XCAMPUS

Software Requirements and Analysis

Cross Campus

Version 1.0

2016-11-20

Revision History

Revision	Date	Author	Reviewed By	Summary of Changes
Draft	11/20/2016	Mark Gallant	Pavle Boraniev	Initial draft
Final	12/8/2016	Pavle Boraniev	Kazuma Sato	Final Copy

Document Approval List

Version	Approved By	Signature	Date
0.1	Pavle Boraniev	x	10/20/2016
1.0	Kazuma Sato	x	12/8/2016
		x	
		x	

Document Distribution List

Version	Name of the Receiver/Group	Date
1.0	Project Co-ordinator	12/8/2016

Table of Contents

1. Introduction

1.1. Purpose

The purpose of the document is to record our software requirements and designs for the Cross Campus Collaborative Learning Environment (CCCLE). It will provide the technical needs of the system for the purpose of designing its physical architecture.

1.2. Scope

The Cross Campus Collaborative Learning Environment is a new web application with an Android client linked through a REST-style backend API. The CCCLE will host student uploaded notes and educational content with an integrated commenting and rating feature. The CCCLE will also have a classified advertisement posting service for students to sell their used textbooks and other equipment. In the classified adverts, students will be able to arrange tutoring sessions to either tutor or receive tutoring. The application will include social media elements allowing users to save valuable time registering, login, and share content with their Twitter or Facebook accounts. The CCCLE will provide space on the website and Android GUI for APIs to fill with third-party advertising, as a means of generating revenue.

2. System Overview

The CCCLE is a service which will allow post-secondary students to engage with other students for the purpose of exchanging services and learning materials. Users will also be able to engage in discussion with one another over the available content through posting comments and ratings.

2.1. Project Perspective

This project will be mainly self-contained. All parts excluding our web hosting and parts of our front end will be developed in house to allow for the following reasons:

- Ease of support
- Increased efficiency
- Low overhead costs
- Complete oversight
- To demonstrate our programming and design skills

2.2. System Context

The specific parts which will form the core of the CCCLE will be a REST style API, a SQL modelled database, and Amazon web services hosting. For rolling out the release, this backend will be interfaced with a website and an Android application to form our front-end. This design will allow for easy feature management, as well as a consistent design across any platform. It will also allow for a simple maintenance cycle as all the resources can be managed independently from one another. Lastly, this design will allow the platform to be greatly scalable.

2.3. General Constraints

2.3.1. Business Constraints

As a collaborative student team, we face the sharpest of our constraints with group management, time and human resources:

- Available manpower
- Project focus
- Project deadlines

Our available manpower has considerably shrunk since the forming of our project vision and its scope. Having only three members available for project (from the original five) contributions has forced X Campus to reduce the scope of the CCCLE in terms of the feature richness of our system affecting both the front and back end portions of it. Additionally, no single member of our group has a dedicated focus on the CCCLE due to the scope of being in college. This forces us to constantly readjust our interactions and planning to configure around our other project needs. Finally, the last constraint is that X Campus does not create the final due dates for our own work. This causes stress on meeting our scope as the semester develops and is not always foreseeable.

2.3.2 Technical Constraints

Our technical constraints include:

- Limited institutional support
- Managing content and moderation
- Fluid design and visibility
- Rapid response times
- Intuitive and easy to use

Adding the ability to get information from an institution or validate through an institutional email will require a certain amount of time to implement for each institution that we choose to support; therefore the number of institutions we support will be limited by time the CCCLE is released. Our ability to handle out of scope content will also be a challenge to meet; we can constrain how much content will be allowed to be uploaded and track our users through their unique email but this certainly does not guarantee a failsafe process. The amount of moderation we will be able to utilize will be a constant strain on our business, product and end users. We will be aiming to use the breadth of our software to make the best of a fluid and attractive design, inviting users to utilize our platform. To maintain a fluid design and keep the product user friendly, rapid response times are crucial. Lastly, in our implementation of our features, we must ensure that our interface remains simple and intuitive for ease of use and avoid the feeling of overwhelming the user with bloated features.

2.4. **Assumptions and Dependencies**

We have made many assumptions from the start of this analysis. All implemented features and interfaces will be accessible by users upon completion. We will have enough server load to handle the release of our product. The cost of running the site will be countered by the revenue of having static advertising space.

3. Functional Requirements

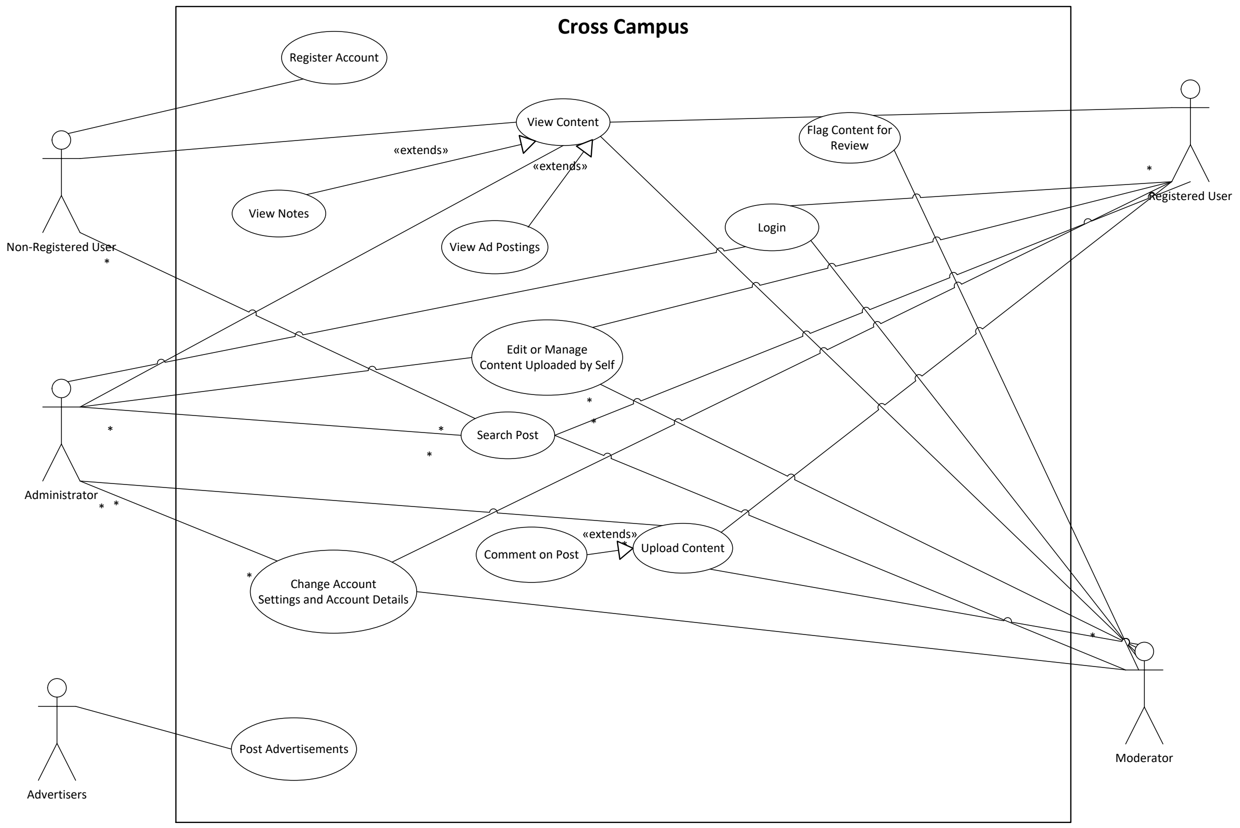
2.4.1. Requirements

- Accounts
  - authCheck
  - register
    - Normal
    - Social Media
  - login
    - Normal
      - Remember
    - Social Media
  - logout
  - update to cert
  - update account / password
  - request pass reset
    - reset pass
  - get favourites
  - get liked
  - disable
    - account
    - entry

- Services
  - get college list
  - get program list
  - get classes
  - get classified
  - get Entries
  - get Entry
  - get comments for post/classified

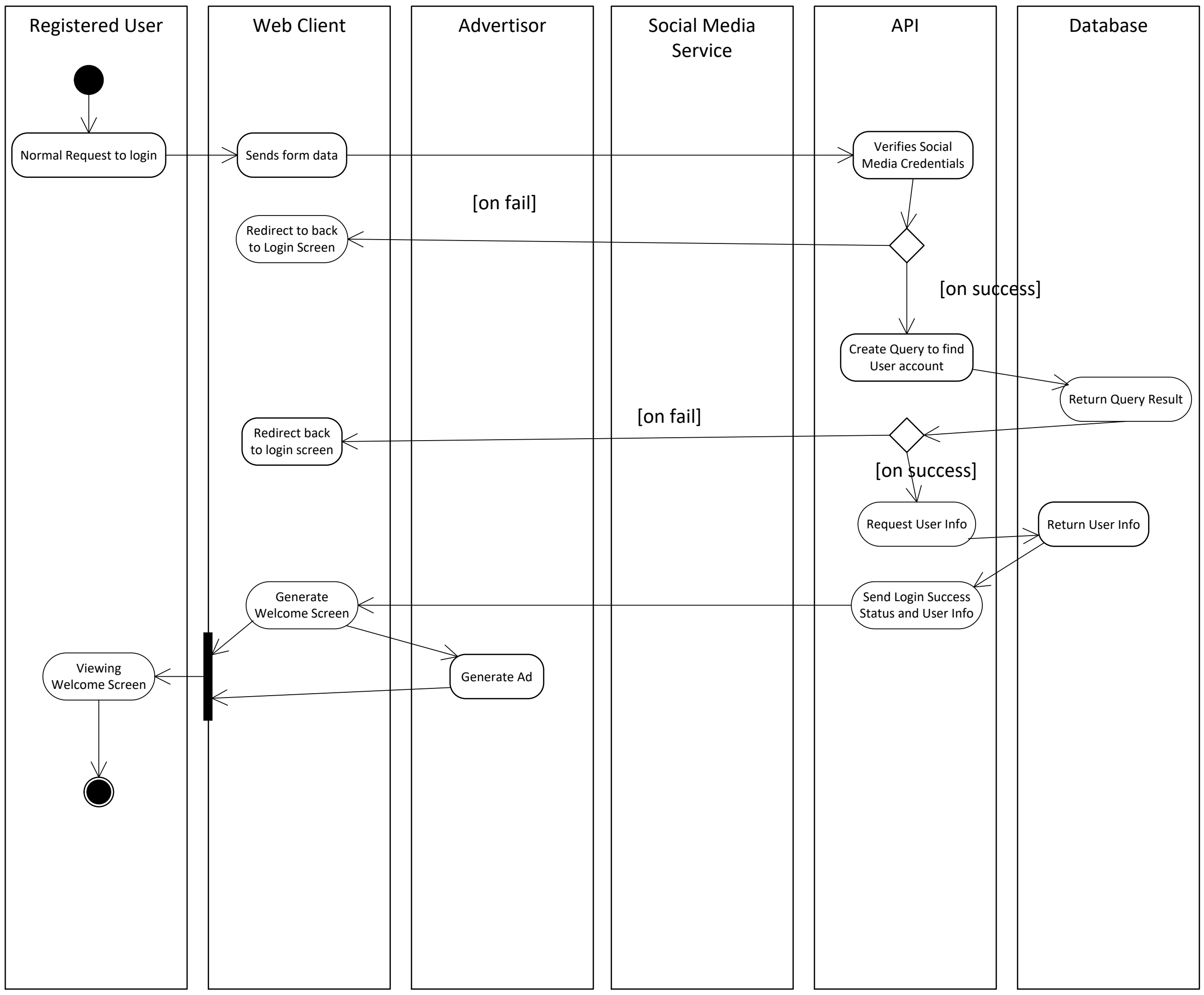
- Entry
  - post
  - add comment
  - add rating
  - Add File
  - flag
  - Fav
  - edit
  - delete

3.2 Use Cases

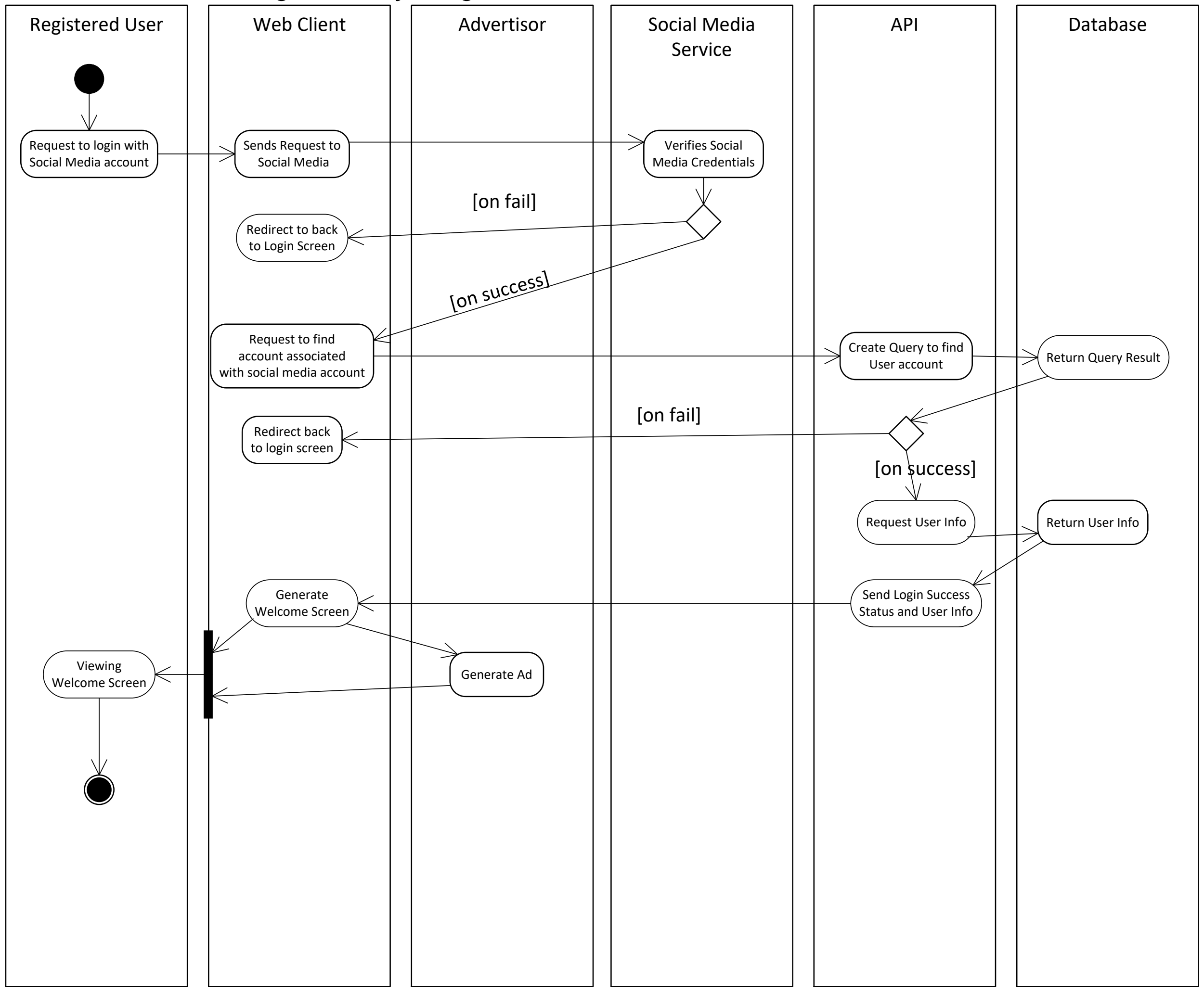

Cross Campus
Register Account
View Content
Flag Content for Review
«extends»
«extends»
View Notes
View Ad Postings
Login
Edit or Manage Content Uploaded by Self
Search Post
«extends»
Comment on Post
Upload Content
Change Account Settings and Account Details
Post Advertisements
Non-Registered User
Administrator
Advertisers
Registered User
Moderator
*

3.3 Data Modelling

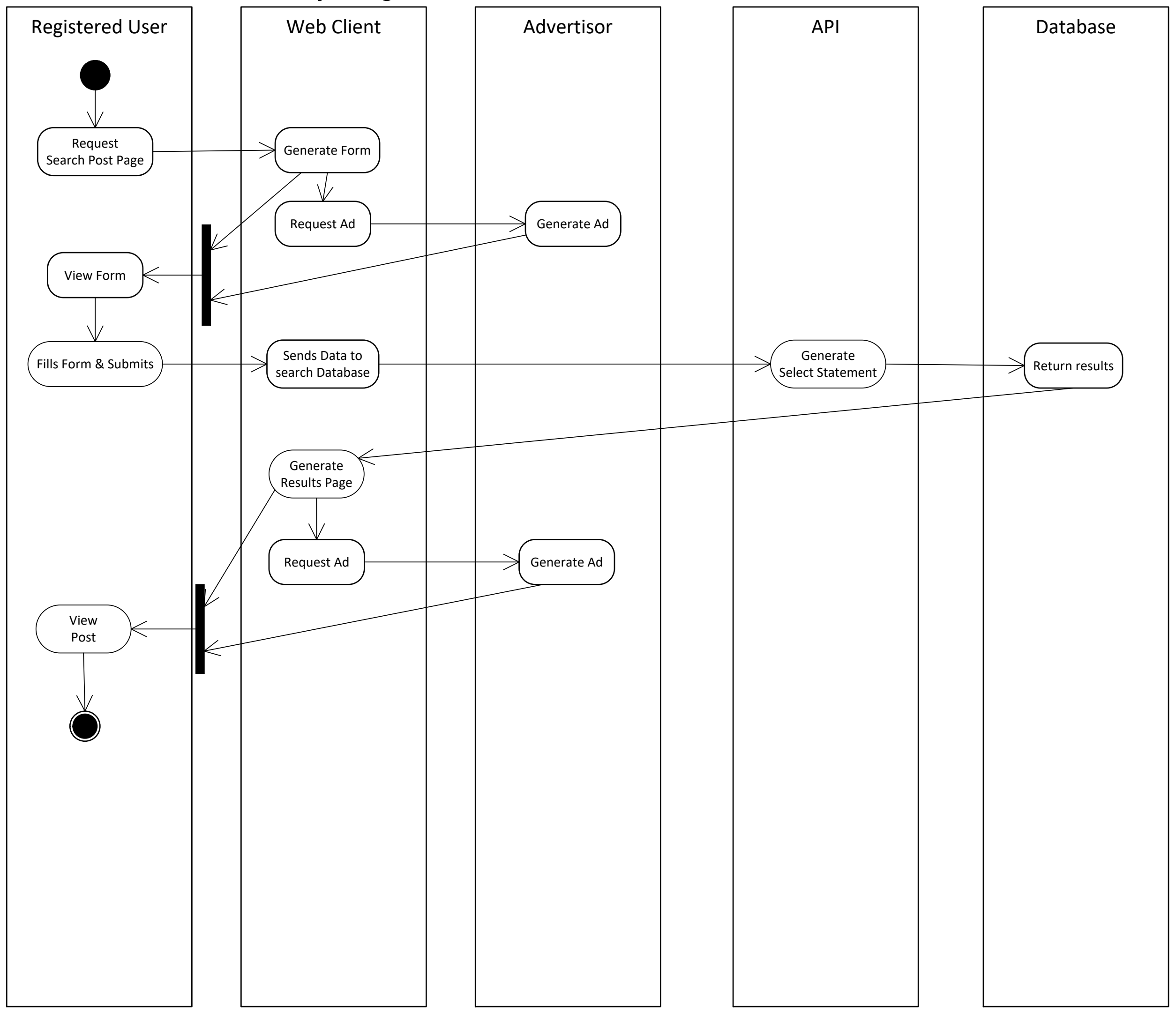
3.3.1 Regular Login Activity Diagram



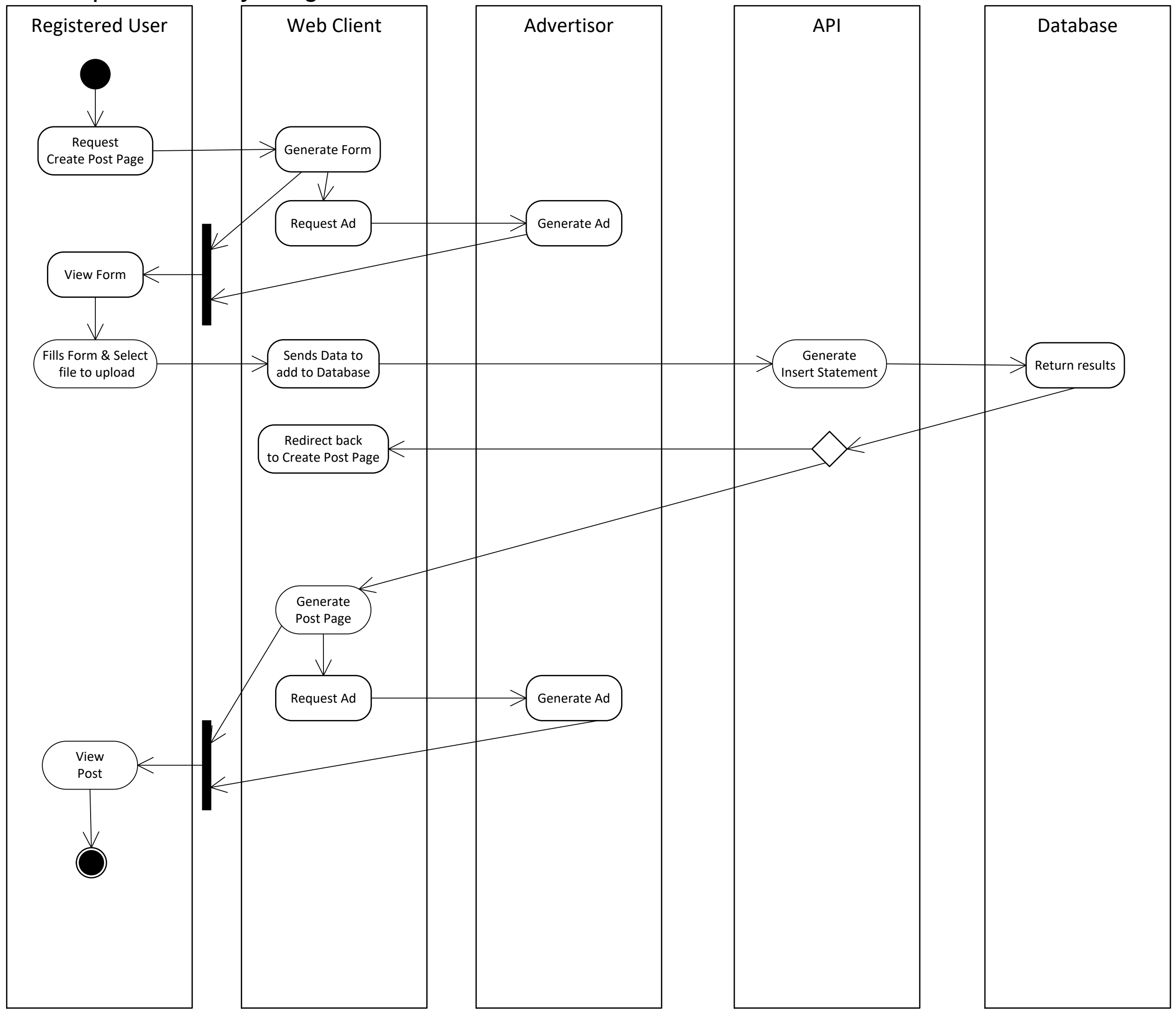
3.3.2 Social Media Login Activity Diagram


Registered User
Web Client
Advertisor
Social Media Service
API
Database
Request to login with Social Media account
Sends Request to Social Media
Verifies Social Media Credentials
[on fail]
Redirect to back to Login Screen
[on success]
Request to find account associated with social media account
Create Query to find User account
Return Query Result
[on fail]
Redirect back to login screen
[on success]
Request User Info
Return User Info
Send Login Success Status and User Info
Generate Welcome Screen
Generate Ad
Viewing Welcome Screen

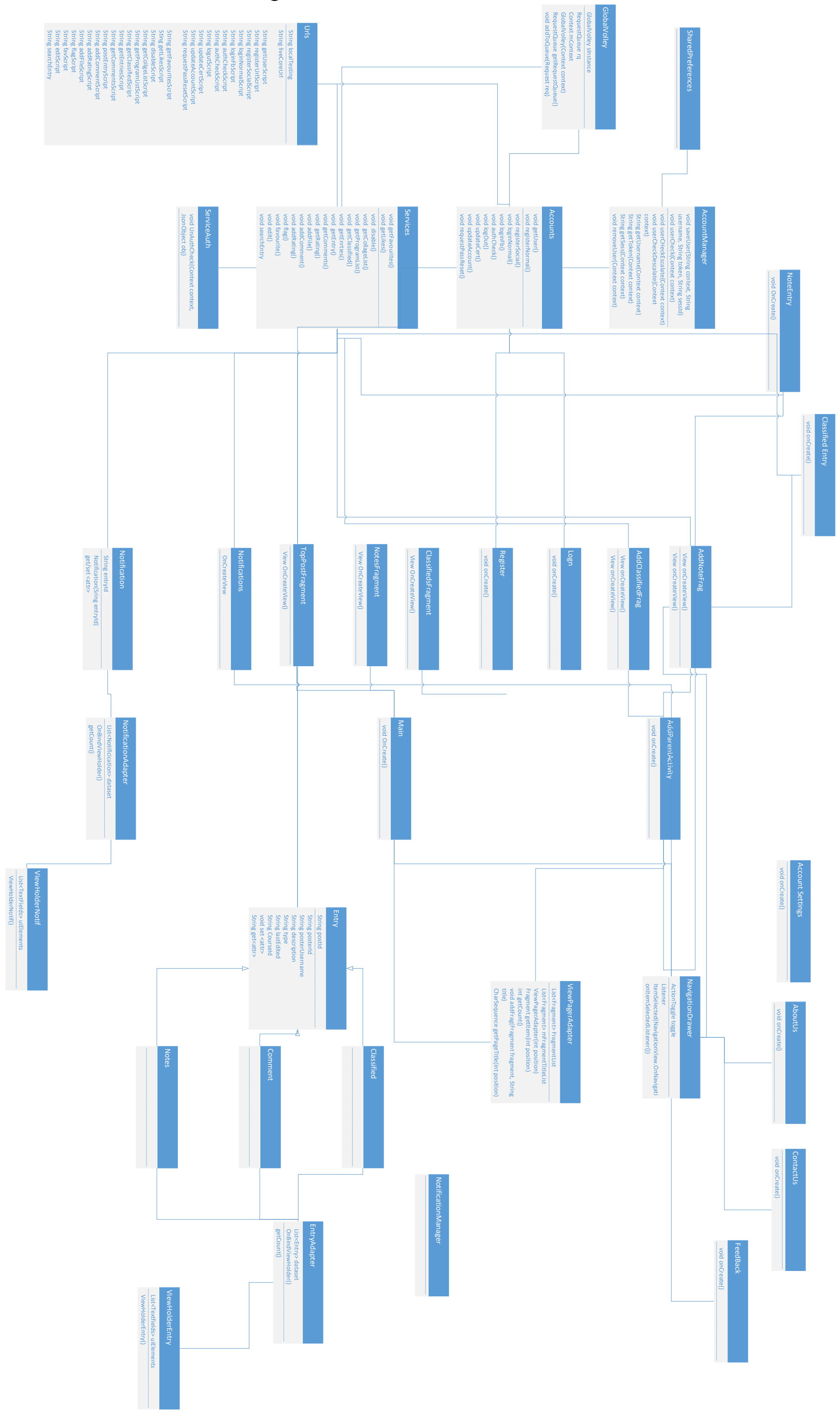
3.3.3 Search Posts Activity Diagram



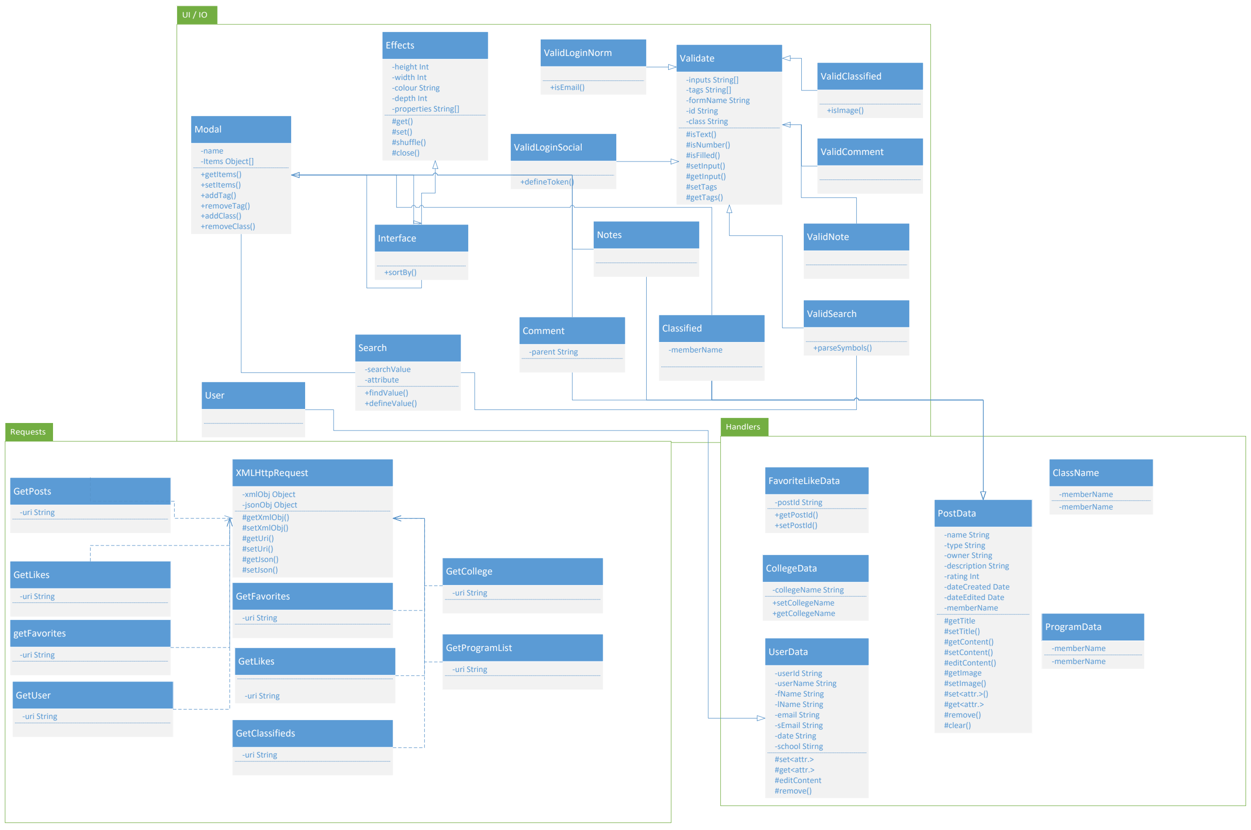
3.3.4 Upload Activity Diagram



3.3.4 Android Class Diagram



3.3.5 Android Class Diagram



API Script Pre Post and Result Conditions

getUser

Pre-Condition:	$user[]
Post-Condition:	$user [”<attribute>” = “<value>”]
Result:	Assigns User attributes to User

registerNormal

Pre-Condition:	$email = <value>, $username = <value>, $password = <value>
Post-Condition:	$validationKey = “<value>”
Result:	$email containing user validation key is sent to $user upon successful validation. Valid $email, $username and $password stored in database.

registerSocial

Pre-Condition:	$username, $email, $authToken
Post-Condition:	$_Session = [userName = “<value>”, id = “<authToken>”] Valid Username, Email
Result:	Social API provides authentication. User provides credentials. Validated credentials are stored into database.

loginNormal

Pre-Condition:	$username = "<value>", $password = "<value>"
Post-Condition:	$_Session[username = "<value>", $id = "<authToken>"]
Result:	$username and $password is verified with the database; User session is created or user is prompted to correct errors.

authCheck

Pre-Condition:	$authToken = <$_Session[$id]>, $authToken = <other>, $_Session[id = "<authToken>"], $Session[id = "<other>"]
Post-Condition:	
Result:	Comparison between $authToken and $_Session[$id], returning true only if they both contain matching values.

logout

Pre-Condition:	$_Session = [$username = "<value>", $id = "<value>"]
Post-Condition:	$_Session = [id = <null>]
Result:	Breaks user's session

updateCert

Pre-Condition:	$schoolEmail = “<null>”
Post-Condition:	$schoolEmail = “<value>”
Result:	-User has a validated institutional email associated with themselves.

updateAccount

Pre-Condition:	user = true, $_Session[$id = “<authToken>”]
Post-Condition:	user[attr] = “<value>”
Result:	-Validated user attributes are updated.

requestPassReset

Pre-Condition:	$userName = “<value>”, $password = “<value>”, $email = “<value>”
Post-Condition:	$password = <new value>
Result:	-User is sent an email containing the new value for $password

getFavorites

Pre-Condition:	$userId = “<value>”, $entryId = “<value>”
Post-Condition:	$favorite[$userId = “<value>”, $entryId = “<value>”]
Result:	User favorites a $post; the association is stored into Favorites Called into $favorite[]

getLikes

Pre-Condition:	$userId = <value>, $entryId = <value>
Post-Condition:	$liked[$userId = “<value>”, $entryId = “<value>”]
Result:	-User rates a $post; the association is stored into Rating. -Called into $liked[]

disable

Pre-Condition:	userId = <value>, userName = <value>
Post-Condition:	userName = <null>
Result:	-userName of User is set to <null>, disables account

getCollegeList

Pre-Condition:	$institutions[1][“id” = “0129382”]
Post-Condition:	$institution[][“id” = “<Institution name>”, “name” = “<Institution name>”, “description” = “<Institution Description>”]
Result:	-Returns an assoc. array of institutions

getProgramList

Pre-Condition:	$program[#][]
Post-Condition:	$program[#][“id” = “<Program Id>” , “name” = “<Program Name>”],
Result:	-Returns an assoc. array of college programs

getClassified

Pre-Condition:	$ads[]
Post-Condition:	$ads[“id” = “<ad token>”
Result:	-Returns array of ad keys

getEntries

Pre-Condition:	$entries[#][]

Post-Condition:	$entries[#][“id” = “<Entry Id>” , “parent” = “<Entry Parent Id>” , “name” = “<Entry Name>” , “type” = “<Entry type>” , “owner” = “<Entry Owner>” , “description” = “<Entry Description>” , “rating” = “<Entry Rating>” , “datePost” = “<Entry DatePosted>” , “dateLastEdit” = “<Entry DateLastEdited>” , “crn” = “<Entry CRN>”] = “<Entry ID>”
Result:	-Populates $entries[] with Entry data

getEntry

Pre-Condition:	$entries[][]
Post-Condition:	$entry = “<Entry ID>”
Result:	Assigns an Entry ID to $entry

getComments

Pre-Condition:	$entries[][]
Post-Condition:	$entryType = comment $entries[#][“<attribute>” = “<value>”]
Result:	-Assigns entry id where Entry is Type “comment”

postEntry

Pre-Condition:	$entries[#][“<attribute>” = “<value>”]

Post-Condition:	$query = “insert values $entries[#][...] into entry”
Result:	-$query is used to add a new Entry

addComment

Pre-Condition:	$entryId = “<value>”, $parentId = “<Id of comment parent>”, $type = “comment”
Post-Condition:	$query = “insert into Entry values $entry[#][“id” = <Entry Id>, “parentId = “<Id of comment’s parent>” … “type” = “comment””]
Result:	-Adds an Entry of type comment to the database with the id of the parent Entry as parentId

addRating

Pre-Condition:	$commended && $userId = false
Post-Condition:	$query = “insert $rating[#][] into entry where $rating[#][“commended” = true]”
Result:	-Updates the rating table where Commended is true under UserId

addFile

Pre-Condition:	$entries[#][“Id” = “<value>”], $files[#][“<attributes>” = “<value>”]

Post-Condition:	$fileEntry[#][entryId = “$entries[#][“Id” = “<value>”], fileId = $files[#][“Id” = “<value>”]”]
Result:	-Finds the id of the entry and files -Creates a record on the File table -Record is added to File_Entry Table

flag

Pre-Condition:	$entries[#][..., “type” != “flagged” ,...]
Post-Condition:	$entries[#][..., “type” = “flagged” ,...]
Result:	-Sets entry type as flagged

favorite

Pre-Condition:	$favorites[#][“UserId” = “<value || null>”, “EntryId” = “<null>”]
Post-Condition:	$query = “insert $favorites[#][] into Favorites where $favorites[#][“UserId” = “<value>”, “EntryId” = “<value>”]”
Result:	-UserId and EntryId are associated through the favorites table

edit

Pre-Condition:	$entries[#][..., "<Entry UserId>" = "<User UserId>" ,<description> = <value>,...],
Post-Condition:	$entries[#][...,<description> = <new value>,...]
Result:	-replaces the description attribute with a new/edited value if the Entry UserId matches the User UserId

searchEntry

Pre-Condition:	$entries[#]["<attribute>" = "<value>"], $value != "<entryId>" || "<valueId>"
Post-Condition:	
Result:	Attempts to find relevant Entry records based on <attribute> $value pairs

sortEntry

Pre-Condition:	$entries[#]["<attribute>" != "<null>"], $attribute != "<entryId>" || "<valueId>"
Post-Condition:	
Result:	Attempts to sort by relevant non null $attribute values

4. Non-Functional

Usablility

All schools and courses will be up to date. There will be backend scripts that will crawl all supported institutions recourses to keep an updated list of all schools, courses, classes. This insures will have no difficulties finding their class while looking for classifieds or notes.

Each user will be able to distinguish between useful files and non useful files. This will occur through a user feedback loop, our own admins looking through content, and a validation system in place. This will allow users to find what they are looking without wasted time looking for files.

Operation

Analytics are provided for each user. A detailed record of what each user does on the website from page visits to what they post. We'll use google analytics and query our database on a regular schedule to build reports of user activity. This allows us to have a base for future improvements and monitoring activity.

All files that run through our system will be moderated. All files served by our our system will be legal and relevant to whatever subject the file is purposed for. We will insure this with user input, constant database checks, and validation on input. This is to insure no illegal or irrelevant content is stored on our service.

Many people should be able to access any given script at the same time. There should be no slowdowns when there is a flood of traffic on an exam time file. We insure this by balancing loads, minimizing heavily sql queries, efficiently store files for retrieval. The goal being every script can handle thousands of users sending http requests to the same script, and receiving results in 10 seconds or less.

There should a load balancing system in place. Amazon Web Services will handle all load balancing for us. This insures always optimal run time.

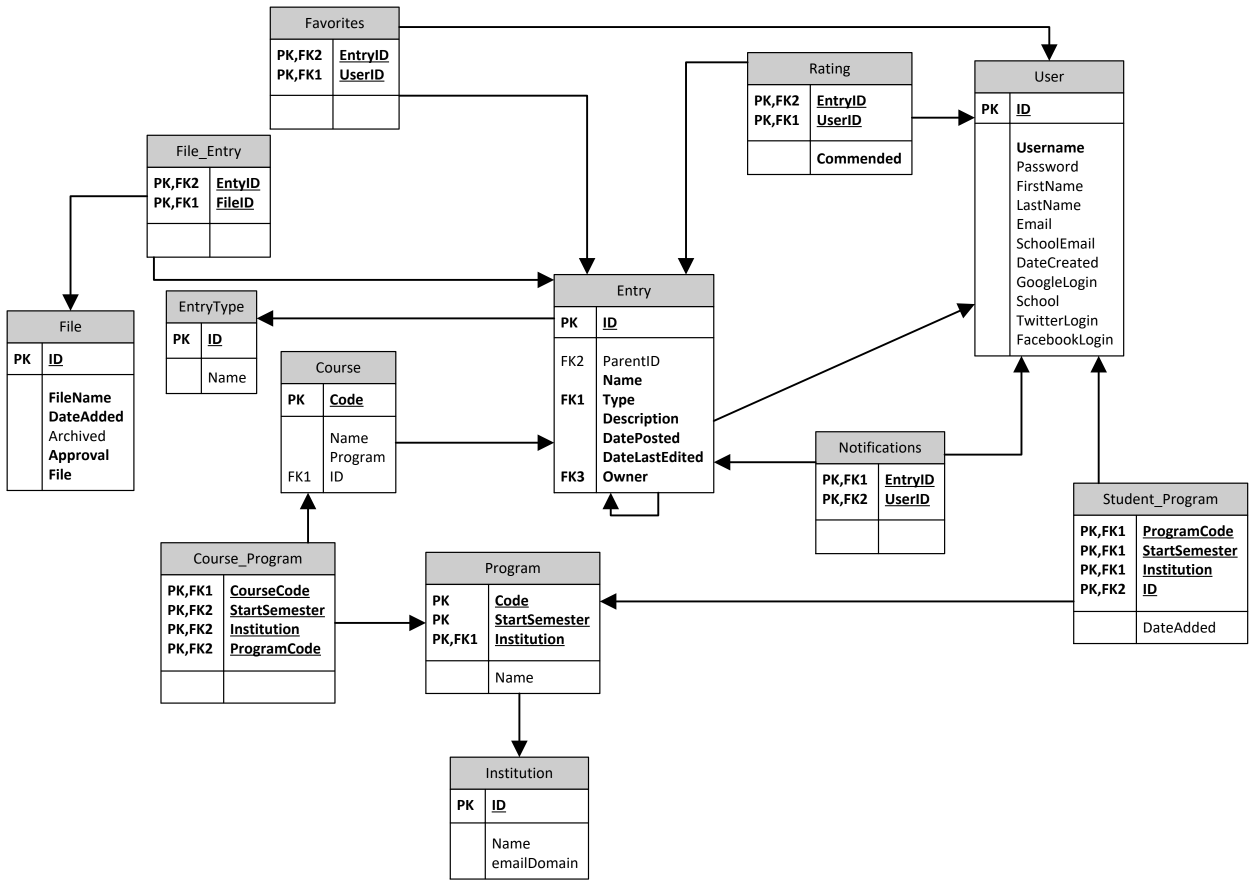
Maintenance

Testing environments for all backend scripts should be easy to create and get results from. This will be handled by our documentation for each backend script, having ready made client side files that call each script with whatever parameters needed and pulling a copy of a database. This insures testing will be done efficiently and accurately.

Backend should easy to maintain and modular. A minor change should not result in a string of changes that result in multiple scripts being edited and database schema changed. This insures updates and bug fixes will not be overhauls of the system.

Client side should not need to be updated when minor changes occur on the backend side. This is to insure that updates on the android platform only come around when there is a client side update and not a backend bug fix.

5. Logical Database Requirements



6. Approval

Project Role	Name	Signature	Date
Project Manager	Kazuma Sato		06/12/2016
Lead UX Designer	Mark Gallant		06/12/2016
Lead Developer	Pavle Boraniev		06/12/2016