# **Software Object-Oriented Design**

*Help Me! Laurier*

Vaughan Hilts (120892740)   
Brandon Smith (120201510)   
Colin Gidzinski (120663460)

***Group #1***

***Dr. Chinh Hoang***

Table of Contents

[Introduction 4](#_Toc403656301)

[Purpose 4](#_Toc403656302)

[Overview 4](#_Toc403656303)

[System Overview 4](#_Toc403656304)

[System Architecture 5](#_Toc403656305)

[Data Design 5](#_Toc403656306)

[Component Design 5](#_Toc403656307)

[Interface Design 5](#_Toc403656308)

[Further Reading / Technology Glossary 6](#_Toc403656309)

[System Overview 7](#_Toc403656310)

[Client Overview 7](#_Toc403656311)

[Server Overview 9](#_Toc403656312)

[System Architecture 11](#_Toc403656313)

[High Level Overview 11](#_Toc403656314)

[Detailed Subsystem Analysis 12](#_Toc403656315)

[User Interface System 12](#_Toc403656316)

[Account System 13](#_Toc403656317)

[OAuth System 13](#_Toc403656318)

[Submission System 14](#_Toc403656319)

[Question Transit System 14](#_Toc403656320)

[Notification System 15](#_Toc403656321)

[Report System 15](#_Toc403656322)

[Database System 16](#_Toc403656323)

[Data Design 17](#_Toc403656324)

[Data Flow with Objects 17](#_Toc403656325)

[QuestionSubmission 17](#_Toc403656326)

[AnswerSubmission 18](#_Toc403656327)

[AbuseReport 19](#_Toc403656328)

[Data Dictionary 20](#_Toc403656329)

[Object Breakdown 23](#_Toc403656330)

[AbuseReport 23](#_Toc403656331)

[QuestionSubmission 24](#_Toc403656332)

[AnswerSubmission 25](#_Toc403656333)

[Interface Design 26](#_Toc403656334)

[Brief Overview of User Interactions 26](#_Toc403656335)

[Screen Images and Description 27](#_Toc403656336)

[Landing Page (1) 27](#_Toc403656337)

[Login Page (2) 29](#_Toc403656338)

[Home Page (3) 30](#_Toc403656339)

[Alerts Page (4) 31](#_Toc403656340)

[Profile Page (5) 32](#_Toc403656341)

[View Questions (6) 33](#_Toc403656342)

[View Question (7) 34](#_Toc403656343)

[Report Question (8) 35](#_Toc403656344)

[Ask Question (9) 36](#_Toc403656345)

[Reply Question (10) 37](#_Toc403656346)

[Usage Instructions (11) 38](#_Toc403656347)

# Introduction

## 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.

## 

## 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:

### 

### 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.

### 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.

### 

### 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.

### Component 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.

### 

### 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.

## 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/>)

**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.

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

# 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.

## 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.

## 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.

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.

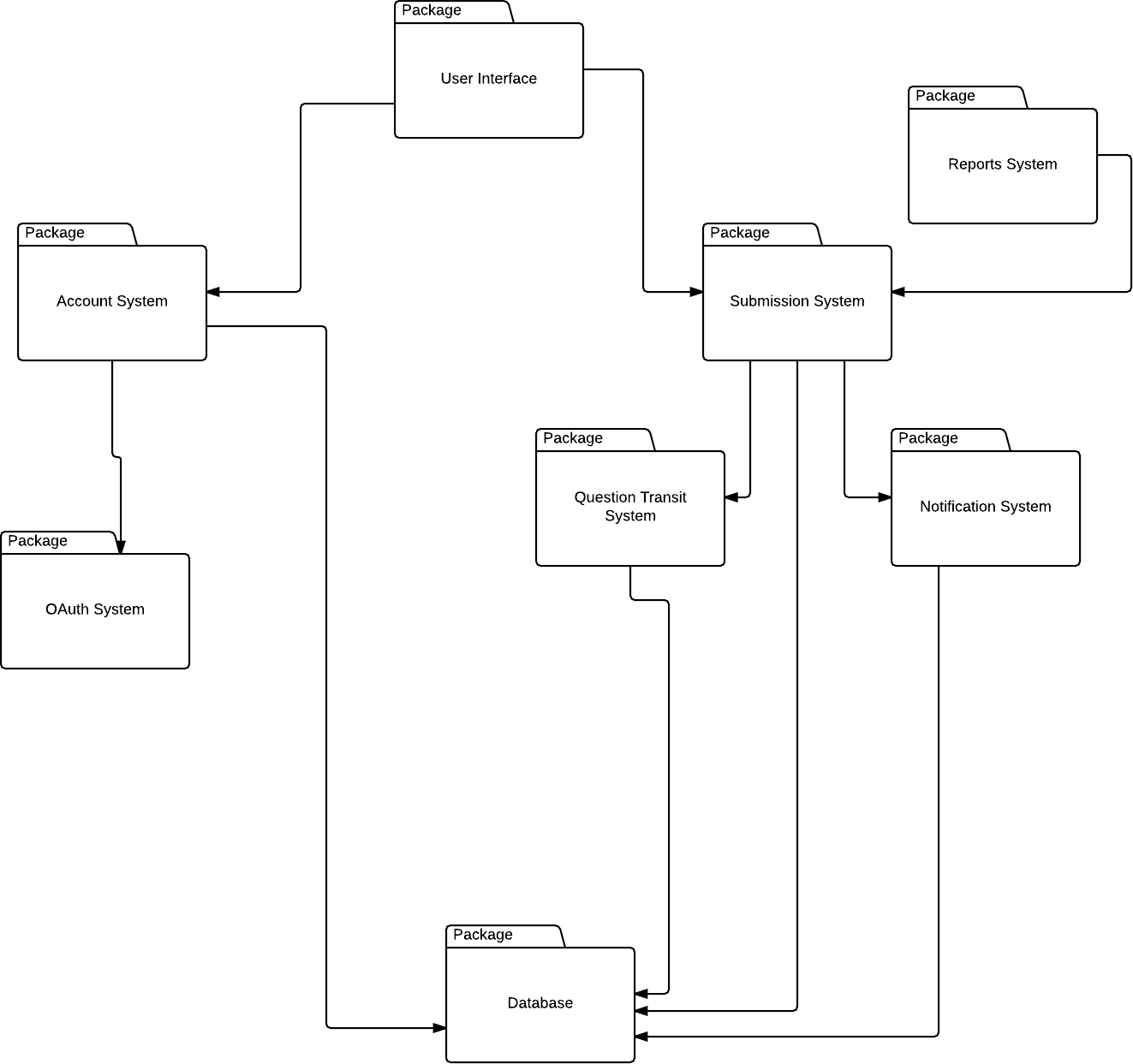
# System Architecture

## 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.

## Detailed Subsystem Analysis

### User Interface System

#### 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.

#### 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 |  |  |

### Account System

#### 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.

#### CLass Breakdown

|  |  |  |
| --- | --- | --- |
| Boundary | Entity | Control |
| LoginPage | **AuthenticationToken** | **ServerControl** |
| WelcomePage | **User** | **AuthenticationControl** |
| LoginForm |  |  |
|  |  |  |

### OAuth System

#### 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**.

#### CLass Breakdown

|  |  |  |
| --- | --- | --- |
| Boundary | Entity | Control |
|  | **AuthenticationToken** | **ServerControl** |
|  | **User** | **AuthenticationControl** |
|  |  |  |
|  |  |  |

### Submission System

#### 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.**

#### CLass Breakdown

|  |  |  |
| --- | --- | --- |
| Boundary | Entity | Control |
| QuestionPage | **AuthenticationToken** | **AnswerControl** |
| SubmissionPage | **User** | **QuestionControl** |
| ViewTopicsPage | **Submission** |  |
|  | **Question** |  |
|  | **Answer** |  |
|  |  |  |

### Question Transit System

#### 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.

#### CLass Breakdown

|  |  |  |
| --- | --- | --- |
| Boundary | Entity | Control |
|  | **User** | **ServerControl** |
|  | **Submission** | **QuestionControl** |
|  | **Question** |  |
|  | **Answer** |  |
|  |  |  |
|  |  |  |

### Notification System

#### 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.

#### CLass Breakdown

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

### Report System

#### 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**.**

#### CLass Breakdown

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

### Database System

#### 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.

#### CLass Breakdown

|  |  |  |
| --- | --- | --- |
| Boundary | Entity | Control |
|  | **User** | **ServerControl** |
|  | **Submission** | **QuestionControl** |
|  | **Question** | **NotificationControl** |
|  | **Answer** |  |
|  | **AbuseReport** |  |
|  | **Notification** |  |
|  | **ProfileQuestionEntry** |  |
|  | **AuthenticationToken** |  |

# Data Design

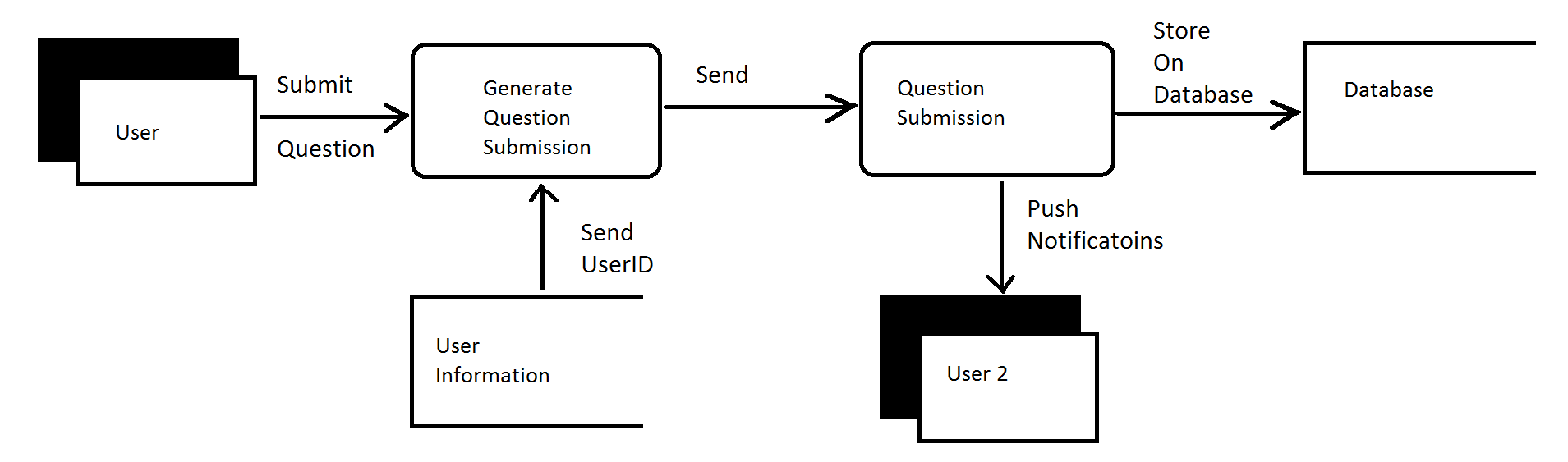
In the next few sections, different objects and how their data can be interacted with and manipulated are discussed. For each object, a brief description of the process and a diagram is provided. At the end of this section, a dictionary is provided with all elements and various attributes that describe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

## Data Flow with Objects

### 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.

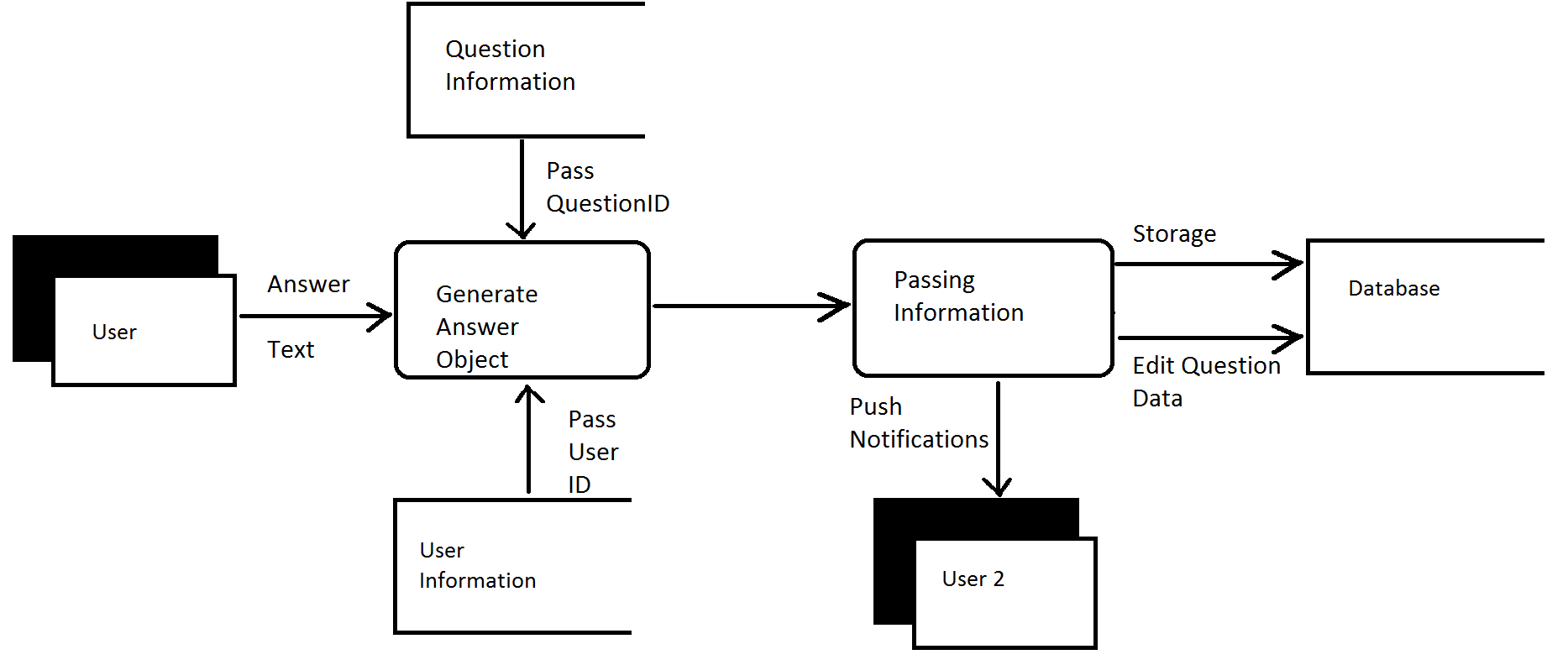
The system creates a **QuestionSubmission** object by setting **Title**, a string equal to the title textbox contents, setting **CategoryID,** an integer, to the selected category, initializing **QuestionAnswer**s, an array, for **AnswerID**sas an empty array, and generating the remainder of a **Submission** 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**, aJSdate 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:

****

### 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.

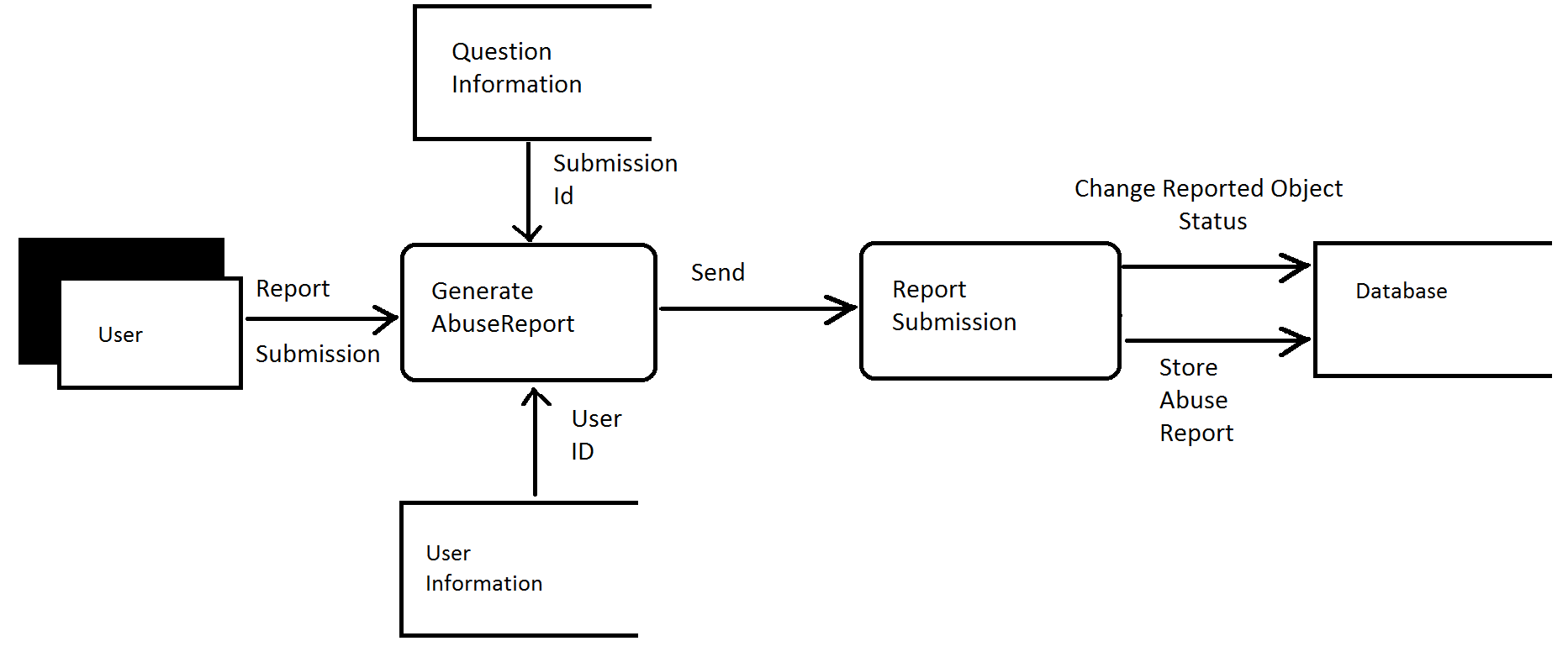
The **AnswerSubmission** object is created by setting **ParentQuestionID**, an integer, equal to the **AuthorID** of the question the user is answering and generating the remainder of a **Submission** object which **AnswerSubmission** objects inherit from. This involves setting **Body,** a string, equal 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 JS 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:



### 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 JS 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:



## 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 JS 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 |

# 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.

## AbuseReport

**AbuseReportID**

An integer ID generated by **generate\_reportID**

**Generate\_reportID**

The backend contains a Counter **AbuseID**s which is incremented by this function and this function returns an **AbuseReportID**

**AbuseIDs**

An integer stored within the backend

**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 JS date function. Its purpose is to keep track of when the **AbuseReport** is submitted which should help with giving faster feedback to the reports.

## QuestionSubmission

**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

**Body**

This is the bulk of the question. It is entered by the user and used as a parameter in the creation of a **QuestionSubmission** Object.

**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.

**Date**

This is a Date Object generated by the JS date function. Its purpose is to keep track of when the question is submitted which should help with giving faster feedback and retiring old questions

**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.

**ID**

An integer generated by **generate\_SubmissionID**

**Generate\_SubmissionID**

This function generates an **ID** for submission functions. To do this it takes a stored value, increments it by one and returns an **ID**. It is generated when the object is initialized.

**Status**

This contains three states. Upon creation it is initialized to active. If it receives a report it is moved to in review. If it is cleared from the report it is returned to active if it is not it is set to deleted. A question **Status** may also be changed to deleted when it gets too old or spends too long inactive.

**AuthorID**

This contains the **UserID** of the user who creates the submission. It is used in case users need to be located and in order to pass feedback from the other users, specifically answers.

**Score**

This is an integer initialized to zero. It can be increased or decreased by users pressing buttons on the Question in the UI to increase or decrease its score.

## AnswerSubmission

Comprised of:

**AuthorID**

This contains the **UserID** of the user who creates the submission. It is used in case users need to be located and in order to pass feedback from the other users, specifically answers.

**Body**

This is the bulk of the answer. It is entered by the user and used as a parameter in the creation of an **AnswerSubmission** Object and is passed in through a text box in the UI.

**Date**

This is a Date Object generated by the JS date function. Its purpose is to keep track of when the answer is submitted

**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.

**ID**

An integer ID generated by **Generate\_SubmissionID**

**Generate\_SubmissionID**

This function generates an **ID** for submission functions. To do this it takes a stored value, increments it by one and returns an integer value. **Generate\_SubmissionID** is used when the object is initialized.

**Score**

This is an integer initialized to zero. It can be increased or decreased by users pressing buttons on the Answer in the UI to increase or decrease its score.

**Status**

This contains three states. Upon creation it is initialized to active. If it receives a report it is moved to in review. If it is cleared from the report it is returned to active if it is not it is moved to delete. A question **Status** may also be moved to deleted when it gets too old or spends to long inactive.

# 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.

## 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:

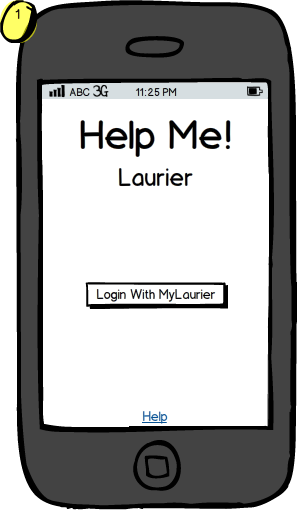
* **Home button**-A button that takes the user back to the **Home Page** (3)
* **Alert button** - A button that displays the current amount of notifications the user has and will redirect them to the **Alerts Page (4)** when clicked
* **Profile button** - A button that takes users to the **Profile Page**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

## Screen Images and Description

### Landing Page (1)



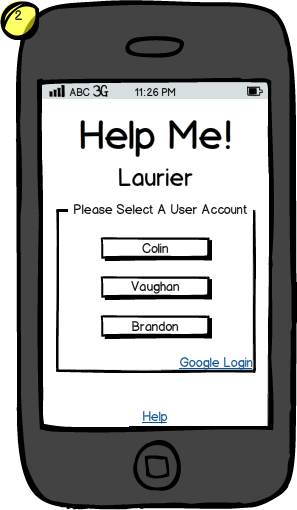
The **Landing Screen** is the first page that the user will encounter when opening Help Me! Laurier. On this page the user will touch the **LoginButton** and be sent to **the Login Page** (2).

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.

### Login Page (2)

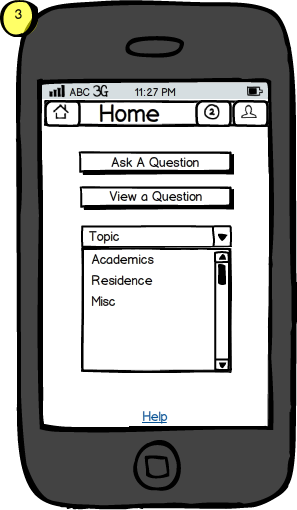


The **Login Page** will allow the user to enter their Laurier username and password if they are not currently logged into their smartphone. Otherwise, a popup will appear asking the user to select the account they wish to use if multiple accounts are logged into the Android smartphone. After selecting 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.

### Home Page (3)

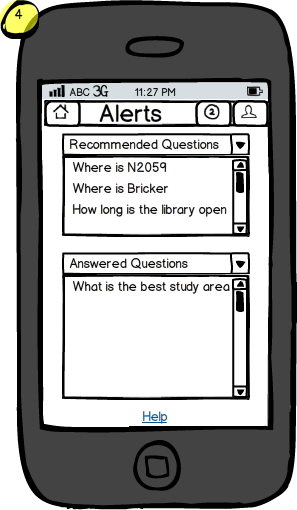


The **Home Page** contains a scrolling dropdown box which will allow the user to select a category of interest. Following this selecting the user can choose to “**Ask a question**” (9) or “**View questions**” (6) for that category. The user must select an item from the list first and then click the appropriate buttons. The buttons will 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.

### Alerts Page (4)

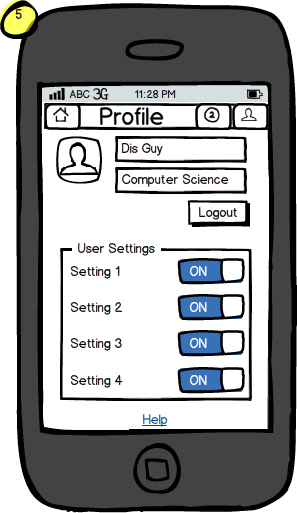
****

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.

### 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.

### View Questions (6)

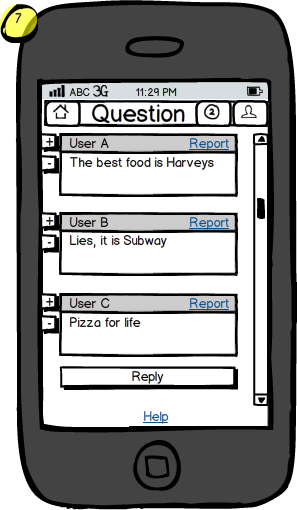


The **View Questions Page** will contain a list of questions pertaining to the users chosen category. When the user selects a topic they are redirected to the view **Question Page** (7).

**Special Notes:**

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

### 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)

### Report Question (8)

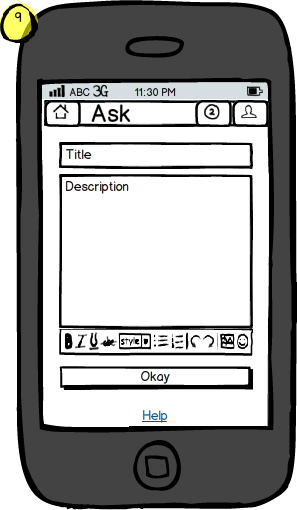


If the user see's that the content is unsuitable for a viewing audience the user can select the report button. Once the report button has been selected the user will be able to select a reason for reporting such as language or spam. This is a modal dialog, so flow exists back to the **View Question**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

### Ask Question (9)

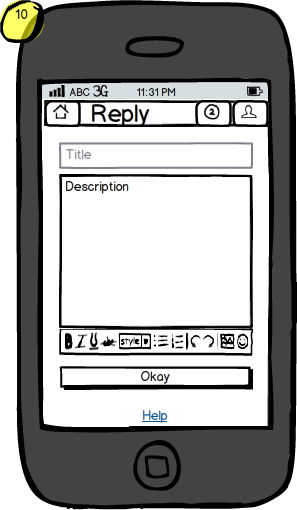


The **Ask Question** page will allow a user to submit a new question by filling in a title and a description of their issue. After the user leaves the description box, *Help Me! Laurier* will try to suggest questions that are similar to ensure repetitive questions are not being asked. If none of the 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.

### Reply Question (10)

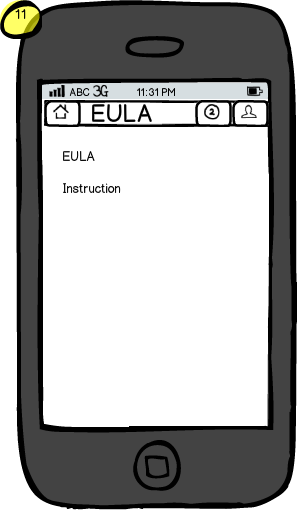


The **Reply Question** is a page that allows a user to reply to a given question by filling out the description textbox and selecting “**Okay**” which will submit the reply. The user will then be redirected back to the **View Question** page (7).

**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.

### 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.