# Software Design Document

**Team:** Red Alert
**Sponsor:** State Farm Insurance
**Faculty Team Mentor:** Han Peng

**Team Members:**
Sal Galan (Lead)
Calvin Harper
Myles Dailey
Nick Nannen

**2/7/22**

## 1.0 Introduction

The insurance industry in the United States is enormous. In 2020, companies providing life insurance made a combined revenue of over 800 billion dollars in the United States alone. This is not very surprising given most homeowners, renters, and car owners have a desire to secure their belongings in case of a major accident. Companies like State Farm, Berkshire Hathaway, Progressive, and Allstate generated more than 30 billion dollars each in total insurance premiums paid by customers in 2020 with many other companies closely trailing behind. Property, home, and auto insurance companies are huge and growing due to the fact that citizens care about guaranteeing that their property will be replaced or fixed in the event of a disaster.

State Farm is the largest property, casualty, and auto insurance provider in the United States. Like most insurance companies, State Farm Insurance generates most of their revenue by selling insurance policies to customers. When a customer buys an insurance policy, that customer has to pay a monthly premium in order to keep their insurance policy. If a customer incurs some type of damage such as a car accident, a home catching on fire, or a family member passing away that customer will receive some benefit from State Farm in accordance with the customers insurance policy. In order to generate a stable revenue stream, State Farm and most other insurance companies must intake more premiums from customers than payouts for insurance policies. In order to sell policies to customers, State Farm employs agents which are independent contractors who are responsible for acquiring new clients and selling them insurance policies. Agents are solely paid on commission which is earned by onboarding new customers and selling them insurance policies. This is why the State Farm agent to customer relationship is so important to State Farm as a company and their agents.

Our clients are Glenn Austin who is a Technology Analyst and Hans Yeazel who is a Technology Manager at State Farm. Glenn Austin is in charge of managing employee infrastructure. This means Glenn is in charge of projects that aim to improve the educational development of employees at State Farm. For example, Glenn’s team works on projects that provide additional educational education to employees such as developers, data analysts, and even consultants.

Similar to Glenn, Hans Yeazel works with State Farm product and development teams to ensure that each team is on the path to a quality solution. Hans also works with product planning teams to provide direction and leadership when discussing a solution.

In total, State Farm employs over 19,000 insurance agents and handles over 84 million insurance policies. State Farm also employs over 55,000 thousand internal employees that are not State Farm agents. Among the thousands of non-agent employees, State Farm employs lawyers, software developers, data analysts, paralegals and many other types of people with a broad range of skill sets.

As mentioned above, the agent-customer relationship is paramount to State Farms financial success and reputation as being a customer centric company. As a result our team has been tasked with creating an agent-client notification web app that will allow agents to send notifications to their clients easily using a visual map interface. Our envisioned product will enable agents to tailor communications with their clients based on their clients physical location. Currently, agents have no easy way to visualize where their clients are located. Our solution will allow agents to send warning and reminder notifications to their clients in case of natural disaster or problematic weather. Furthermore, agents can also use our product to send clients notifications to remind their clients to renew their policies or to simply wish their clients happy holidays.

The specific features of our web application are implementation of agent user accounts, a dashboard view, and usage of a MongoDB database in our website's backend.

Agents will have their own account that maintains a record of the agent’s personal information such as their name, address, State Farm firm, and agent code. Agents will also be able to save subsets of their client list to send group notifications. Similarly, agents can save search queries they make for clients to easily find a group of clients they have searched for previously. Most importantly, agents will be able to visually select one or more clients on a map to select clients to send notifications too. Lastly, agents will be able to create recurring notifications that get sent to a selected group of clients at a recurring interval or at a specific date and time.

The functional requirements for agent user accounts are as follows:

- Agents will have a personal profile page that allows agents to edit their personal information.
- An agent's profile page will also contain a list of saved search queries, client subsets, and notification automations.

Our site will also contain a single page that will enable agents to use almost every feature available by our web app without needing to navigate to separate pages. This dashboard view will contain a search bar, map, a list of saved search queries, automated notifications, and client subsets. Everything an agent needs to be productive will be located on one simple and efficient web page.

The functional requirements for the dashboard view are as follows:

- The map on the dashboard page will allow agents to draw an outline with their mouse around clients they would like to select to send notifications too.

- Creating notification automations, client subsets, and saved search queries will also be available from the dashboard view.

Environmental constraints required for this project include the use of a MongoDB database as this is the database that State Farm uses to store information for their own services.

## 2.0 Implementation Overview

Our clients have emphasized the importance of our web application implementing Web2.0 standards for our product solution. To achieve a modern, responsive, and secure website we are utilizing the Django backend framework, a MongoDB database, a javascript frontend, and a NGINX and Gunicorn web server to build our web application.

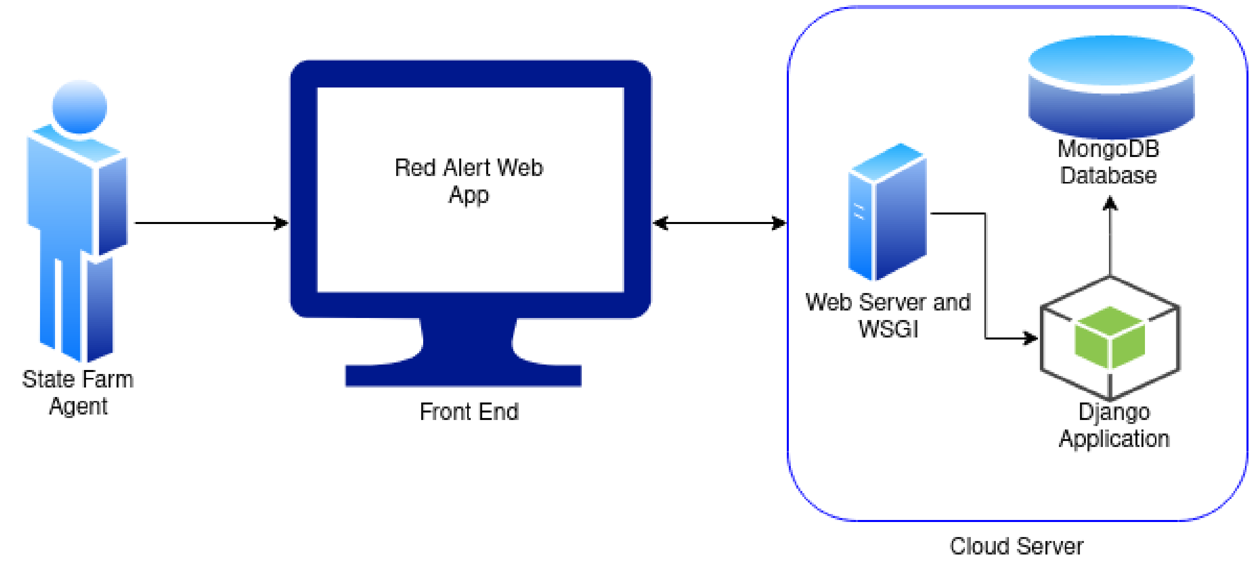


*Figure 1.0 - System Implementation Overview*

Our solution will be a Web2.0 Django application hosted on a cloud server which allows for our application to scale to fit possible user growth needs. The Django backend framework is modern and has a strong developer support community. Django utilizes a Model View Controller design pattern as well as a highly modularized code base which enables our product to be easily maintained. The Model View Controller design pattern allows us to separate business logic (making database queries or preparing data for user display) from our site presentation which prevents complications incurred by mixing presentation and logical operations in the same file.

The front end of our server consists of javascript files that run in the user's browser. Our front end will be responsible for displaying a map to the user with their clients addresses displayed on the map. *Figure 1.0* shows that the front-end of our site communicates with our backend to send and receive data in order to display client data properly. Our Django backend will provide the data for our front end application.

While the MVC design pattern supports code organization, Django takes this a step further by utilizing app based code organization. For example, all the code and files responsible for handling user logins and account creation is stored in a completely separate folder than the code and files responsible for showing the user dashboard page. To summarize, Django is the backbone of our website and is responsible for determining html, css, javascript, and python files to run or serve to the user. The Django backend is also responsible for sending text messages and emails as well as allowing users to search through our database for clients. Django also provides core functionality such as user authentication and password hashing. Django is the sole communicator with the database, as it is the only program with access to the database per *Figure 1.0*.

Our MongoDB database stores every piece of information required for our site to work properly. This means user account and client information is stored in our servers database. User passwords are also stored in our database. When users change their profile information the Django application communicates with our database to make the users change in the actual database. MongoDB is a NoSQL database which allows for flexible data storage that does not enforce complex schema requirements compared to a traditional relational database. This makes it easy for us to change data format on the fly if necessary without having to deal with conforming to complex shema requirements. NoSQL is comparable to a stack of paper that contains a list of terms with definitions where there are no requirements for how a definition is defined.

Furthermore, our web server software consists of two programs, a NGINX server and a WSGI application. The NGINX server is used to serve static files quickly, allowing for responsive web pages and complete control over how our website handles client traffic. The WSGI application is like a translator placed between the NGINX server and our Django web application. The WSGI application is similar to a translator as it enables NGINX to communicate with our Django application since NGINX does not natively support the Python programming language used in our Django application.

# 3.0 Architectural Overview

Continuing, our project will consist of our Front-end, our Web server, and also our Django backend. The database is located within our backend but we have shown it separately for easier visualization. This can be seen in Figure 2.0. A majority of our project will be constructed in our backend as this is where many of the modules need to be created. In our backend, the modules consist of UserModule, SMS and Email Module, and the Dashboard module Within our frontend, we will have our GIS Map Display. This allows the user to view and interact with the map module. Lastly, we also have our web server component. This will contain the WSGI software along with our web server. These sections will be the core components for our project.

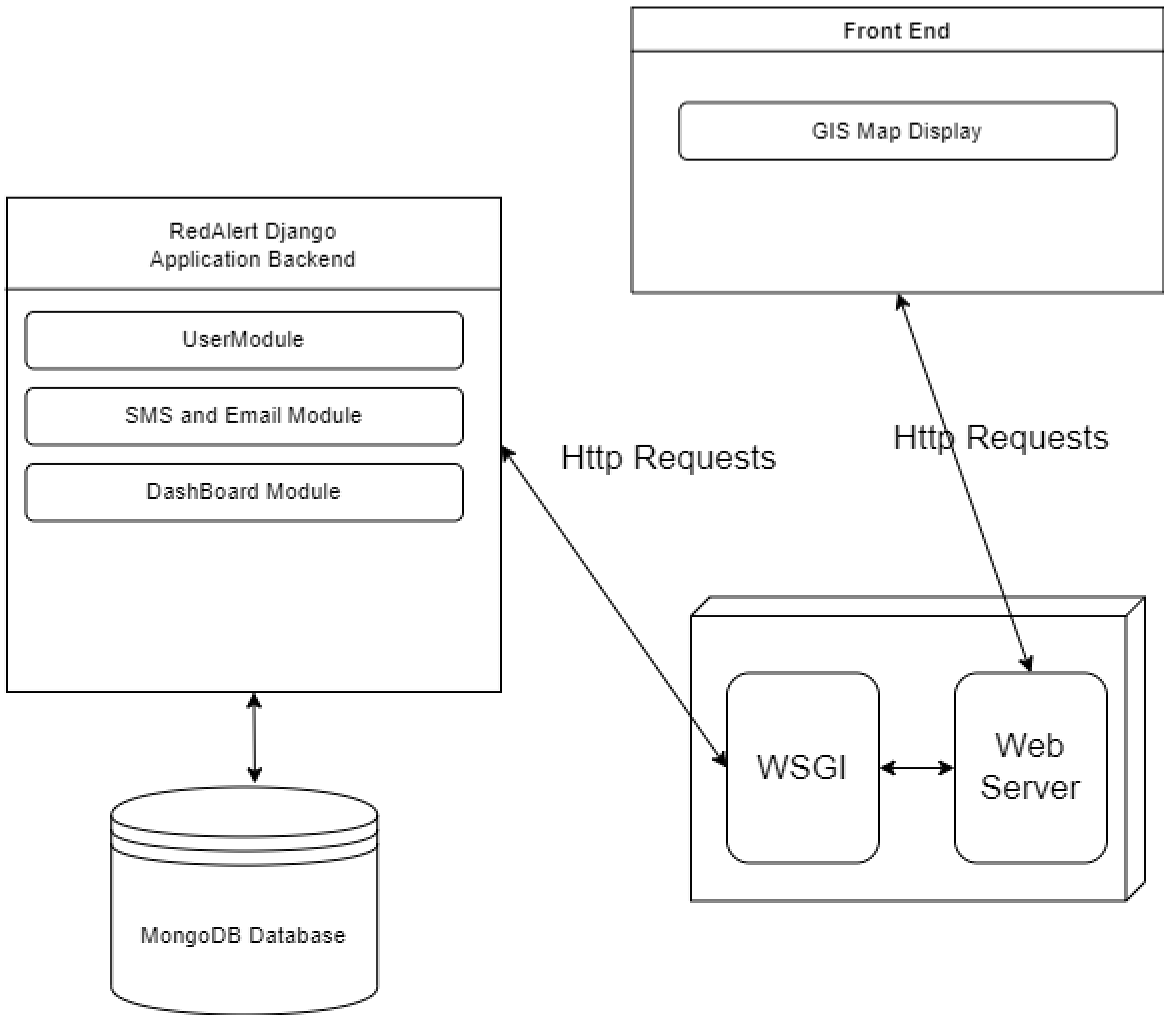


*Figure 2.0 - Architectural Overview Diagram*

Our system architecture is composed of 4 main components. These include, the backend application, the user facing front end, the database, and the web server.

The web app backend is responsible for handling the business logic of our application. This means our Django backend handles database queries, executing user requests to change application information, and preparing data to be shown to users. In other words the backend application utilizes custom algorithms and functions defined by the development team to assist in data exchange between end users and the web application.

The user facing front end is the portion of our application responsible for preparing and presenting web pages to the user and executing some site functionality. Most notably, the front end of our web app is responsible for displaying an interactive map to the user and allowing that user to utilize the map to select clients to communicate with. The backend provides information from our database that is required for the map to function properly. General map functionality such as showing user and client location is executed in the users browser rather than on the backend.

The database is responsible for keeping track of user profile data, client information data, and other pieces of data required for the site to run properly. The database is responsible for keeping track of data that is not static and is subject to change.

Our web server is the interface responsible for receiving user requests for specific web pages which are then delegated to the backend application to respond to. In *Figure 2.0*, its important to note that our web server is composed of two applications, a NGINX server and a Gunicorn WSGI application. The NGINX server is responsible for executing user requests for web pages and sending those web pages to the users browser. The Gunicorn WSGI application is a translator for the NGINX server and the Django backend as NGINX does not support the python programming language. When a request is received by NGINX, NGINX passes the request to the WSGI application which then translates the request into a form that the Django application can understand and respond to. Responses from the backend are then passed to the WSGI which are passed to the NGINX server and finally passed to the user.

The server and database portion of our architecture require relatively little configuration compared to our systems backend and frontend which are the most important parts of our web app. The backend application is broken down into a UserModule, SMS and Email module, and a Dashboard module. The UserModule is responsible for updating and creating user accounts. This module is also responsible for fetching and preparing user data to be displayed. The Dashboard module is responsible for all logic related to our website's dashboard view. This means the Dashboard module handles all logic related to gathering information that enables site features such as a client search interface, saved search queries, saved client subsets, and saved notification automations to be displayed to the user. This module only handles logic responsible for executing search queries, and displaying data related to user accounts, but does not handle logic for executing client notifications although this functionality is visible in the Dashboard module. Because Django allows for a web application to be divided into sub applications, our program will import functionality from the SMS and Email notification module rather than trying to mix SMS and Email and Dashboard responsibilities into a single module. The SMS and Email module will allow users to create notifications and will handle the execution and scheduling of notifications. This module will also handle database operations required to save user made notifications.

## 4.0 Module and Interface Descriptions

As described by our architectural overview, our core design consists of different components referred to as modules . Drawing from *Figure 2.0*, the most important parts of our web application are the backend modules and the frontend module. Our Django backend application is divided into the User module, the SMS and email module, and the Dashboard module. These modules support the core functionality of our entire web application. The frontend of our application consists of a single module called the GIS Map Display Module. This module provides the map that State Farm agents will be using to interact with their client list.

## 4.1 The User Module

The user module is responsible for user account creation, user login, and the user profile page. Any type of interaction that involves the display of user account data or changing user data is handled by this module alone.

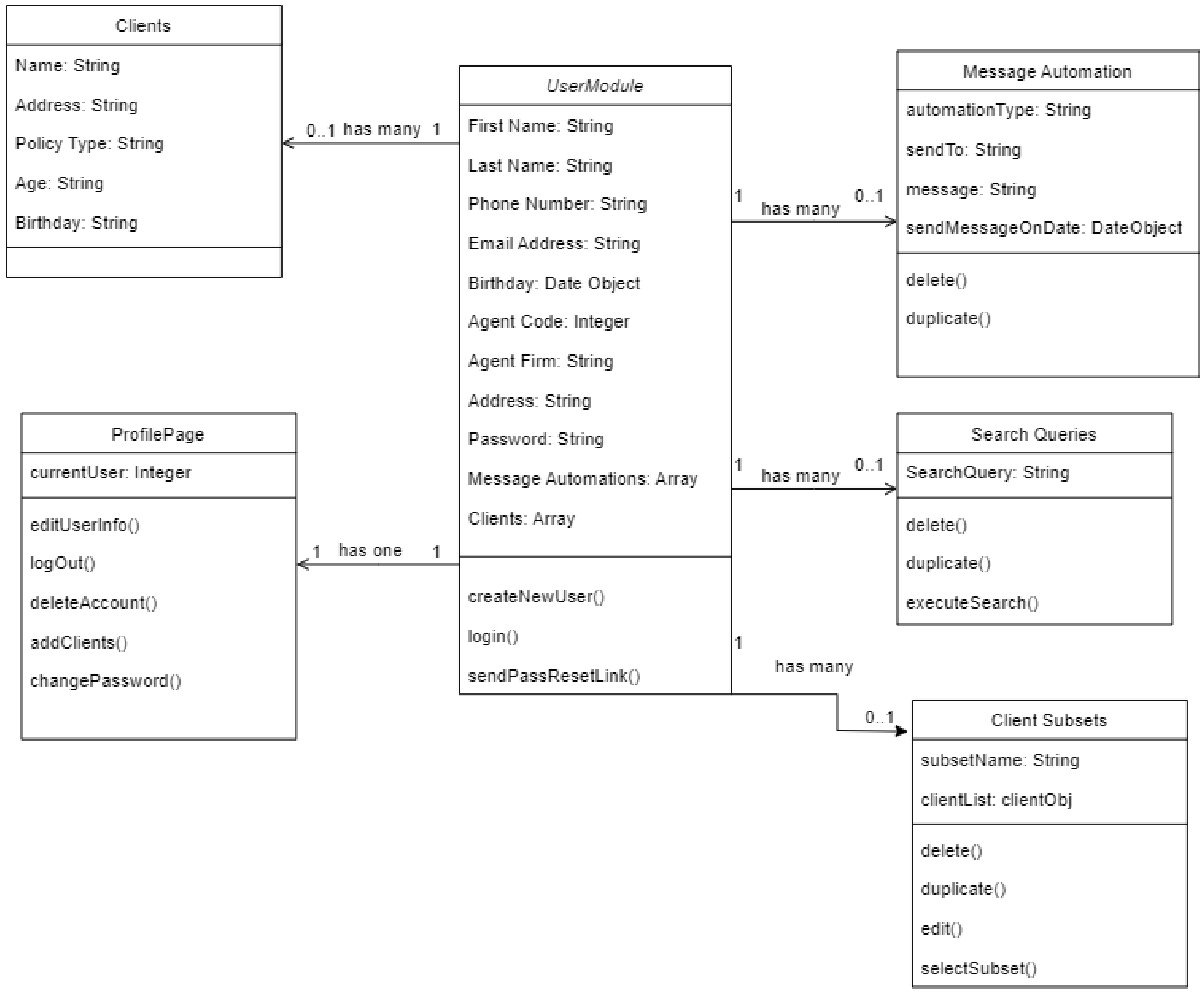


*Figure 3.0 - User Module UML Diagram*

### 4.1.1 The User Class

The user class is the primary representation of a user account and its related information. A user is defined as being a State Farm agent. This class is responsible for handling user account operations including creating new user accounts, displaying user information in the form of a profile page, and handles authenticating users to allow them to log in. This class also stores user information such as name, address, and birthdate. If a user needs a password reset link, this module handles sending users a link to update their password.

Fields of the User class:

- First and last name: The user's first and last name used to identify the user.
- Phone number: The user's phone number. This field is optional as a phone number is not required for any of the web applications functionality.
- Email Address: The user's email address. This is the field used to identify a user account and is used to allow users to log into their accounts.
- Birthdate: A date object representing the user's birthday. This field is optional.
- Agent Code: A number that represents an agent's identity in the agent's firm's databases. This is used to identify an agent in systems other than our own but is necessary to enable users to access correct information from client databases located in State Farm firms systems.
- Agent Firm: The name of the firm that the agent works for.
- Address: The user's location. This address is obtained explicitly from the user to ensure that the user's location is properly displayed on the map.
- Password: The user password which is used in conjunction with the users email address to log into the web application.

Functions of the User class:

- createNewUser(): this function is run when the user requests to create a new account. The accompanying new account webpage is then displayed to the user.
- login(): This function handles displaying the login page to the user and authentication of user credentials.
- sendPassResetLink(): If the user forgets their password, the user requests a password reset link which displays a web page allowing the user to reset their password.

### 4.1.2 The Profile Page

The profile page class is another component of the user module. This class is responsible for fetching and displaying user account information. This class also allows users to edit their account information if changes are required. This module allows users to reset their password from their account page rather than needing to request a password reset link from the account login page. This is also the class responsible for allowing agents to manage their list of clients. Agents can create new clients to add to their list of clients. Ideally, in a production environment, an agent's lists of clients would be fetched by a remote server and agents would not be able to manually edit and manage their clients without going through a secondary service such as one provided by State Farm. Because this project is a proof of concept, we are using locally stored client information created by the development team to emulate real agent clients so we must allow agents to edit and manage their own list of clients for testing and demonstration purposes.

Profile Page Fields:

- User Object: An object representing a user's information which is stored in our database. The profile page class fetches the user object and then extracts information about the users to display it.

Profile Page Functions:

- editUserInfo(): This function makes most user fields editable on the user profile page besides the users password and agent code. In a production environment the agent code would be verified by a third party system before the agent would be able to finish creating their account and therefore can not be changed.
- logOut(): Logs the user out of their account and returns the user to the user login page.
- deleteAccount(): Allows the user to delete their account entirely, wiping the users data from the database.
- editClients(): Enables the user to add, edit, or delete clients from their list of clients in the database.
- changePassword(): Allows the user to request a password reset link from their account page rather than needing to navigate to the user login page.

### 4.1.3 The Client

The client class is used to store the attributes of clients stored in our database. The client class contains no functions of its own since interactions with the user class are handled by other classes such as User and Profile Page classes.

Client Fields:

- Name: The name of the client
- Address: The address of the client which enables us to pin the clients location on the map.
- Policy Type: The type of policy the client owns from State Farm. Could be policies such as fire, auto, or home.
- Age: The clients age.
- Birthday: The clients birthday:

### 4.1.4 Search Queries

Search queries are strings saved by the user to execute at a later time. If an agent uses the search bar to search for clients and finds the results particularly useful, the agent can save the search query to make the same search again without needing to retype the query. Search queries are associated with the user class.

Search Query Fields:
- Search Query: A string representing the search terms to re-execute.

Search Query Functions:
- Delete(): This function deletes a search query from the users list of saved queries.
- Duplicate(): Creates a new query identical to the query being duplicated.
- executeSearch(): Executes the search query in the search bar.
- editQuery(): Allows the user to edit the string representing the search query.

### 4.1.5 Client Subsets

Client subsets are groups of clients created by users to allow users to easily perform actions on these groups. Users could create a group of clients who all share the same birthday which would enable the user to select the subset of clients instead of each client individually to send them a notification.

Client Subset Fields:
- subsetName: Name of the client subset.
- clientList: An array of client ids that the user has selected to be a part of the subset.

Client Subset Functions:
- delete(): Deletes the client subset from the database.
- duplicate(): Creates a duplicate client subset.
- edit(): Allows the user to change which clients are included in the subset.
- selectSubset(): Selects every single client in the subset and adds the subset of clients to the select client pool which allows the agents to send notifications to the subset.

Notifications automations are also represented in the UML diagram but are not directly part of the User Module. Rather, this class is presented to demonstrate the relationship to the User Module. The notification automation class is further described in section 4.2

## 4.2 The SMS and Email Module

The SMS and Email module takes care of all of the external functionality of our web application. This means that it handles most of the actual messaging functionality that will actually be sending messages to an agent's customer(s). The responsibilities of this module include sending alerts by both SMS and email, creating, editing, and deleting automated recurring messages, and sending mass alerts to several SMS or email recipients at once. This module is mainly used after the necessary information is gathered using our searching methods, whether it be from our map tool or the search bar itself. When the data is gathered, the user will then specify the service they want to use to send the alert and this module will take care of the actual sending of said alert based on the agent's preferences and input.

Below is the UML class diagram for the Services module consisting of the associated classes, their data fields, and the methods associated with each class that shows how each class interacts with one another to achieve the functionality that this module provides to our larger system.

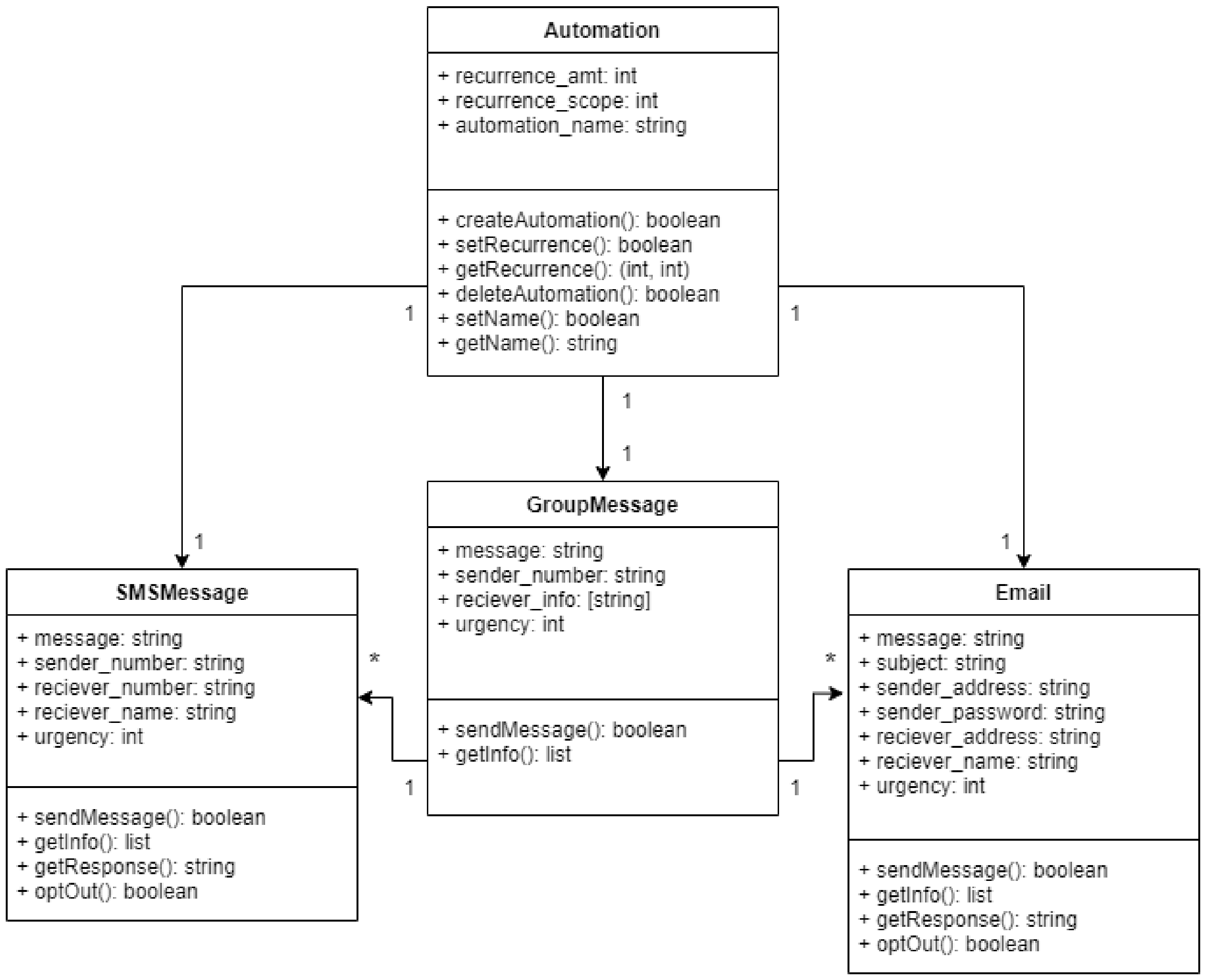


*Figure 4.0 - SMS and Email UML Diagram*

The communication module consists of four main classes; Automation, GroupMessage, SMSMessage, and Email. The most important of these classes is the last two, SMSMessage and Email.

### 4.2.1 SMS and Email

These are the two classes that are actually responsible for performing the action of sending messages out to the customers which is the main goal with our web application. These classes are very similar in that they both contain functions for sending messages be it email or SMS messages. They also have other functions for if the customer sends a response and to opt a customer out of a notification type in the future (if they reply with a certain phrase). The main difference between these two classes is the type of data they contain and use to perform their said functions. An example of this is how the Email class must be given a subject line value while the SMSMessage class doesn't need such an attribute as text messages do not have a subject line.

### 4.2.2 Group Message

Another important part of our project is the implementation of group messages. The class that achieves this functionality is the GroupMessage class as the name implies. This class takes in a list of receiver info and utilizes the SMSMessage and Email classes in order to actually send the messages. It has its own send_message function that loops through the list of receiver information and sends the given message to each member in that list.

### 4.2.3 Message Automation

The last class in this module is the Automation class which is responsible for message automation. This class uses the other three classes inside of itself in order to send messages on a particular date with a recurrence of a given length. The methods that it contains include methods for creating and deleting automations. There are also methods for renaming an instance of this class as well as editing the recurrence settings such as changing a message to be sent once a month instead of once a year. The class has three variables that can be edited after creation with the aforementioned methods; recurrence_name, recurrence_amt, and recurrence_scope. The recurrence_name variable contains the name that is given to a particular Automation instance. The recurrence_amt and recurrence_scope refer to the frequency of an Automation instance with recurrence_amt being the integer value between messages being sent and recurrence_scope referring to the integer flag for the incrementation amount of the recurrence_amt (i.e 1=days, 2=weeks, 3=years, etc). For instance, if recurrence_amt = 3 and recurrence_scope = 1, then the message would be sent every 3 days.

As you can see, this module forms a sort of hierarchy in which each class above the messaging classes uses the classes below them to help perform more and more complex tasks. We designed the module this way so that each class is modular and can use others in order to accomplish all of our project goals that we have put in place.

## 4.3 Map Module

Our web application provides an effective and efficient way for State Farm agents to interact and communicate with their respective clients. The map module serves as the primary interface for agents to find and select the clients that they would like to communicate with.

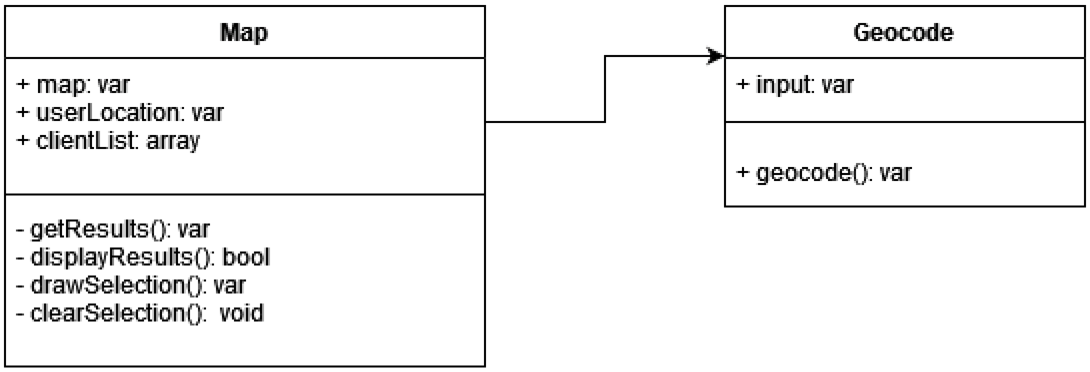


*Figure 5.0 - Map module diagram*

### 4.3.1 Map

The GIS mapping component lives in the front-end of our architectural overview. The map displays client locations and information based on a given agent's subset of clients. Most of the mapping display and logic is taken care of in what we'll call the "main" map class, there are other helper classes that support the map class, but the map class itself is what is predominantly responsible for what the end user sees on the screen in relation to the interactive map. Inside of the map class, Leaflet.js is what is used for rendering the visual map as well as all of the interactive functions available to the user. Users can search for clients by selecting specific regions on the interactive map using tools provided by the map class such as selection by drawing a specific area.

Map class fields:

- Map: This is the initialization of the Leaflet map. All operations in regards to the interactiveness and visual aspects of the map are handled by modifying this map field.
- userLocation: The location of the agent/user. This is used as an optional reference in order to more easily orient map operations in accordance with where the specific user is located.
- clientList: A list of clients

Map class functions:

- getResults(): Returns results in the form of a client list based on a selection on the map, such as a region defined by the user/agent.
- displayResults(): Takes in results in the form of a list of clients and displays their respective locations onto the map. This function can take in results from functions inside of the class such as the getResults(), or as a subset of data that comes directly from a user search outside of the class.
- drawSelection(): Handles operations involving the user defining regions on the map that they would like to define for further interactions. This returns a selection region in the form of boundary coordinates that can be later used to calculate clients that fall into that subsection.
- clearSelection(): Clears specified user selections on the map.

### 4.3.2 Geocode

When a specific region is selected by the user, this data will be sent to the geocoding class in order to translate latitude and longitude coordinates into addresses. This is an important feature to have since clients inside of the database are obviously stored with their addresses rather than coordinates and data retrieved from the interactive map is often in the form of geo coordinates. The geocoding class can also do the reverse of the process described above and turn addresses into geo coordinates as needed.

Geocode class fields:

- Input: This is the input provided from the map class that needs to be processed. This input comes in the form of either geo coordinates for geocoding, or in a normal address for reverse-geocoding.

Geocode class functions:

- geocode(): This is what takes in the input from the map class and performs specified geocoding operations. It has two parameters: boolean reverse and var input. Reverse is used to specify whether the function should perform regular geocoding(coordinates into address) or reverse geocoding(address into coordinates). The input is the data passed in by the map class that should be converted.

## 4.4 Dashboard Module

Our dashboard module helps bridge the gap between backend and frontend. Based off of this module we create web pages for our web application that really bring it to life. The dashboard module is responsible for all the elements that are viewed and used on our dashboard. This entails searching clients, reviewing saved search queries, viewing client subsets, and notifications settings. However, the actual notifications themselves are in their own module labeled Notifications, yet they are implemented onto the dashboard itself. This is similar to how the Mapping module works and interacts with the dashboard module. Overall, the dashboard will be where we implement the majority of the modules to create our project. We plan that the user will spend the majority of their time working within this dashboard module.

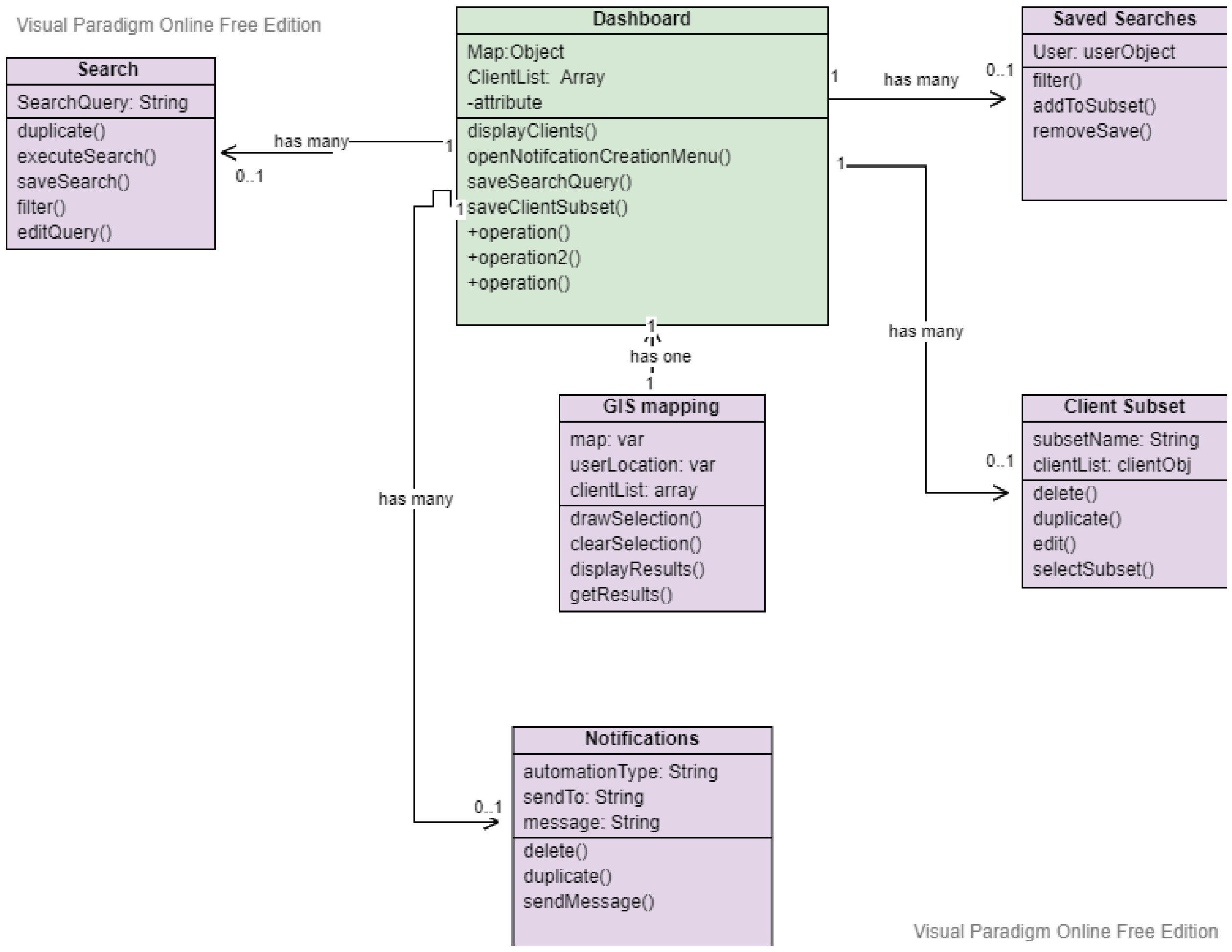


*Figure 6.0 - Dashboard Module Diagram*

The dashboard module consists of five main classes; Search, Notifications, GIS Mapping, Saved Searches, Saved Client Subsets.

### 4.4.1 Search

On the dashboard, user's will be able to easily search for clients. Within search queries they can filter certain elements to narrow results. These elements can range from Names, Age, or even policy. From there the user can create or add the client to a subset. The user can also just save the client search, for easier reference if adding to or creating a subset is not ideal yet.

Search Fields:

- Search Query: A string representing the search terms to re-execute.

Search Functions:

- Duplicate(): Creates a new query identical to the query being duplicated.
- executeSearch(): Executes the search query in the search bar.
- editQuery(): Allows the user to edit the string representing the search query.
- saveSearch(): Allows the user to save the search to the Saved Searched class.
- filer(): Allows the user to filter in or out certain key attributes when searching.

### 4.4.2 Saved Searches

Saved Searches are searches that the user finds important, however might not be ready to be placed into a subset. For example, if a user is inspecting an issue regarding two people with the same name, he may have one as his client and the other in the saved searches for easier reference.

Saved Searches Fields:

- User: Name of the saved search.

Client Subset Functions:

- filter(): Allows the user to filter in or out certain key attributes.
- addToSubset(): Will add the given search to a subset that the user inputs.
- removeSave(): Will remove the saved search from the saved search box.

### 4.4.3 Client Subset

Client subset are groups of clients created by the users to organize and easily interact with these groups if necessary. For example, a user could create a subset based on the same city, age ranges, policy, etc. The client subset class originates from the User Module (4.1). It is displayed in this UML diagram to help better show the relationship between the class and Dashboard module.

Client Subset Fields:

- subsetName: Name of the client subset.

- clientList: An array of client ids that the user has selected to be a part of the subset.

Client Subset Functions:

- delete(): Deletes the client subset from the database.
- duplicate(): Creates a duplicate client subset.
- edit(): Allows the user to change which clients are included in the subset.
- selectSubset(): Selects every single client in the subset and adds the subset of clients to the select client pool which allows the agents to send notifications to the subset.

### 4.4.4 GIS Mapping

The GIS mapping class is represented within the UML diagram, however is not directly a part of the Dashboard module. The class is demonstrated to show the relationship between the dashboard and the mapping system. The GIS mapping class is fully described in section 4.3.

Map class fields:

- Map: This is the initialization of the Leaflet map. All operations in regards to the interactiveness and visual aspects of the map are handled by modifying this map field.
- userLocation: The location of the agent/user. This is used as an optional reference in order to more easily orient map operations in accordance with where the specific user is located.
- clientList: A list of clients

Map class functions:

- getResults(): Returns results in the form of a client list based on a selection on the map, such as a region defined by the user/agent.
- displayResults(): Takes in results in the form of a list of clients and displays their respective locations onto the map. This function can take in results from functions inside of the class such as the getResults(), or as a subset of data that comes directly from a user search outside of the class.
- drawSelection(): Handles operations involving the user defining regions on the map that they would like to define for further interactions. This returns a selection region in the form of boundary coordinates that can be later used to calculate clients that fall into that subsection.
- clearSelection(): Clears specified user selections on the map.

### 4.4.5 Dashboard

The dashboard module is responsible for all the elements that are viewed and used on our dashboard. This entails searching clients, reviewing saved search queries, viewing client subsets, notifications settings, and interacting with the GIS map.

Dashboard Fields:

- Map: Display and interact with GIS map.
- clientList: An array of client ids that the user has selected to be a part of the subset.
- clientName: String names of clients that the user can select to interact with.

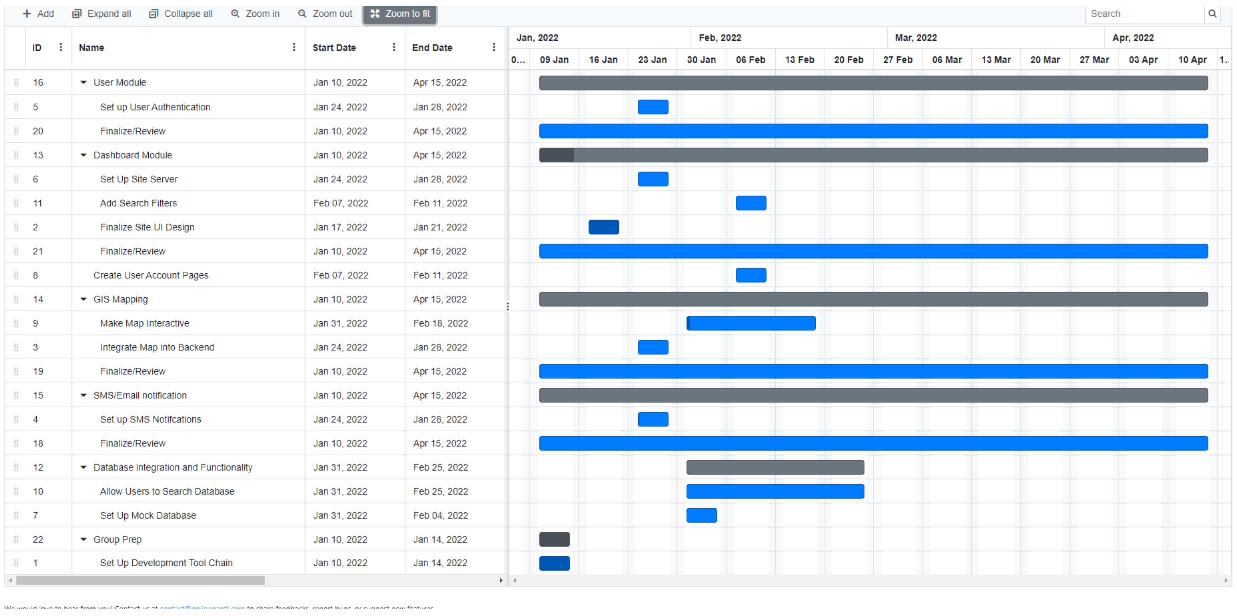
Dashboard Functions:

- displayClients(): Display list of clients from subset.
- openNotificationCreationMenu(): Opens notification menu for interaction with notification model.
- saveSearchQuery(): Saves search query.
- saveClientSubset(): Save the selected entire client subset.
- sendNotification(): Sends notification.
- search(): Allows users to search for clients or information from the dashboard.

### 4.4.6 Notifications

The Notifications class is another class that is represented within the UML diagram, however is not directly a part of the Dashboard Module. This class is demonstrated to show the relationship that the Dashboard will have with the notifications class. The notification class is fully described in section 4.2

# 5.0 Implementation Plan



| ID | Name | Start Date | End Date |
|---|---|---|---|
| 16 | User Module | Jan 10, 2022 | Apr 15, 2022 |
| 5 | Set up User Authentication | Jan 24, 2022 | Jan 28, 2022 |
| 20 | Finalize/Review | Jan 10, 2022 | Apr 15, 2022 |
| 13 | Dashboard Module | Jan 10, 2022 | Apr 15, 2022 |
| 6 | Set Up Site Server | Jan 24, 2022 | Jan 28, 2022 |
| 11 | Add Search Filters | Feb 07, 2022 | Feb 11, 2022 |
| 2 | Finalize Site UI Design | Jan 17, 2022 | Jan 21, 2022 |
| 21 | Finalize/Review | Jan 10, 2022 | Apr 15, 2022 |
| 8 | Create User Account Pages | Feb 07, 2022 | Feb 11, 2022 |
| 14 | GIS Mapping | Jan 10, 2022 | Apr 15, 2022 |
| 9 | Make Map Interactive | Jan 31, 2022 | Feb 18, 2022 |
| 3 | Integrate Map into Backend | Jan 24, 2022 | Jan 28, 2022 |
| 19 | Finalize/Review | Jan 10, 2022 | Apr 15, 2022 |
| 15 | SMS/Email notification | Jan 10, 2022 | Apr 15, 2022 |
| 4 | Set up SMS Notifcations | Jan 24, 2022 | Jan 28, 2022 |
| 18 | Finalize/Review | Jan 10, 2022 | Apr 15, 2022 |
| 12 | Database integration and Functionality | Jan 31, 2022 | Feb 25, 2022 |
| 10 | Allow Users to Search Database | Jan 31, 2022 | Feb 25, 2022 |
| 7 | Set Up Mock Database | Jan 31, 2022 | Feb 04, 2022 |
| 22 | Group Prep | Jan 10, 2022 | Jan 14, 2022 |
| 1 | Set Up Development Tool Chain | Jan 10, 2022 | Jan 14, 2022 |

*Figure 5.0 - Gantt Chart for Red Alert Production*

As seen in Figure 5.0, we have separated and planned out the time to construct our web application, Red Alert. Our main sections that we need to complete are our GIS mapping tool, Database integration and functionality, User Module, and our Dashboard Module. The tasks on our gantt chart will stem from these sections. Our clients have emphasized that they would like us to take a creative approach in our design and see what we are able to produce as a team. With that in mind, we truly feel as we have scheduled out all the elements required for the construction of our product. We have also allowed extra time near the deadline of our project for any issues not previously thought of.

Continuing, one of the more prevalent sections in our project is the GIS mapping tool. The tasks associated with this section involve learning the different metrics for geographical locating and how we can implement them into our system. Currently, we are able to highlight and select key locations onto our interactive map. We will build on this by making sure the mapping system is integrated into our backend so that we can save certain data such as highlighted locations, cities, or zip codes.

Another, important section within our project is the integration and functionality with our databases. Some of the tasks associated with this section involve setting up mock user data, as well as making sure users are able to search through the database. On the users end, they should be able to complete functions such as, searching for clients and have the given data return. Overall, this backend part of the project is key as it allows for simplicity on the user's end when it comes to having to interact with data pertaining to the web application.

Lastly, our Dashboard and User modules are key elements in our product. They will help be the glue when connecting the project together. Within the User module, tasks related to maintaining or creating users will be stored. This includes creating new accounts, logging in, and updating account information. Our Dashboard will utilize this module as well as the other to produce an environment where the State Farm agents can easily work. Our project has a mixture of loose and tight coupled webpages. For the loose coupled webpages such as log-in, we will agree on a design as a group and then we will separate the work for the construction of the web pages. One member will produce our log-in page, while another will produce our FAQ page, and etc.  For the tightly coupled web pages such as Dashboard, our group will agree on a design, then appoint a single member to take lead on the construction of the web page while the other members assist where needed. This method for the tight coupled webpage is expected to help reduce errors that could arise from constructing web pages with multiple inputs simultaneously.

## 6.0 Conclusion

Our project is a web application that will be used by agents at State Farm to send notifications to their clients. As of right now, State Farm agents have no easy way to visualize where their clients are located. Our project aims to rectify this issue by providing an easy and quick to use alert system that allows agents to search for clients both by attribute as well as visually on a map. As stated in this document, our system will consist of the web server, MongoDB database, and Django application. We then specify further in this document the finer aspects of each component and how they work together to create our web application. An organized and consistent design plan helps any project's development run smoothly and quickly. It also helps future developers maintain the project better. As a team, we hope to have achieved an organized and consistent design plan in order to make development easy on both our team, as well as any future teams at State Farm.