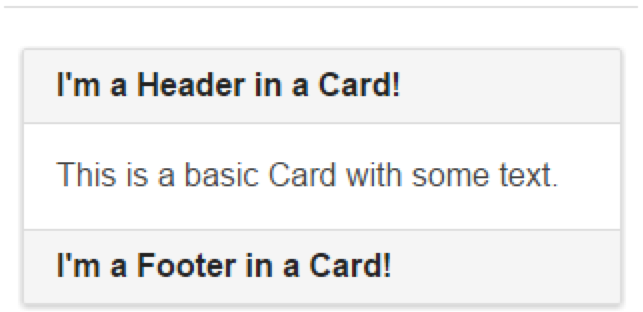
Notes:

- Instructions and a full EULA are too exhaustive and comprehensive to display here. See further documentation for information regarding licensing. You can find these in **Appendix A.**

15.3 Listing of Major Ionic Components

In this section, we will briefly go over some of the major components that are provided by the **Ionic Framework** to help develop the application interface.

15.3.1 Card

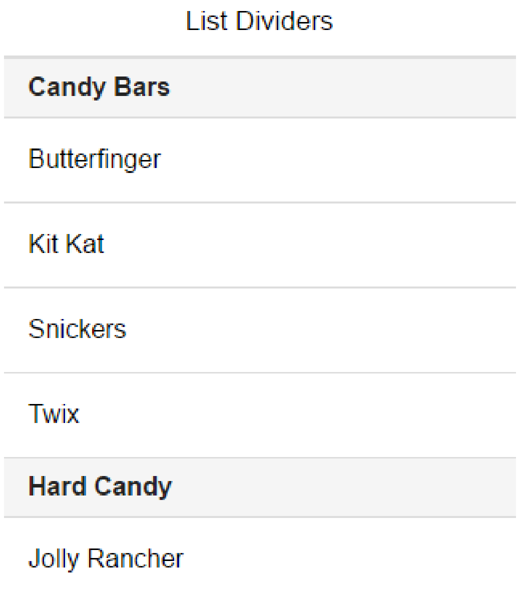

I'm a Header in a Card!

This is a basic Card with some text.

I'm a Footer in a Card!

Frequently mentioned and shown throughout the various application slides, the **Ionic Framework** card provides a quick and easy way to provide card interfaces. These should be used where you see them. For examples, see **(7)**. (Read more: http://ionicframework.com/docs/components/#card-headers-footers)

15.3.2 List and Dividers


List Dividers

Candy Bars

Butterfinger

Kit Kat

Snickers

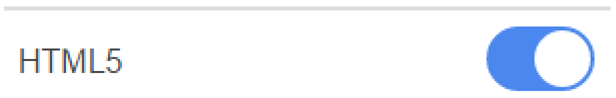
Twix

Hard Candy

Jolly Rancher

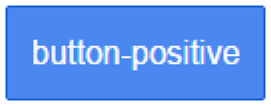
Where they are seen accordingly, such as the **Notifications / Alerts** view, these types of list views from the **Ionic Framework** should be used. (Read more: http://ionicframework.com/docs/components/#item-dividers)

15.3.3 Toggles



Where toggles are shown, use the ones from the **Ionic Framework.** (Read more: http://ionicframework.com/docs/components/#toggle)

15.3.4 Buttons



(Read more: http://ionicframework.com/docs/components/#buttons)

15.3.5 Header

(Read more: http://ionicframework.com/docs/components/#header)

15.3.6 Footer

(Read more: http://ionicframework.com/docs/components/#footer)

Appendices

16 Appendix A: EULA

The licensing agreement has been determined by the stipulations of the project. Enclosed below is a copy of the EULA that should be used within the application, which can be taken into account during development:

BY DOWNLOADING, INSTALLING OR USING THE APP YOU ACKNOWLEDGE AND AGREE THAT:

1. Your use of the App in conjunction with the *Help Me! Laurier* service is solely at your own risk;
2. All content found on the *Help Me! Laurier* App is not the personal opinion of the creators.
3. The App is licensed, not sold to you and you may use the App only as set forth in this EULA;
4. You consent to the collection, use, sharing and transfer of your personally identifiable information, including the transfer and processing of your information outside your home country;
5. You acknowledge that third party terms and fees may apply to the use and operation of your mobile device in connection with your use of the App, such as your carrier's terms of service, and fees for phone service, data access, or messaging capabilities, and that you are solely responsible for payment of any and all such fees;
6. As set forth in this EULA, the App is provided "as is";
7. "WLU", "Wilfrid Laurier" and "Laurier" are all property of Wilfrid Laurier University.

IF YOU DO NOT AGREE TO BE BOUND BY ABOVE TERMS (AS DESCRIBED IN FURTHER DETAIL BELOW) YOU MAY NOT DOWNLOAD, INSTALL OR USE THE APP.

MODIFICATION OF THE EULA. The *Help Me! Laurier* creators reserve the right to modify and/or change any of the terms and conditions of this EULA at any time and without prior notice. If *Help Me! Laurier* creators materially modifies this EULA it will make reasonable efforts to notify you of the change. For example, we may send a message to your email address, if we have one on file, or generate a pop-up or similar notification when you access the App or the Service for the first time after such material changes are made. By continuing to use the App after *Help Me! Laurier* creators have posted a modification of this EULA, you agree to be bound by the modified EULA. If the modified EULA is not acceptable to you, your only recourse is to discontinue the use of and uninstall the App. This Agreement will also govern any software upgrades and/or updates provided by *Help Me! Laurier* creators that upgrade and/or supplement the App, unless such upgrades and/or updates are accompanied by a separate license, in which case the terms of that separate license will apply.

USE RESTRICTIONS. You may not use the App in any manner that could: (i) damage, disable, overburden, or impair the App (or any server or networks connected to the App), or (ii) interfere with any third party's use and/or enjoyment of the App (or any server or networks connected to the App). Except as expressly specified in this EULA, you may not: (a) copy or modify the App; (b) transfer, sublicense, lease, lend, rent or otherwise distribute the App to any third party; or (c) use the App in any unlawful manner, for any unlawful purpose, or in any manner inconsistent with this EULA. You acknowledge and agree that portions of the App, including, without limitation, the source code and the specific design and structure of individual modules or programs, constitute or contain trade secrets of *Help Me! Laurier*. Accordingly, you agree not to disassemble, decompile or otherwise reverse engineer any components of the App.

LIABILITY. All information on *Help Me! Laurier* is not the personal opinion of the creators. The *Help Me! Laurier* creators are not liable for any actions caused by the use/misuse of anything found on the App. You agree to hold *Help Me! Laurier's* creators, its affiliates, officers, directors, employees, agents, and third party service providers harmless from and defend them against any claims, costs, damages, losses, expenses, and any other liabilities, including attorneys' fees and costs, arising out of or related to your access to or use of *Help Me! Laurier*, your violation of this user agreement, and/or your violation of the rights of any third party or person.

17 APPENDIX B: REVISIONS

17.1 REVISION HISTORY

October 27th, 2014

Person	Accomplishments
Vaughan Hilts, Brandon Smith, Colin Gidzinski	• Discussed use cases • Discussed the overall app

October 28th, 2014

Person	Accomplishments
Vaughan Hilts	• Selected jobs in skill matrix • Wrote Framework of the requirements • Discussed Use Cases for program
Brandon Smith	• Assisted in writing documentation • Discussed Use Cases for program
Colin Gidzinski	• Assisted in writing documentation • Discussed Use Cases for program • Selected jobs in skill matrix

October 29th, 2014

Person	Accomplishments
Brandon Smith	• Proofread • Selected jobs in skill matrix

October 30th, 2014

Person	Accomplishments
Vaughan Hilts	• Discussed and worked out Boundary entity and control objects • Fixed document based on proofreading
Brandon Smith	• Discussed and worked out Boundary entity and control objects • Fixed document based on proofreading
Colin Gidzinski	• Did paper mockups of the UI

November 4th, 2014

Person	Accomplishments
Vaughan Hilts	• Discussed and worked out Boundary entity and control objects
Brandon Smith	• Discussed and worked out Boundary entity and control objects
Colin Gidzinski	• Did paper mockups of the UI

November 5th, 2014: Worked on the “Analysis” document individually

Person	Accomplishments
Vaughan Hilts	• Created the frame and a significant portion of the document
Brandon Smith	• Created several sequence diagrams
Colin Gidzinski	• Created several sequence diagrams

November 6th, 2014: Complete documentation for the entire "Analysis" document during this date. Some of it had come together prior but many diagrams had to be created still and some sections required fleshing out.

Person	Accomplishments
Vaughan Hilts	• Finished documentation of controllers and some boundary objects • Created more sequence diagrams to be inserted into the final report • Added various documentation regarding scope, entity attribution and persistence • Demoed and trialed various wireframing software packages
Brandon Smith	• Proofreading of entire report was done. • Edited Activity Diagram and then modeled it using yEd • Various clean-up of sequence diagrams was completed • Demoed and trialed *Ninja Mockups*
Colin Gidzinski	• Completed the "Activity Diagram" that was to be inserted into the final report as an overall draft • Assisted in editing of paper documentation • Demoed and trialed *Balsamiq Mockups.* This was selected as our software.

November 10th 2014: Started on design documentation

Person	Accomplishments
Vaughan Hilts	• Created Documentation
Brandon Smith	• Looked over and critiqued UI design
Colin Gidzinski	• Created Mockups for UI design

November 11th: Worked on design document

Person	Accomplishments
Vaughan Hilts	• Did introduction of design document
Brandon Smith	• Attempted dataflow diagrams
Colin Gidzinski	• Wrote descriptions of UI diagrams

November 12th

Person	Accomplishments
Vaughan Hilts	• Completed the system overview
Brandon Smith	• Completed dataflow diagrams
Colin Gidzinski	• Split up the UI diagrams

November 13th

Person	Accomplishments
Vaughan Hilts	• Completed the system architecture, formatting, edited the UI descriptions, and did sub system breakdown.
Brandon Smith	• Completed the data dictionary and object break down
Colin Gidzinski	• Explained absence

November 18th

Person	Accomplishments
Vaughan Hilts	• Discussed and edited existing documentation
Brandon Smith	• Discussed and edited existing documentation
Colin Gidzinski	• Discussed and edited existing documentation

November 20th

Person	Accomplishments
Vaughan Hilts	• Edited Use Case entry, exit conditions and edited use cases to reflect new functionality • Added new “problem statement”
Brandon Smith	• Worked on new data flow diagrams • Researched appropriate software for graph editing in order to create proper data flow diagrams
Colin Gidzinski	• Procured list of edits required for Analysis document

November 25th: A day of more report polish

Person	Accomplishments
Vaughan Hilts	• Modified state diagram entry points to reflect properly • Modified Analysis section to include some extra information • Planned and generated UML diagrams for various applications portions
Brandon Smith	• Complete report proof reading • Polished and fixed "Activity Diagram" • Led discussion regarding OAuth considerations during testing
Colin Gidzinski	• Created cover page and graphics for report, back page • Formatting and binding researching for publication • Coordinated changes required for analysis stage

November 26th: Complete Final Documentation was completed

Person	Accomplishments
Vaughan Hilts	• Wrote extra sections in the System Architecture • Finished Cohesion and Coupling Analysis • Wrote style and design guidelines • Formatted, cleaned up report
Brandon Smith	• Completed additional report proof reading • Worked on more object decomposition
Colin Gidzinski	• Colin was involved in development on the nearly final specification.

17.2 Schedule Agreeance

Generally speaking, the work was divided approximately:

Person	Percentage
Vaughan Hilts	40%
Brandon Smith	30%
Colin Gidzinski	30%

I, ____ agree to the above timeline is how the project was divided.

Signature

I, ____ agree to the above timeline is how the project was divided.

Signature

I, ____ agree to the above timeline is how the project was divided.

Signature

18 APPENDIX C: TIMELINE MANAGEMENT

All developers will meet bi-weekly, on Tuesdays and Thursdays throughout the project from 1PM – 4PM. It should be noted additional individual work will sometimes be completed to optimize the quality and clarity of documentation, especially during the implementation phases.

Tuesday	Thursday
October 28th: Begin development of the requirements as a group. Create concrete use cases, skill matrix, create timeline.	October 30th: Polished requirements and agree on any ambiguous conditions and prepare final revision of requirements documentation.
November 4th: Meet about preliminary analysis and begin working on. Finish during time this time. It may extend past the normal hours.	**No meeting**
November 11th: Meet about design and begin fleshing out CRCs.	**November 13th**: Polish design report; prepare to have a final good idea of the architecture of the application.
November 18th: Work on completed report and finish putting it together. Complete some work on implementation.	**November 20th:** Ensure completed report is ready. Complete more work on implementation.
No meeting	**November 27th**: Begin wrapping up the implementation of the project. Perform final testing, ensure stability of software

I, ____ agree to the above timeline.

Signature
____

I, ____ agree to the above timeline.

Signature
____

I, ____ agree to the above timeline.

Signature
____

19 Appendix D: Work Division

The documentation above outlines everything performed by each member at every date, but below gives a general overview of everything agreed on to be completed:

Vaughan Hilts

- Generally, in charge of documentation formatting and report write up.
- Leaded discussion regarding picking technology stacks and design implementations
- Guided communication and in charge of source control management
- Use case write ups, introductory statements
- Contributions will include: **System Architecture, Object Design, UML diagrams, state diagrams**

Brandon Smith

- Data Flow Diagrams, Sequence Diagrams
- Generally, in charge of taking paper mockups of various diagrams and converting them using **yEd**
- In charge of proofreading documents to ensure quality
- During the design section, in charge of object breakdown and maintaining the data dictionary

Colin Gidzinski

- In charge of creation of user interface, mockups, and graphical work
- In charge of report publishing, including back and front covers
- Sequence Diagrams, Activity Diagrams

The work division has been agreed upon to be fair, the following signatures indicate willingness to work on the listed tasks:

I, ____ agree to the above work division.

Signature

I, ____ agree to the above work division.

Signature

I, ____ agree to the above work division.

Signature

This page is intentionally left blank.