Proximity

Project Report 2

Group 16: Ryan Wortmann (Leader), Song Vu Nguyen, Ryan Rottmann, Nathan Kulczak, John Oatey

Group Website: https://github.com/Rdubya54/Software_Engineering_Project

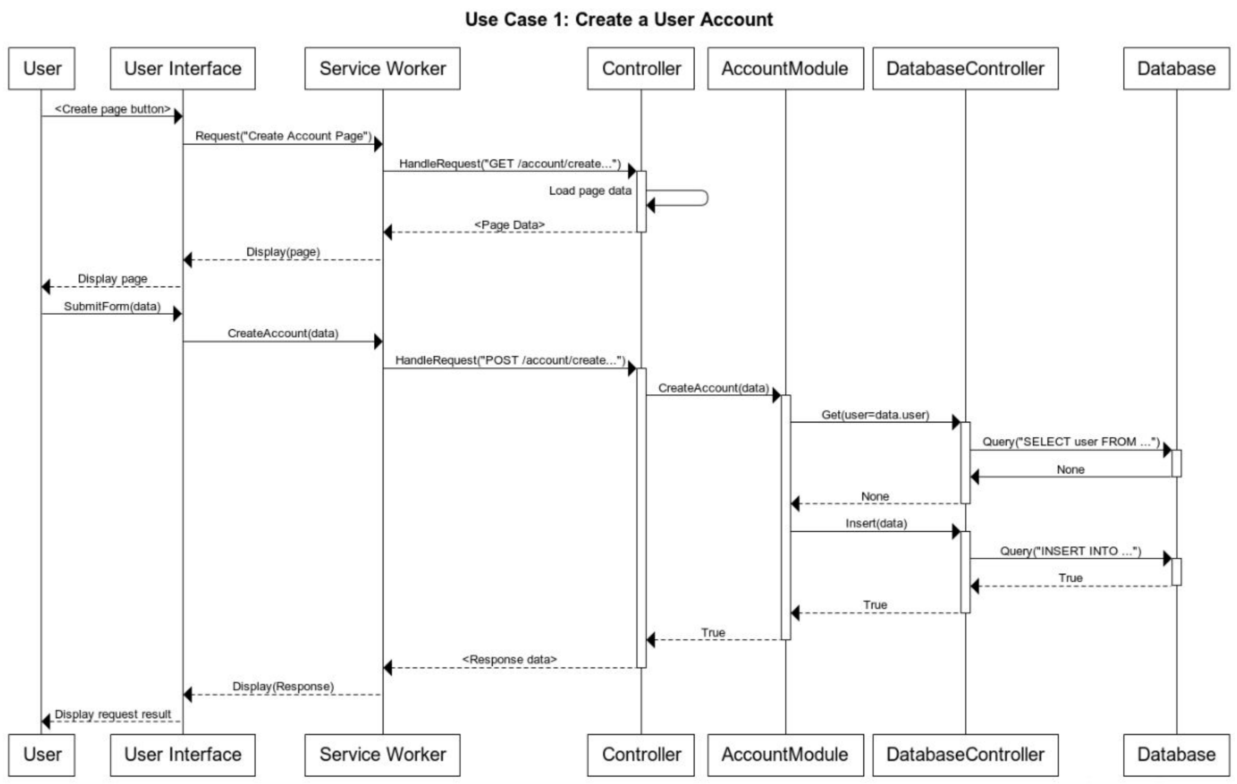
Individual Contributions Breakdown

******Each team member has contributed equally in the planning of this project and the drafting of report 2.

Table of Contents

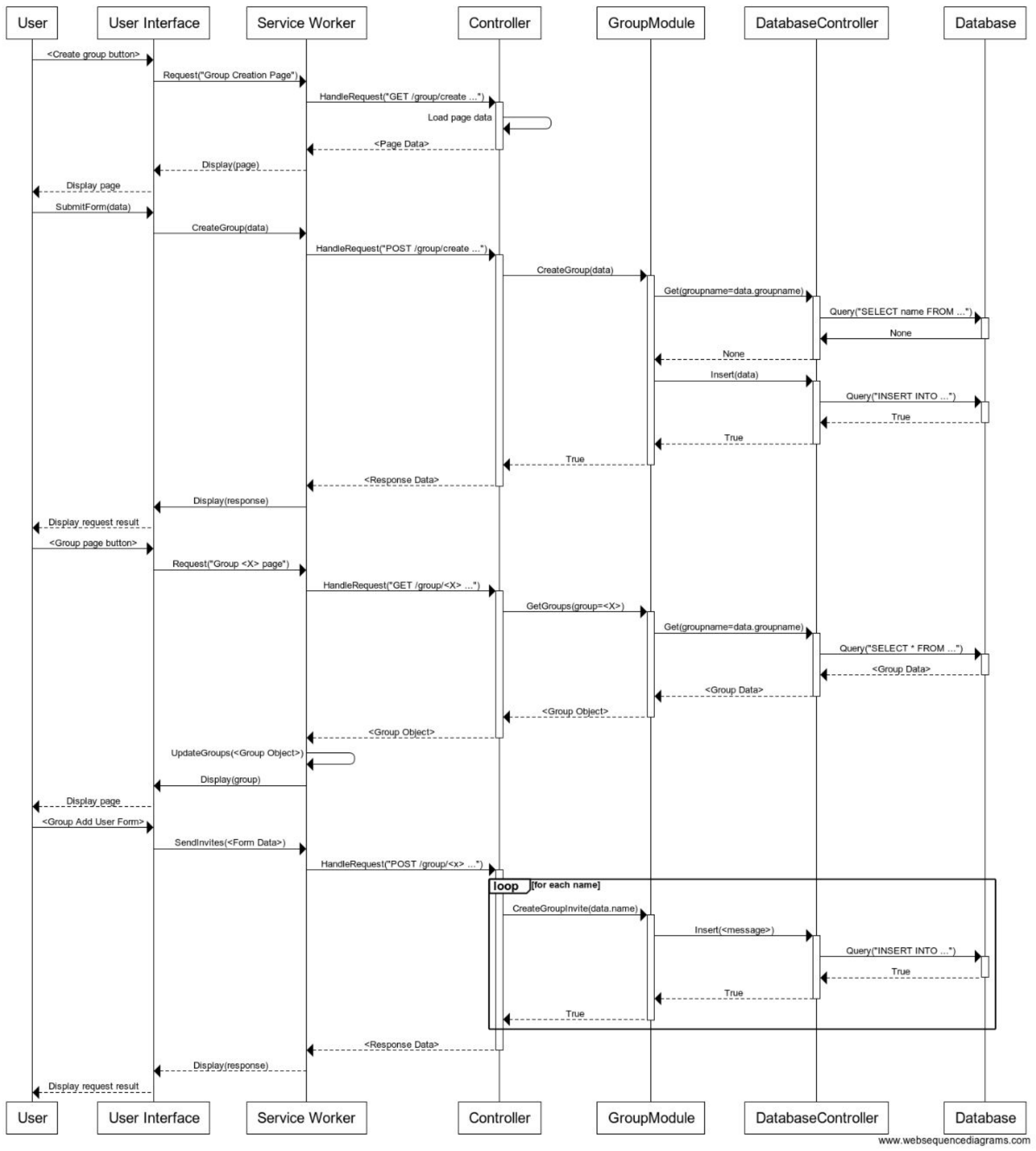
1) Interaction Diagrams

Our design tries to minimize the number RDD types that each object has. For instance, the AccountModule is Type 2 because it has many methods that request data from other objects and return said data to the object calling it. It is also Type 3 because it calls methods from DatabaseController in its methods to get requested data. However, AccountModule is completely stateless. It maintains no information about user accounts other than how to request and transform user account data.
The database is strictly Type 1. It knows and can return information but it communicates with no other objects unless it is called upon and it does no data transformations on the data it contains.



All modules are designed with a balance of the High Cohesion and Low Coupling Principles in mind. The AccountModule does nothing but request data from the DatabaseController and return it to the requesting object. Since it only communicates with the DatabaseController it has low coupling, and since it only handles data about user accounts, it has high cohesion.

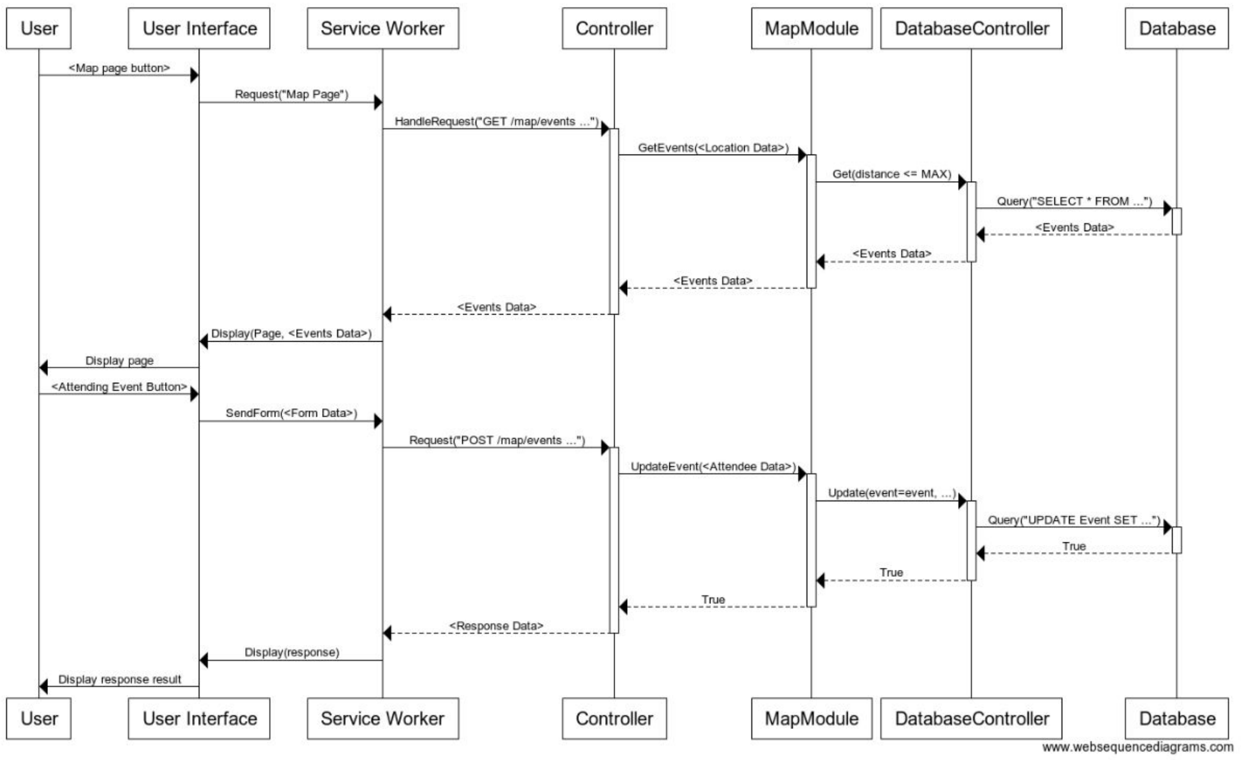
Use Case 4: Create and Manage a Group



The GroupModule is similar to the account module in that it operates only on GroupData.
In this we can see that the ServiceWorker also has additional responsibilities because it must maintain state information as well as communicate with the controller.
While this means that the ServiceWorker is a Type 1, 2 and 3 object we felt this was necessary to maintain asynchronous data flow between the user interface and the backend objects.
The controller also does not follow the overall design principles closely. It is highly coupled because it must communicate with all modules that get data from the database. However, the

controller has no Type 1 or Type 2 responsibilities because it exists only to pass messages from the service worker to back end modules.

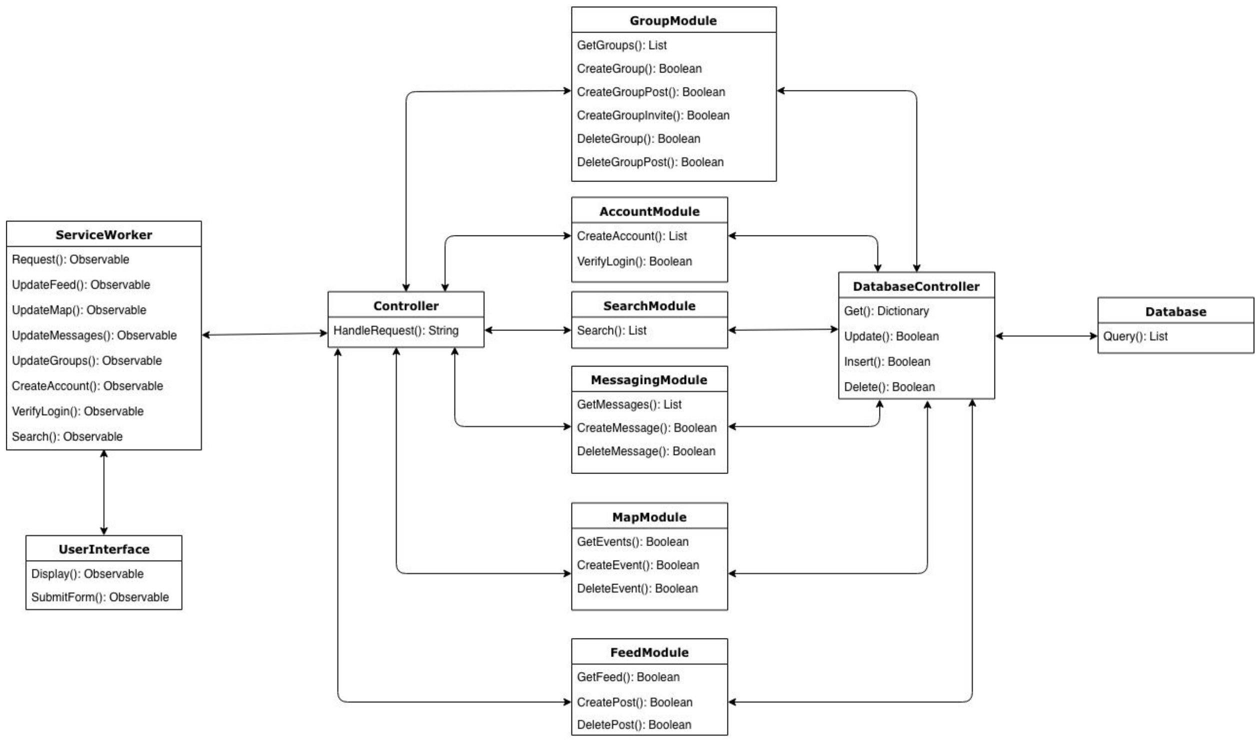
User Case 9: User can view map and interact with events other users have created



The MapModule is similar to other modules in that it has low coupling and high cohesion.
The DatabaseController is another object that has very low coupling and very high cohesion.
The DatabaseController communicates only with the database and does nothing other than request data from the database.
It exists to separate other modules in the backend from the database to reduce overall module coupling. This also aids in overall design because it means that only one object is interacting with the database directly. With all data from the database flowing through the DatabaseController it is easy to map the data flow throughout the backend making testing the objects much easier.

2) Class Diagram and Interface Specification

(a) Class Diagram



(b) Data Types and Operation Signatures

UserInterface

Display(): Boolean

- Boolean variable corresponds to if UI is displayed

SubmitForm(): Boolean

- Boolean variable corresponds to if the form submitted

ServiceWorker

Request(): Boolean

- Boolean variable corresponds to if the request was successful or not.

UpdateFeed(): Observable<String[]>

- Observable variable corresponds to a String array that will populate the feed as new data is received. The data is subscribed to

UpdateMessages(): Observable<String[]>

- Observable variable corresponds to a String array that will populate the messages for the user. The data is subscribed to

UpdateGroups(): Observable<String[]>

- Observable variable corresponds to a String array of Groups. The data is subscribed to to keep up to date.

CreateAccount(): Boolean

- Boolean variable corresponds to if the request to create an account was successful or not.

VerifyLogin(): Boolean

- Boolean variable corresponds to if the login was successful or not

Search(): Observable<String[]>

- Observable variable corresponds to a String array of Events and People nearby. The data is subscribed to

Controller

HandleRequest(): String

- String variable corresponds with the data that is returned from the modules

GroupModule

GetGroups(): List

- List variable corresponds with the list of groups that are returned from the search

CreateGroupPost(): Boolean

- Boolean variable corresponds to if the group post was successful or not

CreateGroupInvite(): Boolean

- Boolean variable corresponds to if the group invite was successful or not

DeleteGroup(): Boolean

- Boolean variable corresponds to if the group was able to deleted or not

DeleteGroupPost(): Boolean

- Boolean variable corresponds to if the group post was able to deleted or not

AccountModule

CreateAccount(): List

- List variable corresponds with the account information that is provided by the user

VerifyLogin(): Boolean

- Boolean variable corresponds to if the login was successful or not

SearchModule

Search(): List

- List variable corresponds with the list that is returned as a result of the search

MessagingModule

GetMessages(): List

- List variable corresponds to if the messages were properly retrieved

CreateMessage(): Boolean

- Boolean variable corresponds to if the message was able to deleted or not

DeleteMessage(): Boolean

- Boolean variable corresponds to if the message was able to deleted or not

MapModule

GetEvents(): Boolean

- Boolean variable corresponds to if the events were properly retrieved

CreateEvents(): Boolean

- Boolean variable corresponds to if the event was able to be created or not

DeleteEvent(): Boolean

- Boolean variable corresponds to if the event was able to deleted or not

FeedModule

GetFeed(): Boolean

- Boolean variable corresponds to if the feed was properly retrieved

CreatePost(): Boolean

- Boolean variable corresponds to if the post was able to be created or not

DeletePost(): Boolean

- Boolean variable corresponds to if the post was able to deleted or not

DatabaseController

Get(): Dictionary

- Dictionary variable corresponds with the data that is returned from the database to the controller

Update(): Boolean

- Boolean variable corresponds to if the database was able to delete or not

Insert(): Boolean

- Boolean variable corresponds to if the database was able to insert or not

Delete(): Boolean

- Boolean variable corresponds to if the database was able to delete or not

Database

Query(): List

- List variable corresponds with the returned information from the database

(c) Traceability Matrix

	Class					
Domain Concepts	Search Module	Messaging Module	FeedModul e	MapModule	Database Controller	Database
Controller					X	
View						

Data Storage						X
Communicator						
Data Manipulator	X	X	X	X		
Data Receiver						
Location Driven	X		X	X		

	Class				
Domain Concepts	User Interface	ServiceWorker	Controller	GroupModule	AccountModule
Controller			X		
View	X				
Data Storage					
Communicator		X			
Data Manipulator				X	X
Data Receiver	X				
Location Driven				X	

Due to the nature of the application, the domain concepts were fairly simple. The UserInterface receives data from the user and sends it to the backend through the service worker. There is then a module for each type of data we need. Each module (Group, Account, Feed, Map, and Messaging) knows how to manipulate its data that it receives from the database. The controller orchestrates the above. It is a very typical MVC (Model View Controller) design.

The main purpose of Proximity is to create a location driven social media application. The modules that implement this location driven idea are the Group, Feed, Map, and Search Modules.

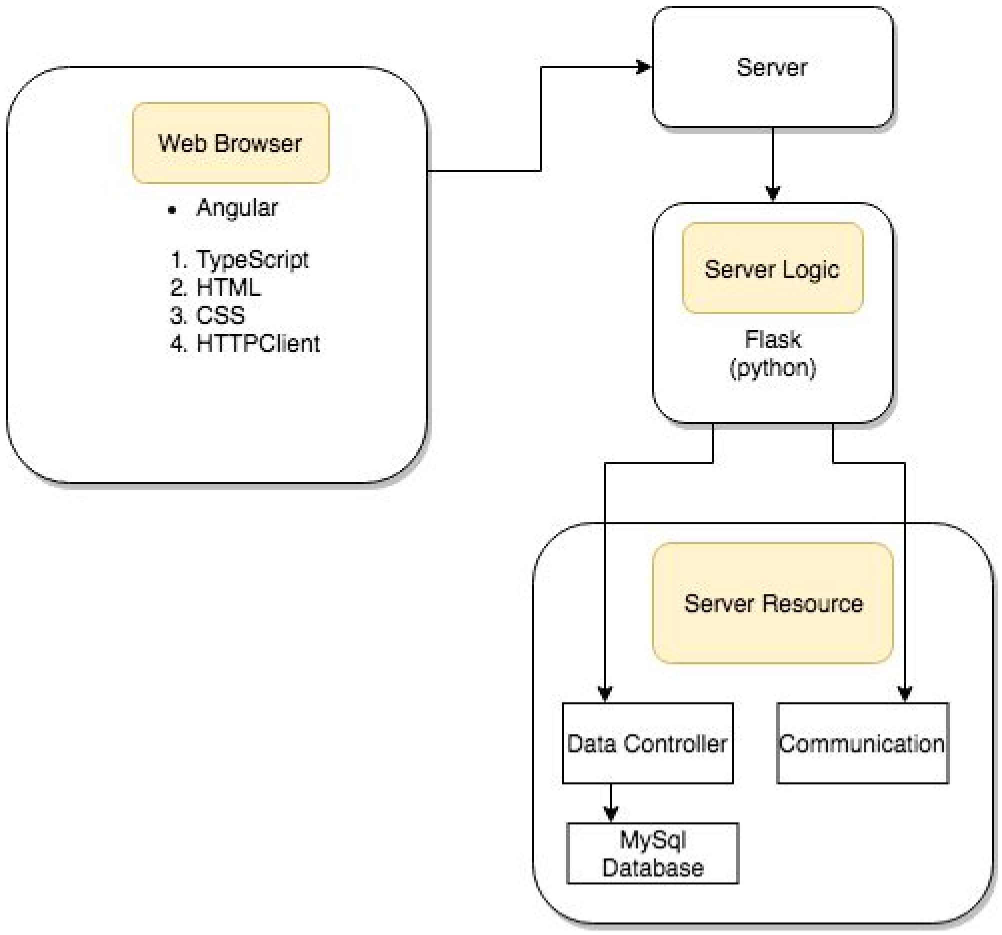
3) System Architecture and System Design

(a) Architectural Styles

Proximity uses a combination of a server-client architecture system and model-view-controller system. Using the Model View Controller architecture pattern, we have divided our application into three parts. The model handles storing data and CRUD operations and the view handles requests from the user. The model and the view are totally separate from each other, and do interact directly with each other in any way. The controller is used to handle the flow of requests and response between the two.

We also utilize a client-server architecture system because are application exists on a server and works by displaying information to clients. Our server exists on Amazon Web Services and the client is the web browser of the user.

(b) Identifying Subsystems



Subsystems Diagram

(c) Mapping Subsystems to Hardware

Proximity runs on a client-server architecture. The client exists on the user's computer or mobile device and the server exists on Amazon Web Services.

(d) Persistent Data Storage

Proximity uitizes MySQL, a relational database system, to store all the information needed about users, posts, events, and groups. Only the model part of the system is allowed to interact with this database. The database lives on Proximity's server.

(e) Network Protocol

Proximity uses the HTTP as the communication protocol. This was chosen due to HTTP being the standard client-server protocol. It also has the advantage of being stateless and platform independent.

(f) Global Control Flow

Execution Orderliness

The execution order is event-driven. The application provides a variety of events that the user can activate in an order they desire. For example, a user can create an event then message a friend, or the user can message a friend and then create an event. Aside from from first logging in, there’s is no required order for these events.

Time Dependency

Proximity is a real time system in the sense that it records the datetime of every post and every message. In addition events that are posted to the map expire after a certain amount of time.

(g) Hardware Requirements

- Screen Display
- An internet connection

4) Algorithms and Data Structures (If applicable)

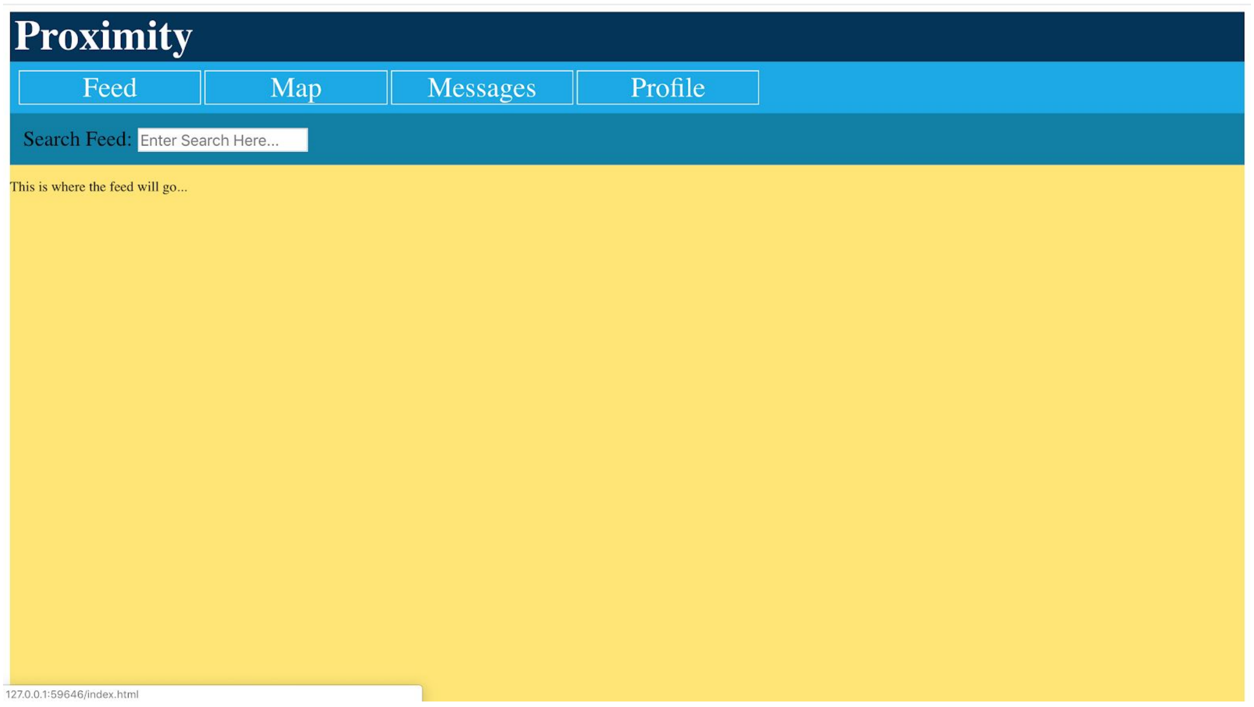
(a) Algorithms

The main algorithm in use is a distance finding algorithm that takes two sets of location data, in (latitude, longitude) format, and returns the distance between the two points in miles. This algorithm is used in database queries to find nearby posts, events, and people.

(b) Data Structures

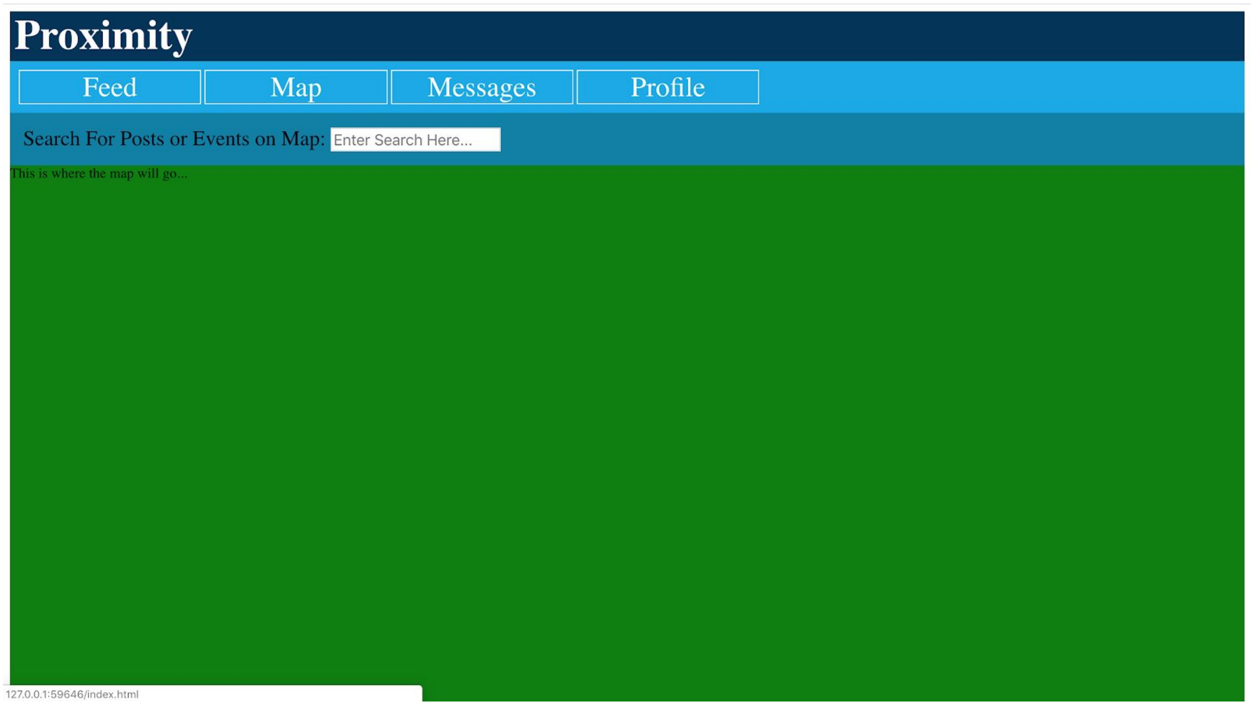
Our WebApp is going to pass data back and forth between the database and controller to update the user interface. These data will have the format of associative arrays, which is easy to implement. Other than that, we are not going to use any other complex data structures in our webapp.

5) User Interface Design and Implementation


Proximity

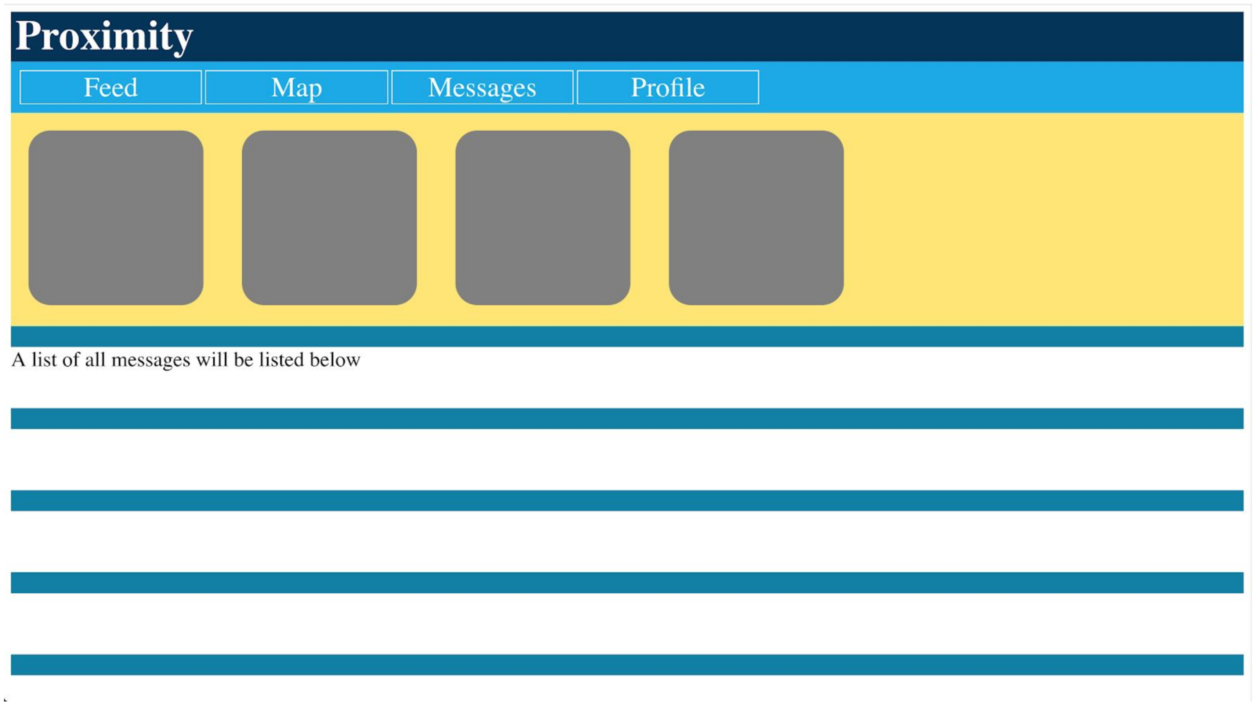
This is where the feed will go...

127.0.0.1:59646/index.html

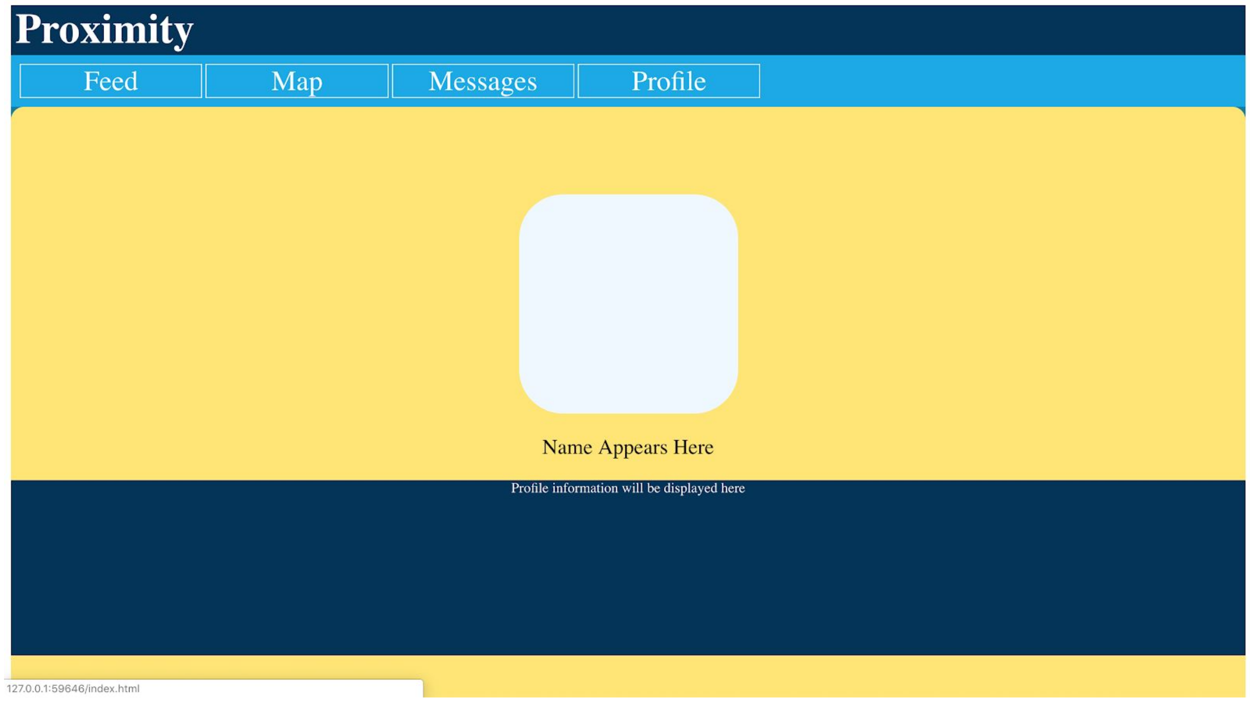

Proximity

This is where the map will go...

127.0.0.1:59646/index.html



A list of all messages will be listed below



Profile information will be displayed here

127.0.0.1:59646/index.html

The changes to our User interface include removing group messaging implementation as well as not having a full maps implementation. We plan on adding to the functionality to the map by allowing users to interact and message through events on the map.

6) Design of Tests

Unit test cases:

1) AccountModule tests:

   Account Register test:
   - User create a new account by entering valid information like name, birthdate, username, password -> expect account information to be added to database and system return message that account was created successfully.
   - Username chosen need to be unique. User enter username that already present in the database -> expect the WebApp to issue a prompt for user to choose different username.

   Account Login test:
   - User entered username and password that matched with the username and password that they registered in the database -> expect user to be able to login and access their account information.
   - The username and password that user entered is doesn't match any combination in the database -> expect the system to take user back to the login screen and said that you have a wrong username and/or password. That way it more secure.

2) MapModule tests:

   Map viewing test:
   - User access to the map by clicking on the map tab on the WebApp interface -> expect the map to be load correctly on the page and accurately display the detail of the created test event.
   - The event on the map will only be visible to people locally so only the people who nearby in the area within 10 miles radius will be able to view the event. User loaded map page 20 mile away from the test event -> expect the system to not show the event when user load up the map page.
   - User create a test event and posted on the map -> expect the map to display the event at the location that the event were created and not somewhere else.

4) GroupModule tests:

   Group Creation test:
   - User finishes filling out the the group page using valid information and hit the button “Create New Group” -> expect group data to be added to the database and system passes back message that the group is successfully created
   - User finishes filling out the the group page using same name of some existing test group in the database and hit the button “Create New Group” -> expect group data to be redirect back to the user with the system message of “The group you try to create already exist. Please choose a different group name”.

5) <u>MessagingModule tests:</u>
Message sending test:

- User finishes typing their message to the chat box and hit send -> expect the message data to be send to the the test account that specify in the header and not to other test accounts.
- User finish typing their message to the group message chat box and hit send -> expect the message data to be send to the of test accounts that specify in the header

6) <u>FeedModule tests:</u>
Post on a feed test:

- User finishes writing a post and hit post -> expect post data to be successfully added to the database. If the post is public, expect to be able to view the post from any test account. If the post is a friend only post, expect to be able to access and view the post only from account that on the test account friend list.

* During the testing process, if any of the test case return unexpected result, it will be considered failure and need to be debug. All test cases that listed above is just a design and maybe change when we actually implement it.

<u>Test Coverage</u>

Each test case coverage is the class that it written under (for example, Map viewing test cover MapModule class). Also, for each test case, although not mentioned explicitly, necessary classes like UserInterface, ServiceWorker, Controller, DatabaseController and Database were also involved.

<u>Integration Testing</u>

Integration Testing will be done as a group by creating several test accounts and data. Everyone will then take turn to serve as the User and go through all the unit test cases for each class to see if all the components will work well together.

7) Project Management and Plan of Work

(a) Merging the Contributions from Individual Team Members

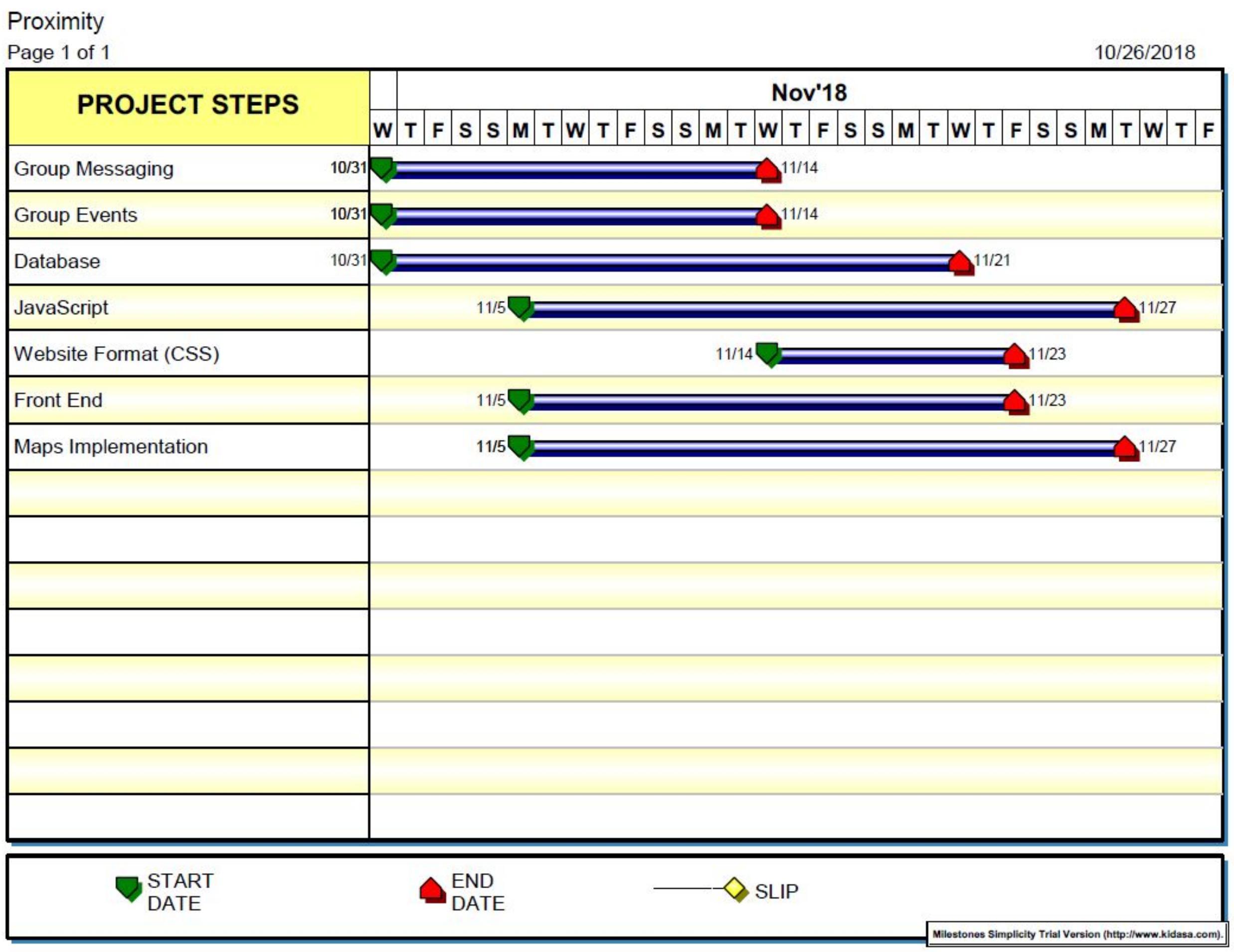
For this report, we divided the works evenly between team members. A single document that included a cover page, table of contents, headers for each require sections were created and shared to all team members on Google Drive. This ensure a consistent format throughout the report. Each team members were assigned specific parts of the report to work on but since some parts are associate with each other, sometimes we need to wait for one part to finish before we can get working on the next part. This wasted lot of time but we were able to speed up the process by maintaining constant communication through our phone and were able to work together and finished the report on time.

(b) Project Coordination and Project Report

As the time of this report, we almost got use case 1 (user register and login) and use case 9 (map interaction) done. We already got the skeleton of the WebApp finished. When we are done with user login and map interaction functions of the WebApp, user will be able to create and login using their account as well as create and view events on the maps. Depend how this is going, we are thinking that we may drop use case 6 (which is group management) before Demo 1 and implement it later. Currently we are all working on the WebApp to get the basic functions down before Demo 1. John is working to finish user register and login function. Nate is working on the code for the back end to connect with the database. Ryan Wortmann is working on the map implementation. Ryan Rottmann is working on the format (CSS) and user interface of the WebApp. Song Vu is working on the code to handle request from front and back end of the WebApp and will assist John and Ryan Rottmann in finishing the front end of the WebApp

(c) Plan of work

These are the things that we plan to do for the rest of the semester. It might change as we progress with the project



Plan of Work Chart

(d) Breakdown of Responsibilities

Below are each team member name and parts of the project they are currently working on:

Ryan Wortmann- currently working on MapModule that include all map functionality

Song Vu Nguyen- currently working on Controller to handle request from user and database and assisting John and Ryan Rottmann on the front end of the WebApp

Ryan Rottmann- currently working on UserInterface that include layout and CSS for the Webapp

Nathan Kulczak- currently working on DatabaseController and Database that will query requested information from the user

John Oatey- currently working on AccountModule that let user register and login to our WebApp using their account information

*Unit testing will be done by member who responsible for writing that unit. Integration testing will be done together as a group before we finish the WebApp for Demo 1.

*All of these responsibilities are not absolute as group members are likely to change role and will assist each other in all parts throughout the implementation of our WebApp

8) References

1. This Project Report 2 is made follow Lecture Slides and example Project Samples (included HeartRateAdjuster.pdf, HeathMonitoring.pdf, RestaurantAutomation.pdf) that posted on Canvas

2. This Project Report 2 document made using tools from Google Drive
https://drive.google.com/

3. "Software Engineering" book by Ivan Marsic
http://www.ece.rutgers.edu/~marsic/books/SE/book-SE_marsic.pdf

4. Interaction Diagrams made using tool from:
https://www.websequencediagrams.com/

6. Using Amazon Web Services
https://aws.amazon.com/

7. Class Diagrams and Subsystems Diagram made using tool from:
https://www.draw.io/

8. Milestones Simplicity 2017 for Plan of Work chart
https://kidasa.com/

9. Complementary Color Generator
https://coolors.co/c1edcc-b0c0bc-a7a7a9-797270-453f3c