Architecture

TITAN

(**T**he **I**ntegrated,**T**echnologically **A**ssisted way to **N**avigate)

Hassan Hassan
Jon Montag
Alex Orellana
Greg Quintanilla
Griffin Rhein

Table of Contents……..1

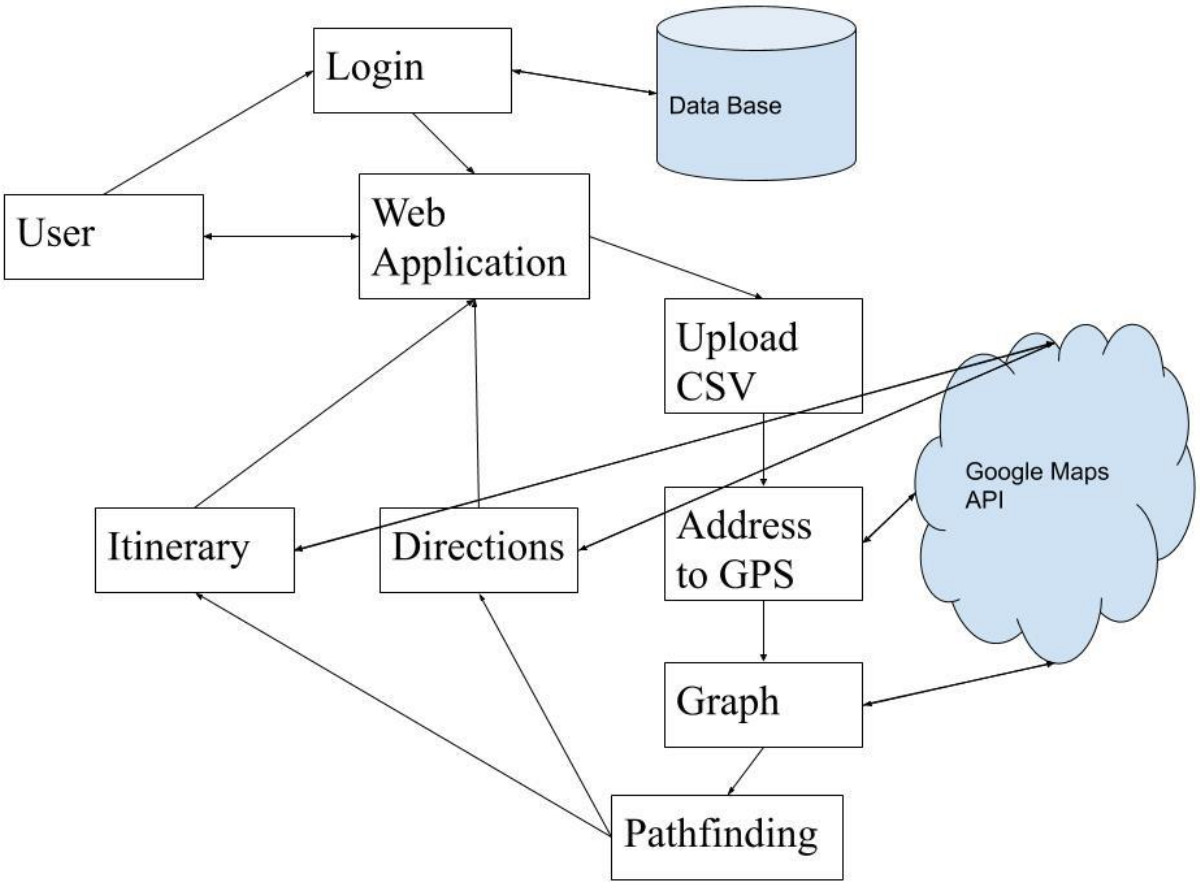
Introduction

Multiple control systems will be utilized, to streamline the process of connecting the individual components. For user view, the system will utilize a model-view-controller architecture to accommodate both the driver view and the administrator view, as both the amount of data and method of presentation varies depending on the user type. In the backend, a pipe and filter pattern will handle the processing and creation of routes.

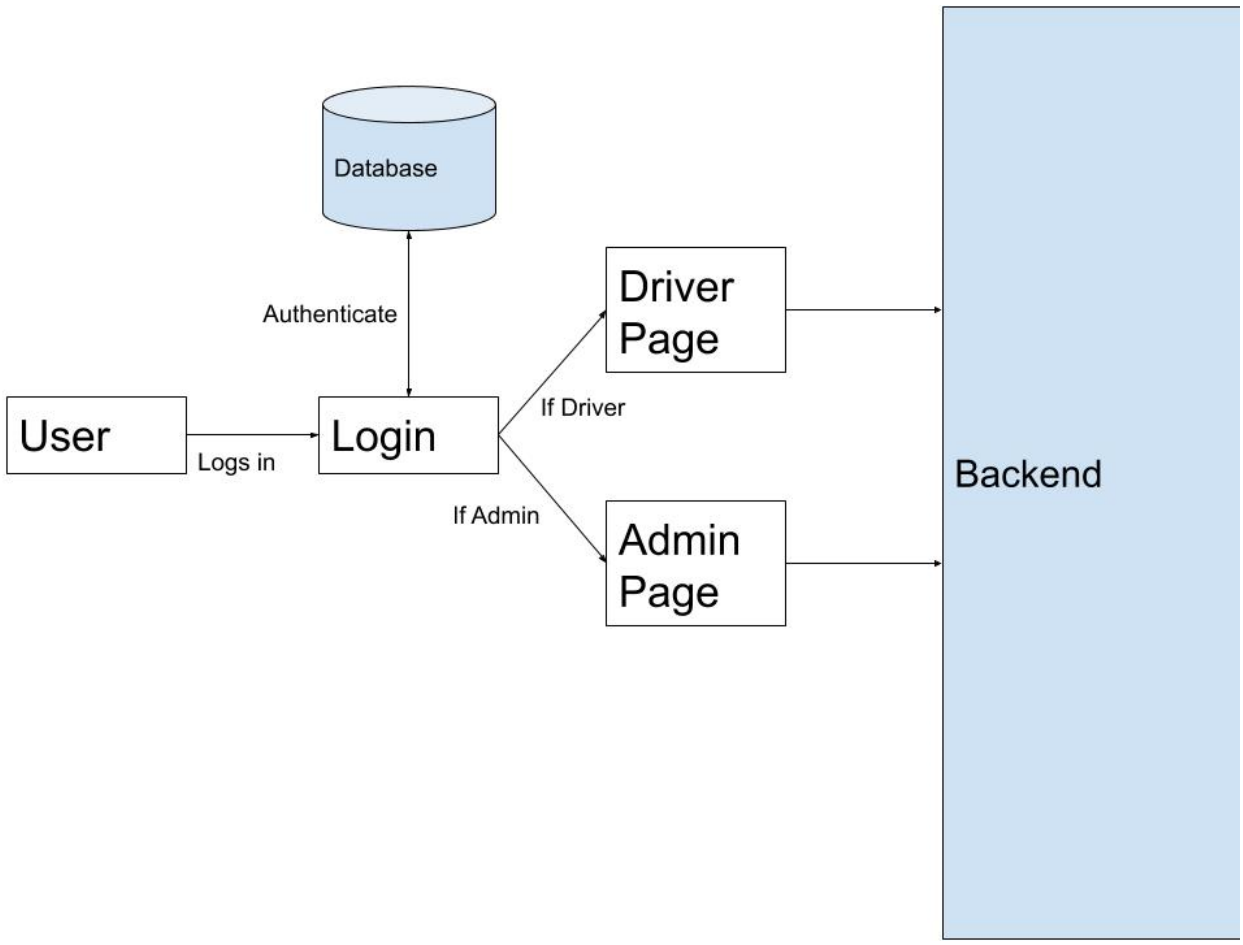
Addresses uploaded by an administrator will be parsed, transformed into GPS coordinates, placed into a central graph, and ordered into a path. Relevant information extracted and created in these steps will be returned at the directions and itinerary, while the Google API acts as a central repository. The web application's user interface for managing and using this information will be a hybrid implementation including both MPA (Multi-Page Application) and SPA (Single-Page Application), with the primary design pattern in use depending on the current view.

Architectural Representation

As outlined in the introduction, the main components to be combined into a complete architecture are a model-view-controller, a pipe and filter, and a repository structure. These will be tied together via a web application at the front end for the user to interact with. An overview box and line diagram is shown below:



The model-view-controller consists of the login page, the web application, and the database. As a user logs in using their credentials, their authorization will be checked against the username and password in the database, with passwords being encrypted for security. If authentication succeeds, the user is directed to the appropriate driver or administrator page which communicates with the application's backend and where the main functions of the app can be employed. If authentication fails, the user will be asked to enter their credentials again.



The backended process will be laid out in a sequential pipe and filter architecture that will take data from a list of addresses and process them all the way into a route with an itinerary and directions that will be ready to be presented back to the web application for administrators to then send out to drivers. This piece also has the most connection to the external Google Maps API. This API works as a central repository for the pipeline to retrieve some necessary information such as distances and location on a map.

The web application's user interface will follow a hybrid architecture that consists of MPA(Multi-Page Application) and SPA (Single-Page Application). This is needed because our web app has a login page and when a login page has been bypassed it needs to reload to a different page depending on the kind of user that has logged in.

This will implement the MPA architecture. But once a user has been shown their respective view, the SPA architecture will be used because it makes navigating around the user interface quicker and more responsive. A perfect example of this implementation is the web page gmail.com. When a user is at the gmail login page and then logs in, the entire page reloads to reflect the users corresponding view. This view is their inbox. But once a user is at their respective view, navigating around the page does not trigger any reloads. For example, if the user clicks on a particular email, then the user gets rendered a different view but the page does not get reloaded.

A strictly SPA web app is great for performance and maintainability but not very good for implementation of features.

A strictly MPA web app is great for features but harder to maintain and a little slower in terms of performance.

Architectural Goals and Constraints

This architecture aims to help define all of the small hookups necessary for the system to talk and function together. Within this project are many sub components with many connections that have to be streamlined. This is especially evident in the pipeline in the back end of the project. The data needs to be able to flow from point to point and be passed along seamlessly to ensure the system continues to work. This document does it's best to layout each individual component from the requirements and project plan documents and show how each needs to be fitted into the larger project.

The biggest issue with this architecture is the amount of subsystems are routines that need to link up and function. There will be a lot of time spent on integration

testing to ensure that each component talks to each of other components. There will be an equal amount of work put into the creation of the pieces as the connection between each point.

Use-Cases

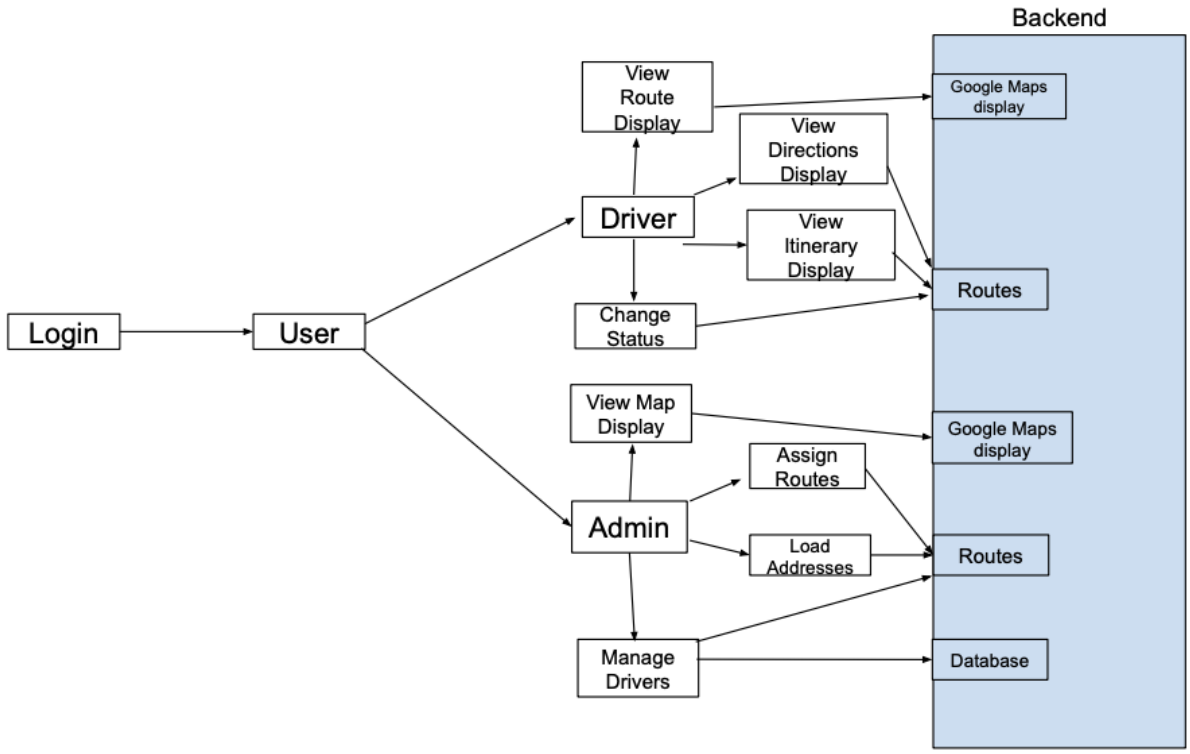
Drivers:

- Login
- View Map/Route display
- View Itinerary
- View directions
- Report Problems
- View ETA to next Stop
- Update Status

Administrators:

- Login
- View Map Display
- View Drivers
- Load addresses
- Calculate Routes
- Assign Drivers
- View Status of Current Routes

Architectural-Significant Use Cases



Driver:

1. View Route Display- Drivers will be able to view a map display that has markers and a highlighted path to indicate the direction of the route. This is a feature of the user interface and it communicates directly with the google maps API
2. View Directions Display- Drivers will be able to view the directions of the routes they're currently assigned to. This communicates with the route creation part of the backend. It also communicates with the google API because that's where the directions will be coming from.
3. View Itinerary Display- Drivers will see all their routes. This display communicates directly with the route creation part of the backend. If the routes

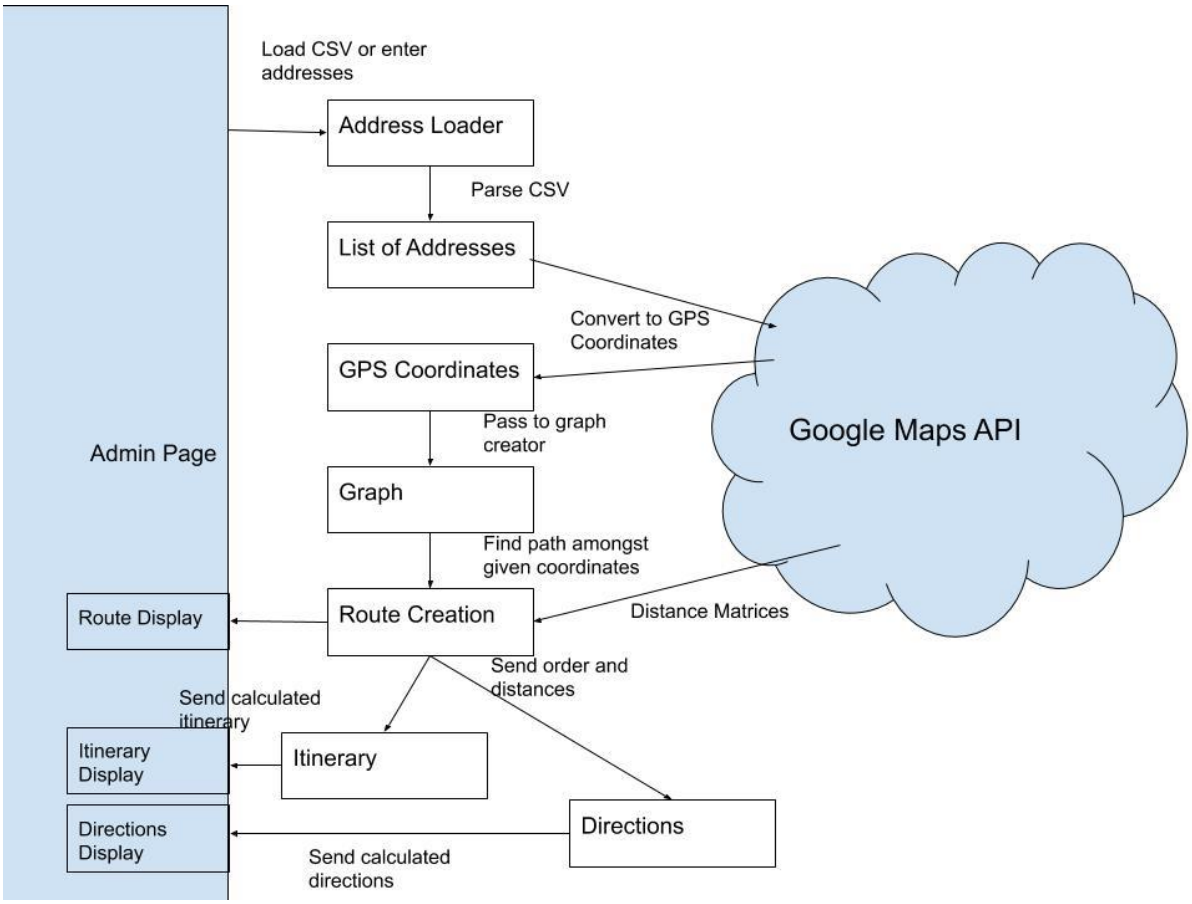
change or if new routes have been calculated, the itinerary will display those changes.

4. Change Status- This will allow drivers to change their status on their current routes. This communicates directly with the route creation part of the backend. If a route has been completed it should be removed from the backend.

Administrator:

1. View Map Display- Admins will see a map display and markers that indicate the destinations. This display communicates with the google maps API.
2. Assign Routes- Admins will be able to assign routes to drivers. This part has direct communication with the our routes backend because it needs to take the information from the routes that have been created and feed that information to the admin in order for the admin to properly assign routes.
3. Load addresses- Admins will be able to load addresses and calculate routes from those addresses. This communicates with the route management section of the backend.
4. Manage Drivers- Admins will be able to manage drivers. This interacts with the database where the drivers are stored.

Process View



The start of the process begins with the address loader from the web application where the administrator will enter in at most 100 addresses either by single entry or

uploaded through a comma separated values file. This data will then be preprocessed to a list of addresses. This list is passed along to the address to GPS coordinate converter, moved to the map graph which will house the information to then form a route where associated spatial and temporal information are used to create an itinerary and direction service. The route, itinerary and directions are all then passed back to the web application for display to the administrator. These steps will all be done in the Javascript language.

Address Loader

This function will take the uploaded addresses and parse them into a list of the addresses. It then passes it along to the address to GPS converter.

Address to GPS Converter

This function takes in the list of addresses and converts them into a list of all the GPS coordinates. This function will use a Google Maps API call to leverage their conversion into GPS coordinates. This newly constructed list will be passed along to the Map graph

Map Graph

Required to store and hold each of the addresses that represent the buildings to be delivered to. The addresses uploaded by the administrator will be converted into LatLng objects with an API request. From here a distance matrix can be requested and this information will then be converted into a complete graph. This feature will primarily be housed and retrieved by the Google Maps API. The Google Maps API is based in the Javascript language.

This is the first step after the entered addresses have been converted into GPS coordinates. With these addresses now in GPS coordinates a graph of addresses will be generated and once the necessary information is pulled they will be passed along to the pathfinding function once the user indicates that all the necessary addresses have been entered.

Pathfinding

Using Javascript and the Google Map API a route will be generated from the distance matrix created in the Map Graph. The administrator may choose whether time or distance will be the factor to organize the route by. The feature will choose the corresponding addresses in the order they should be traveled.

The Use cases of this feature will be for the administrators primarily. Once a set of addresses to deliver to uploaded or entered, the graph of all the points will be generated and then the Pathfinding routine will be called to create and deliver the path to be sent to the drivers. This routine will use a greedy implementation to construct the shortest path as it always selects the shortest distance from the current address. This is one of the simplest and easiest ways to compute the route.

Once the path is generated the information will be passed along to the direction and itinerary creator.

Itinerary

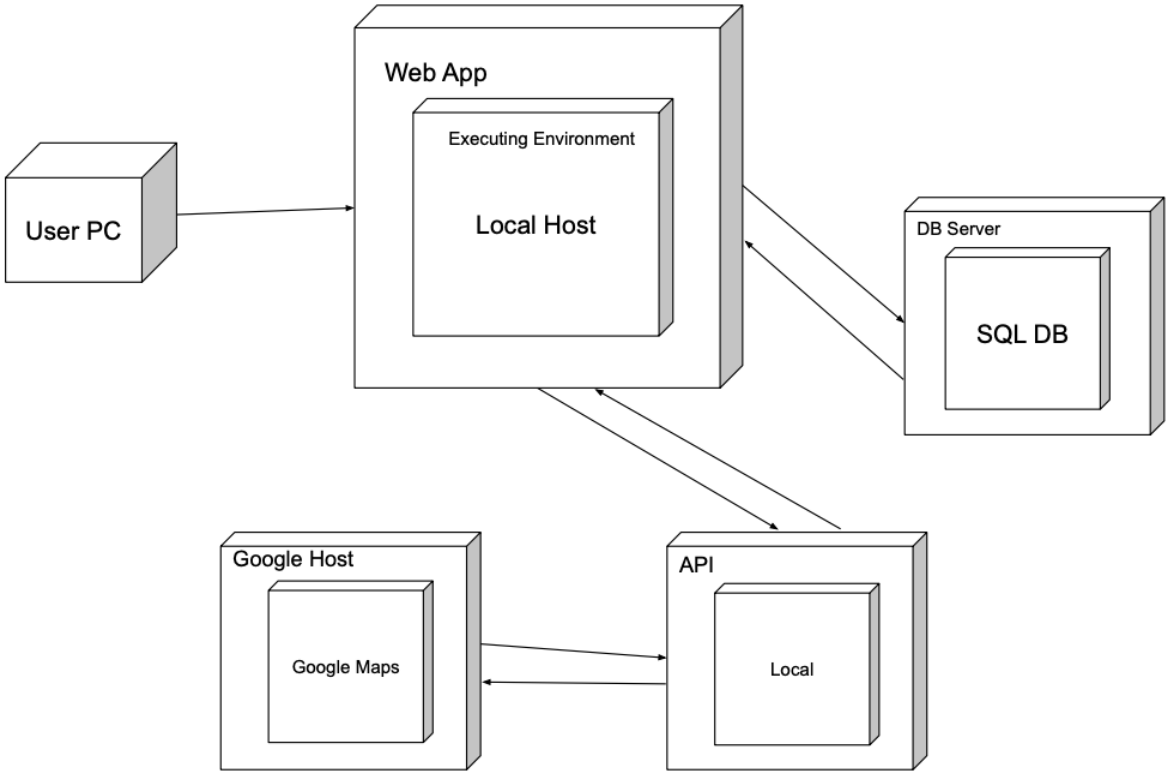
The itinerary which is visible only in the driver view holds information regarding the scheduled stops. Information includes distance in mileage to get there and estimated time to get there. The itinerary communicates directly with the route creation part of our systems and due to this it indirectly communicates with other parts all the

way back to the address loader part of our system. When a new address gets loaded by the administrator the itinerary should be able to display the change accordingly.

Directions

The directions are a part of the user interface display and it is only visible by the drivers. This part interacts with the route creation part of the backend and it also interacts with the google API to get directions based on the routes.

Deployment View



1. Users will use the software locally but it will require an internet connection.
2. The web application will be executed locally and users can access it by typing in the local address.

3. The database will be stored and managed on a database server. This is particularly useful for our application because the database needs to be queried an unspecified number of times and it needs to be able to do it quickly. It also would need to be updated constantly and possible be queried by multiple users at the same time. Keeping this locally would either not provide the functionality that we want or it will but it would not be efficient.
4. The API for our application will be set up locally. This APi will send and receive requests to the google API.
5. The google maps API will be hosted by google. The API created by us will make and receive requests to the google API.

If our application needed to be set up for enterprise deployment we would need to deploy and host our application on a custom domain. Our API will need to also be set up on a server so it can be accessed remotely.

Size and Performance

Our application should be able to calculate routes in a reasonable time. The routes should be accurate and efficient. Users should be able to access the web app only needing an internet connection and the local host URL. Our application would not require any disk space or any installation to the user pc. The user would need a compatible web browser like chrome or safari.

Our application should support no more than 20 drivers at a given time. Only one administrator can be logged in and manage a group of drivers. The maximum number of addresses that can be processed by our application should be no more than 100.

The login page and authentication should not have a latency that exceeds 5 seconds. Our database should be queried quickly and should return proper information if it exists. If information is updated, it should be done quickly and efficiently and it should be reflected on all individual components.

The interaction between our API, the google maps API, and our web app frontend should be done so with little to no latency.

The loading time of our web application should be not excessively long. Since we have chosen a hybrid architecture implementation of our web app, users should get the best of performance and responsiveness alongside the best of features.

Quality

In keeping with the quality assurance section from Project Plan rev. 1.0, our system should be robust meaning that it should not have a down time of more than 3%.

The user interface should have properly formatted sections in order to maintain ease of use. Upgrades or changes to components should be made without disrupting the user too much.

The web app implementation should use an architecture that is well known in order to preserve it’s maintainability. The web application will be built using well known and open web technologies so that it can be accessible through major web browsers.

Different components of our system should be created not only with the ability to function in our specific application but in various other applications in order to maintain its reusability.