Course Title: 14:332:452 Software Engineering

Group 4

Onward (Traffic Monitoring)

Report 2

March 12, 2017

GitHub (e-archive): https://github.com/solejar/traffic_ru_ece
Website: http://www.onwardtraffic.com/

Team Members

Name	Email
Ridwan Khan	ridwankhan101@gmail.com
Brian Monticello	b.monticello23@gmail.com
Mhammed Alhayek	almoalhayek@gmail.com
Sean Olejar	solejar236@gmail.com
Lauren Williams	laurenwilliams517@gmail.com
Shubhra Paradkar	shubhra.paradkar@gmail.com

Individual Contributions Breakdown

All team members contributed equally.

Table of Contents

Section 1: Interaction Diagrams

For our interaction diagrams, we have chosen to do sequence diagrams rather than collaboration diagrams as our system is more easily understand through a chronological order of events rather than a hierarchy of objects.

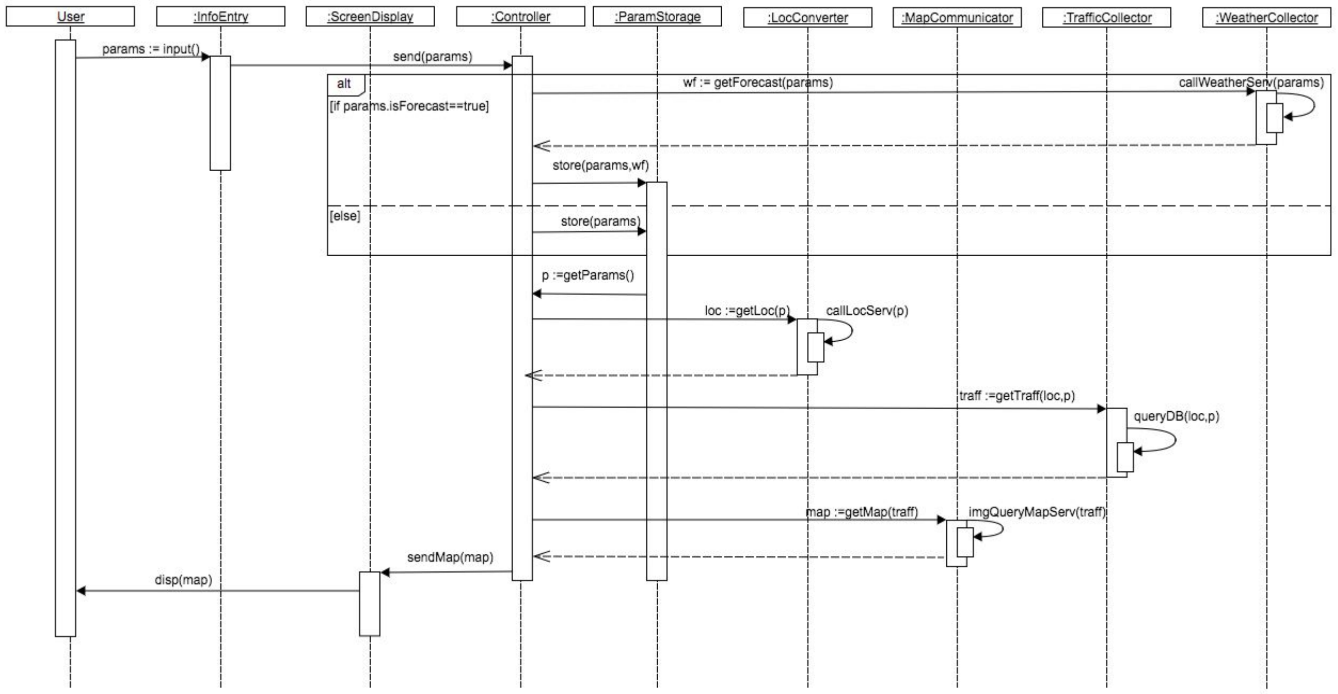


Figure 1: Success sequence diagram of UC-1 with extension UC-6

This diagram shows the success scenario for both UC-1, the “Heat Map” feature, and its extension UC-6. UC-6 is the extension in which the user decides to use forecasted weather as opposed to entering it in manually, therefore UC-6 does not have its own sequence diagram. This diagram is initiated by the user choosing the “Heat Map” feature on the landing page and beginning to input parameters. This diagram corresponds to the combination of Figure 5 and Figure 7 system sequence diagrams from Report 1.

InfoEntry and ScreenDisplay are user-facing JavaScript objects, and every other object is implemented as server-side PHP. We designed all of our concepts with high cohesion and modularity in mind. Having all communication mediated by the Controller entity allows us to decouple the individual components of our system. This helps keep our system modular, since we can add and remove components without risk of disturbing the functioning of other components. Additionally, each component has a focused, cohesive purpose.

This sequence diagram is directly derived from our domain model, with the change that MapDisplay is now renamed to ScreenDisplay. We have made this change from Report 1 because we realized that this module is not only responsible for the map, but any other changes on-screen such as the statistical data graphs.

As described later in our alternate design decisions, we originally had an issue distinguishing in our system between the different parameters of UC-1 and UC-6. In order to clarify the origin of the parameters, we added the entity ParamStorage, which handles the job of distinguishing and storing the different parameters for the different use cases.

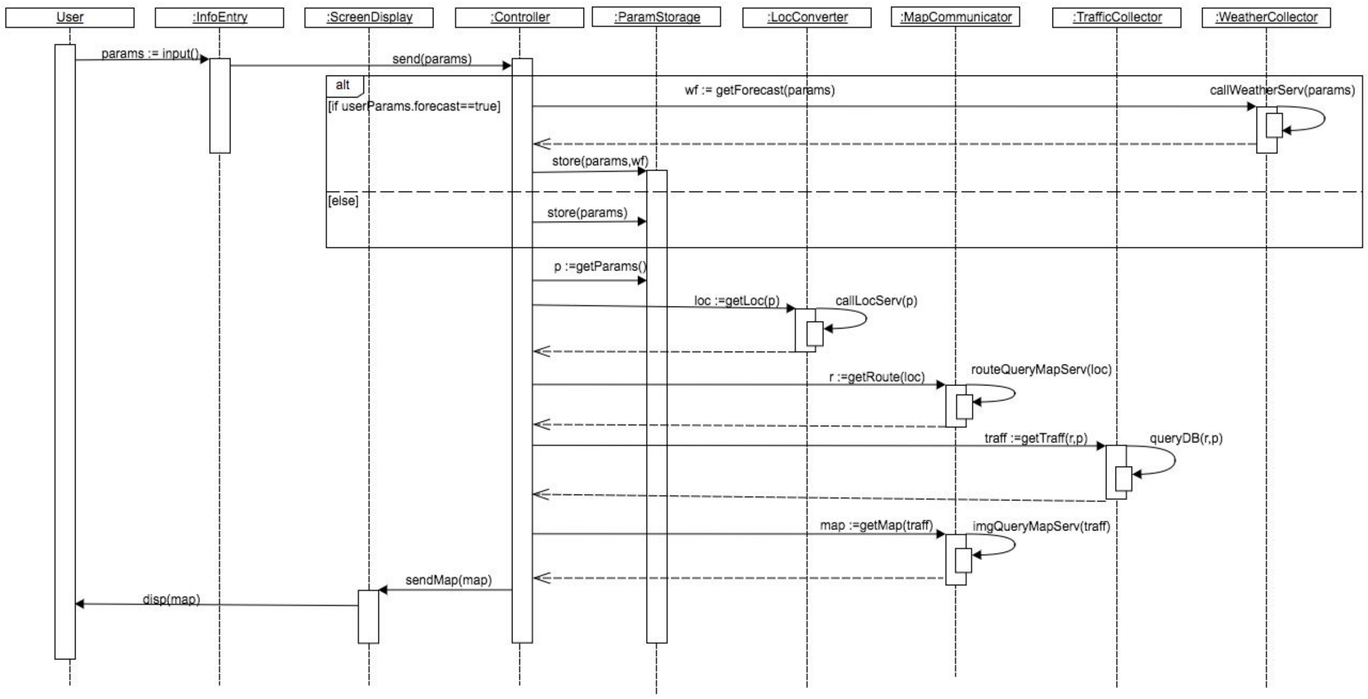


Figure 2: Success sequence diagram of UC-2 with extension UC-6

This diagram incorporates both UC-2, the “Route” feature, and its extension UC-6. This diagram is similar to Figure 1 above. The components used are all the same, though the functions called are specific to the route feature. As a result, the design principles mentioned in the description for Figure 1 still apply here. This diagram is initiated by the user choosing the “Route” feature on the landing page and inputting parameters. This diagram corresponds to the combination of Figure 6 and Figure 7 system sequence diagrams from Report 1. Both use cases use the same conceptual objects, which is made possible by our reusable design. The attributes and operations of the classes in Figure 1 and Figure 2 are described in the Section 2, Class Diagram.

Alternative Design

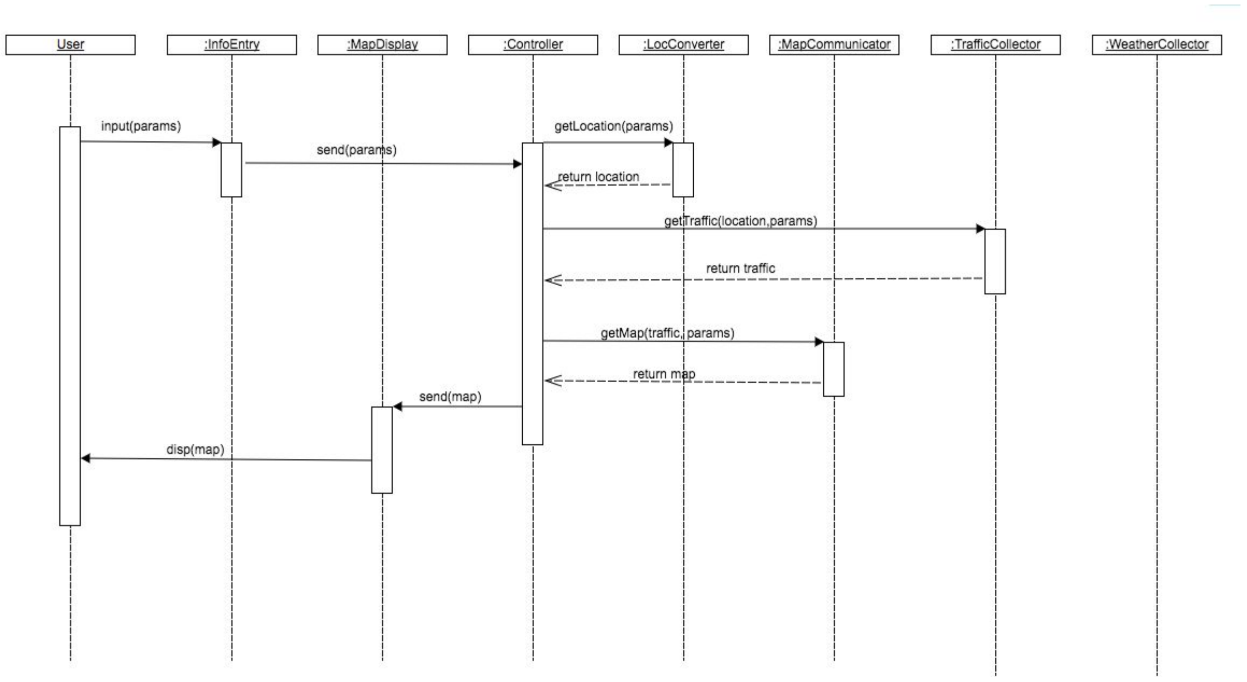


Figure 3: Alternate Design Diagram

This diagram was the first design we came up with for the UC-1 sequence diagram. In this diagram, InfoEntry sends the user inputted parameters to the controller, which stores these parameters for use by the other components. We decided against this because we felt that it was non-cohesive to have the Controller concept also in charge of info storage. This is shown in our final design for the interaction diagram (Figure 1), where the concept ParamStorage receives the user inputs and stores them. We also found this necessary because not all of the parameters that are used come from the user, such as in UC-6. The addition of the ParamStorage concept allows us to resolve parameters of different origin into one easy-to-access location. The final design is, as a result, more cohesive because the Controller is now no longer responsible for keeping track of parameter info. Figure 3 is incomplete for the reason that it was a deprecated design and was abandoned. Many of the namings and concepts are incorrect in this diagram as it was a very early phase in our design of the diagrams.

Section 2: Class Diagram and Interface Specification

Class Diagram

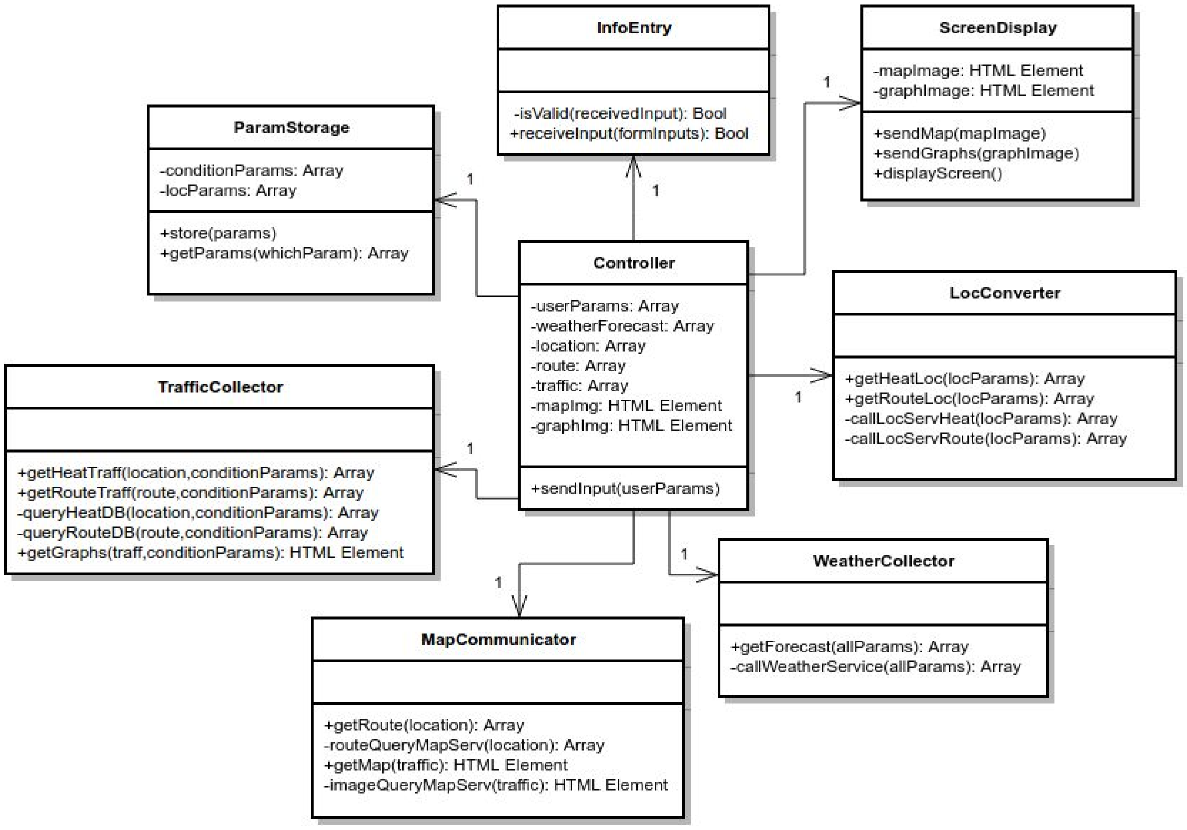

InfoEntry
-isValid(receivedInput): Bool
+receiveInput(formInputs): Bool
ScreenDisplay
-mapImage: HTML Element
-graphImage: HTML Element
+sendMap(mapImage)
+sendGraphs(graphImage)
+displayScreen()
ParamStorage
-conditionParams: Array
-locParams: Array
+store(params)
+getParams(whichParam): Array
Controller
-userParams: Array
-weatherForecast: Array
-location: Array
-route: Array
-traffic: Array
-mapImg: HTML Element
-graphImg: HTML Element
+sendInput(userParams)
LocConverter
+getHeatLoc(locParams): Array
+getRouteLoc(locParams): Array
-callLocServHeat(locParams): Array
-callLocServRoute(locParams): Array
TrafficCollector
+getHeatTraff(location,conditionParams): Array
+getRouteTraff(route,conditionParams): Array
-queryHeatDB(location,conditionParams): Array
-queryRouteDB(route,conditionParams): Array
+getGraphs(traff,conditionParams): HTML Element
WeatherCollector
+getForecast(allParams): Array
-callWeatherService(allParams): Array
MapCommunicator
+getRoute(location): Array
-routeQueryMapServ(location): Array
+getMap(traffic): HTML Element
-imageQueryMapServ(traffic): HTML Element
1
1
1
1
1
1
1

Figure 4: Class Diagram

This is the class diagram for our application. As you can see, all classes' interactions are mediated entirely by the Controller class. This was done intentionally to make our system modular. Eliminating visibility between the classes makes it easy to add and remove modules as required (perhaps for yet-to-be-implemented use cases). When a specific class needs information from another class, it will simply communicate that to the controller, which will then request the information from the appropriate class and then return it back to the class that originally asked for it. For example if the MapCommunicator class needs traffic information from the TrafficCollector class, it will receive that information from the Controller. The Controller would use the getter functions of the TrafficCollector to retrieve this information and pass it back to the MapCommunicator class.

Data Types and Operation Signatures

InfoEntry: Class that is responsible for handling parameter entry for the Heat Map or route feature. Responsible for validating inputs before sending them to the controller.

Operations:

- **-isValid**(*receivedInput: Array*): Bool
    - Checks the user's input for validity. If the entries are valid, it returns true, otherwise it returns false. Parameter receivedInput is an array that contains all the user's inputs.
- **+receiveInput**(*formInputs: Array*): Bool
    - Returns true if the user successfully completed the form to generate a Heat Map or route, otherwise it returns false. Parameter formInputs is an array that contains all the user's inputs.

ScreenDisplay: Class that is responsible for holding all visual elements, such as the Map GUI and the statistical graphs. It is also responsible for displaying these elements.

Attributes:

- **-mapImage**: HTML Element
    - Element provided by the Google Maps API that allows the application to embed the map GUI.
- -**graphImage:** HTML Element
    - This is a graph of statistical data about the traffic.

Operations:

- **+sendMap**(*mapImage: HTML Element*)
    - Takes in the HTML Element for the map as a parameter and uses that to store the map that will be displayed to the user.
- **+sendGraph**(*graphImage: HTML Element)*
    - Takes in the HTML Element for the graph as a parameter and uses that to store the graph that will be displayed to the user.
- **+displayScreen**()
    - This function actually displays the map and graph to the user.

ParamStorage: Class that stores all parameters obtained from the user and from API calls and provides them to functions that need them.

Attributes:

- **-conditionParams**: Array
    - Array of parameters about the traffic and weather conditions for the required locations as well as date and time.
- **-locationParams**: Array
    - Array of parameters about the location that the user needs data for. For the Route feature, this could be starting and ending coordinates. For the Heat Map feature, this could be a zip code and bounding box.

Operations:

- **+store**(*params: Array*)
    - Function that takes all the parameters and divides them up into location parameters and condition parameters then sets the appropriate attributes. The parameter fullParams is an array of all the parameters obtained from the user and API calls.
- **+getParams**(*whichParam:* String): Array
    - Returns an array of either conditionParams, locationParams, or all parameters as an array depending on the string input.

Controller: Responsible for mediating interactions between all other objects (keeping them loosely coupled), and limiting inter-object visibility.

Attributes:

- **-userParams**: Array
    - The portion of parameters specified by the user. In some use cases, these are all parameters used. In other use cases, these are combined with parameters generated by participating actors.
- **-weatherForecast**: Array
    - An array of forecast info generated by the WeatherCollector.
- **-location**: Array
    - A bounding box of lat/long values stored in an array
- **-route**: Array
    - An array of routes that connect two locations.
- **-traffic**: Array
    - Traffic severities in an array
- **-mapImg**: HTML Element
    - Google Maps GUI element, with traffic severities highlighted
- **-graphImg**: HTML Element[9]
    - Statistical data about the traffic, represented as a graph.

Operations:

- **+sendInput**(*userParams: Array*)
    - Acquires user parameters from the client side.

LocConverter: Responsible for taking a user's input parameters for location, such as a zip code or city name, and converting it to a lat/long format

Operations:

- **+getHeatLoc**(): Array
  - Getter function, returns a lat/long bounding box in an array.
- **+getRouteLoc**(): Array
  - Getter function, returns an array of latitudes and longitudes corresponding to the user-inputted start and end coordinates
- **-callLocServHeat**(*locationParams: Array*): Array
  - Helper function for getHeatLoc(). Takes the user-inputted zip code, and uses it to make an API call to the Location Service. Returns a lat/long bounding box in an array.
- **-callLocServRoute**(*locationParams: Array*): Array
  - Helper function for getRouteLoc(). Takes the user-inputted start and end coordinates, and uses them to make an API call to the Location Service. Returns lat/long formatted coordinates.

MapCommunicator: Responsible for acquiring routes and map images from the Mapping Service actor

Operations:

- **+getRoute**(*location: Array*): Array
  - Getter function, takes an array of 2 lat/long coordinates, returns an array of 3 routes that connect those coordinates.
- **-routeQueryMapServ**(*location: Array*): Array
  - Helper function for getRoute(). Makes API call to Mapping Service with input coordinates, returns array of routes connecting them.
- **+getMap**(*traffic: Array*): HTML Element
  - Getter function, takes an array of traffic severities, returns an HTML Element representing the Google Maps GUI displayed to the user.
- **-imageQueryMapServ**(*traffic: Array*): HTML Element
  - Helper function for getMap(). Makes API call to Mapping Service, returns GUI element with all the traffic severities highlighted

WeatherCollector: Responsible for acquiring the weather forecast from the Weather Service.

Operations:

- **+getForecast**(*allParams: Array*): Array
  - Getter function, takes an array of date/time info as parameters, outputs array of strings that detail weather forecast
- **-callWeatherService**(*allParams: Array*): Array
  - Helper function used by getForecast(). Makes an API call to the weather service, and formats the API's JSON response into an array of forecast info

TrafficCollector: Responsible for accessing our database to get traffic severities associated with locations or routes.

Operations:

- **+getHeatTraff**(*location: Array*): Array
  - Getter function, takes in a location bounding box as input, returns an array of traffic severities of roads inside that bounding box.
- **+getRouteTraff**(*route: Array*): Array
  - Getter function, takes in an array of routes, returns an array of traffic severities for all roads in the routes.
- **-queryHeatDB**(*location: Array,conditionParams: Array*): Array
  - Helper function for getHeatTraff(). Takes a location bounding box and array of weather, date, and time conditions as parameters. Queries the DB for all traffic severities that match those params, returns those severities as array.
- **-queryRouteDB**(*route: Array,conditionParams: Array*): Array
  - Helper function for getRouteTraff(). Takes an array of routes and array of weather, date, and time conditions as parameters. Queries the DB for all traffic severities that match those params, returns those severities as array.

Traceability Matrix

Classes:	Controller	InfoEntry	ParamStorage	LocConverter	TrafficCollector	MapComm	WeatherCollector	ScreenDisplay
Domain Concepts								
Controller	x							
Info Entry		x						
Parameter Storage			x					
Location Converter				x				
Traffic Collector					x			
Map Communicator						x		
Weather Collector							x	
Screen Display								x

Figure 5: Class Diagram Traceability Matrix

As you can see from the traceability matrix, there is a one-to-one relationship between the classes and domain concepts. For each domain concept we had, we created one class. We constructed our domain concepts with object-oriented design principles in mind. Each concept's role was cohesive and focused. As a result, when we designed our classes, there was little that needed to be changed. This is also a byproduct of the fact that our system contains relatively simple interactions, which do not necessitate complicated class design. We also make note of the fact that in Report 1 the concept "Map Display" is now the concept, and class, "Screen Display".

Section 3: System Architecture and System Design

Architectural Style

Onward is a web-based application, and the architectural style that most aligns with our system is a three tier architecture. Onward consists of a user interface presentation tier, functional process logic with business rules tier, and a data tier. The three tier architecture is appropriate for our system because the user interface, functional logic, and data are all independent modules of our system. The user interface, which is our presentation tier, can be visually modified in any way without altering the functional process logic and database. This allows the tier to have a certain level of independence from the other tiers. In addition, we have a logic tier that consists of entities that handle the application's logic and information processing. Furthermore, we also have persistent data storage that is explored further in the "Persistent Data Storage" section below. The data tier consists of any data that the logic tier may need for processing. The logical tier interacts with the user interface and the data tier, but the user interface and data have no direct interactions, allowing for modularity and effective decoupling of each tier. The architectural style and the subsystems are further explained in the section below.

Identifying Subsystems

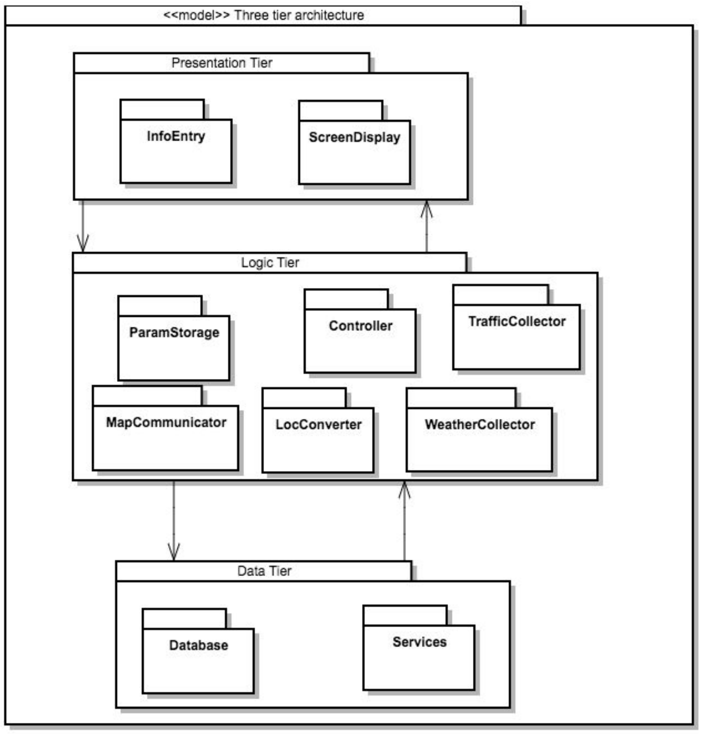


Figure 6: Package Diagram

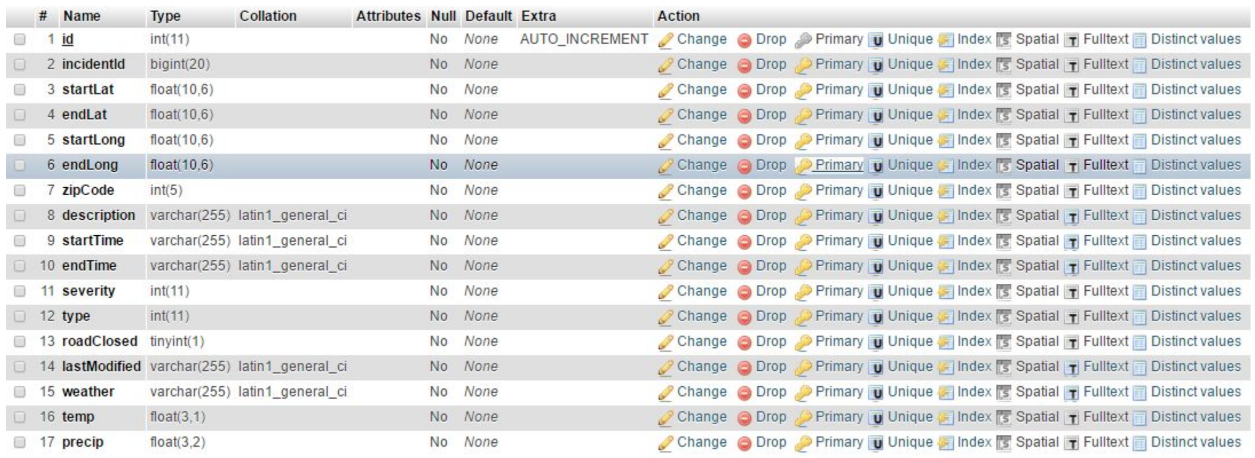
The subsystems for Onward can be broken into the three different tiers of the system architecture: Presentation, Logic, and Data, as can be seen in Figure 6. These are our subsystems because each tier is a large enough subsystem to contain its own set of packages that are independent of each other. The Presentation Tier consists of the InfoEntry and ScreenDisplay concepts from our domain model as these concepts are responsible for the user interface of our system. The Logic Tier consists of these concepts from our domain model: Controller, ParamStorage, LocConverter, MapCommunicator, WeatherCollector and TrafficCollector. These domain concepts handle the application's logic and information processing. Finally, in the Data Tier subsystem, two generalized packages - Database, and Services - are depicted to represent the specific functionalities of the Data Tier that are based on the participating actors, where the Database is the participating actor itself and the Services package includes Mapping Service and Weather Service. As can be seen from our interaction diagram, the concepts in the logical tier query both the database, using SQL, and the data services, and then process this data. After processing the data, the logical tier sends the results to be presented in the user interface. All of these packages are unique to each subsystem; however, the different tiers themselves are communicate with each other in a limited way, as can be seen by the various arrows pointing between the different tiers.

Mapping Subsystem to Hardware

The above subsystems are all not housed in the same hardware. The presentation tier is client-side, and involves either the user's computer or mobile device and their respective web browser. The logic tier houses the controller (among other entities) and maps the input from the presentation to the data tier. This tier and all of its modules are located on a remote server. The data tier works to retrieve the necessary information and supplies it back to the logic tier. This tier, which includes the database of collected data for our application, is located on the same remote server. The server we are using to house the data and logic tier is a webhosting service, 1and1.com.

Persistent Data Storage

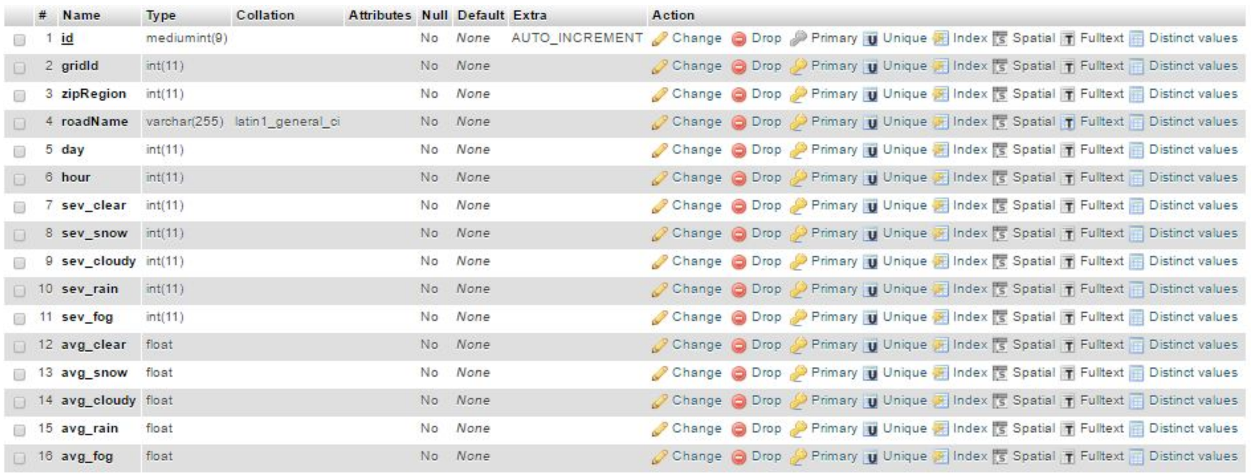
Below are the relational database tables that Onward uses to perform the various tasks needed to collect traffic data, calculate average road/area severity, and collect weather frequency information. Our database is a crucial part of our system and has been designed with great detail and thought. Along with a graphic of each table is an accompanied description. The algorithms for the database are further described in the "Algorithms" portion of this report. We were unable to retrieve the schema using the "description" command but we have attached screenshots of the schema.



#	Name	Type	Collation	Attributes	Null	Default	Extra	Action
1	id	int(11)			No	*None*	AUTO_INCREMENT	Change Drop Primary Unique Index Spatial Fulltext Distinct values
2	incidentId	bigint(20)			No	*None*		Change Drop Primary Unique Index Spatial Fulltext Distinct values
3	startLat	float(10,6)			No	*None*		Change Drop Primary Unique Index Spatial Fulltext Distinct values
4	endLat	float(10,6)			No	*None*		Change Drop Primary Unique Index Spatial Fulltext Distinct values
5	startLong	float(10,6)			No	*None*		Change Drop Primary Unique Index Spatial Fulltext Distinct values
6	endLong	float(10,6)			No	*None*		Change Drop Primary Unique Index Spatial Fulltext Distinct values
7	zipCode	int(5)			No	*None*		Change Drop Primary Unique Index Spatial Fulltext Distinct values
8	description	varchar(255)	latin1_general_ci		No	*None*		Change Drop Primary Unique Index Spatial Fulltext Distinct values
9	startTime	varchar(255)	latin1_general_ci		No	*None*		Change Drop Primary Unique Index Spatial Fulltext Distinct values
10	endTime	varchar(255)	latin1_general_ci		No	*None*		Change Drop Primary Unique Index Spatial Fulltext Distinct values
11	severity	int(11)			No	*None*		Change Drop Primary Unique Index Spatial Fulltext Distinct values
12	type	int(11)			No	*None*		Change Drop Primary Unique Index Spatial Fulltext Distinct values
13	roadClosed	tinyint(1)			No	*None*		Change Drop Primary Unique Index Spatial Fulltext Distinct values
14	lastModified	varchar(255)	latin1_general_ci		No	*None*		Change Drop Primary Unique Index Spatial Fulltext Distinct values
15	weather	varchar(255)	latin1_general_ci		No	*None*		Change Drop Primary Unique Index Spatial Fulltext Distinct values
16	temp	float(3,1)			No	*None*		Change Drop Primary Unique Index Spatial Fulltext Distinct values
17	precip	float(3,2)			No	*None*		Change Drop Primary Unique Index Spatial Fulltext Distinct values

Figure 7: Traffic Incident Table

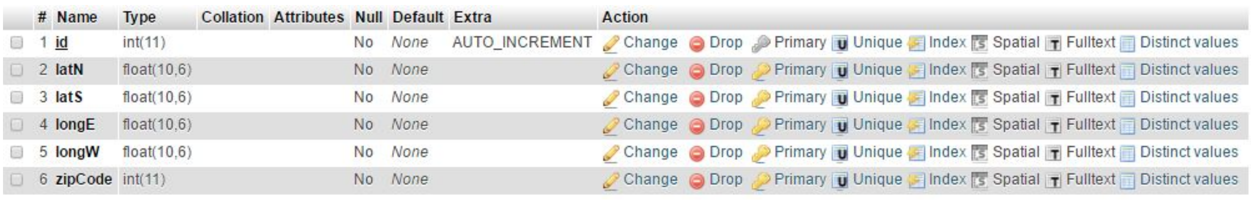
Above is the table used to collect the traffic incident data. The Bing Traffic API[1] is used to collect information about the specific traffic incidents that occur by specifying a "bounding box" of coordinates that represents the region over which we have chosen to collect traffic data. On the top of every hour, our server runs a cron job[4] that calls the Bing Traffic API, parses the JSON result, and stores the relevant data in this table. It is also necessary for Onward to have the specific weather condition associated with the incident in order to accurately present to users future traffic under certain weather. In regards to this, we took the starting latitude and longitude collected from the Bing Traffic API for each incident, used a geocoding API[3] to convert it to a zip code, and then used that zip code to call the Weather Underground API[2] to get the corresponding weather condition, temperature and precipitation level. Additionally, in this table, certain traffic incidents can be modified by Bing at any time, so each cron job run checks to see if the last modified date of each incident has changed, and if it has, updates the entire entry (neglecting weather condition). We keep all this data for each entry, even if it may be extraneous, so that as we move forward in development and we feel we need the data it is there.



#	Name	Type	Collation	Attributes	Null	Default	Extra	Action
1	id	mediumint(9)			No	None	AUTO_INCREMENT	Change Drop Primary Unique Index Spatial Fulltext Distinct values
2	gridId	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
3	zipRegion	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
4	roadName	varchar(255)	latin1_general_ci		No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
5	day	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
6	hour	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
7	sev_clear	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
8	sev_snow	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
9	sev_cloudy	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
10	sev_rain	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
11	sev_fog	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
12	avg_clear	float			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
13	avg_snow	float			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
14	avg_cloudy	float			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
15	avg_rain	float			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
16	avg_fog	float			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values

Figure 8: Severity Table

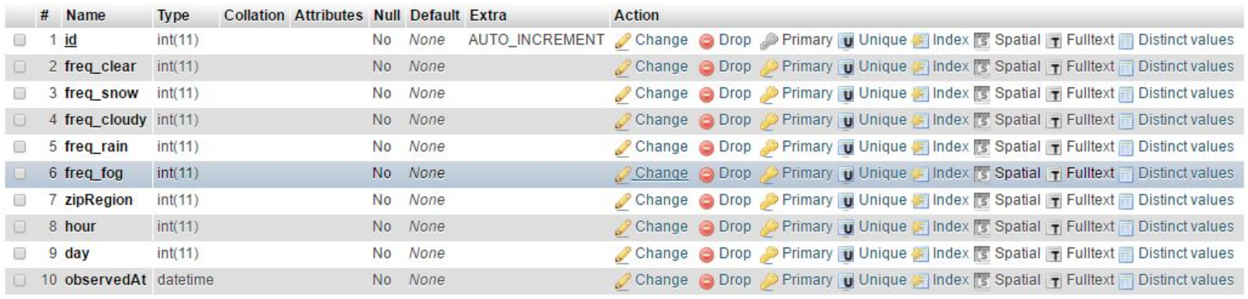
The severity table above is responsible for storing the sum and average of traffic severities for each road in a given Grid-Box (explained below) given a certain weather condition, hour, and day of the week. The purpose of having this information is to be able to calculate (including the null case previously mentioned in the mathematical model) the average traffic severity for a specific road under any weather condition, time of day, and day of the week. Additionally, each entry in this table is also assigned a gridId (explained in the next table), which specifies which segment of the road is being depicted.



#	Name	Type	Collation	Attributes	Null	Default	Extra	Action
1	id	int(11)			No	None	AUTO_INCREMENT	Change Drop Primary Unique Index Spatial Fulltext Distinct values
2	latN	float(10,6)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
3	latS	float(10,6)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
4	longE	float(10,6)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
5	longW	float(10,6)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
6	zipCode	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values

Figure 9: Grid-box Table

An interesting problem that arose as this table was forming was the fact that assigning a general severity to a whole road wouldn't accurately be able to depict where on the road traffic specifically occurs. To solve this problem, we decided to divide our entire "bounding box" into a grid of grid-boxes. Each grid-box is 0.085° in length and width, which corresponds to approximately 5x5 miles per box. We chose these dimensions in order for our grid-boxes to be granular enough to display precise data for specific segments of roads, but also to be large enough to on average cover the entire length of a single incident. Therefore, for each road in the severity table that has had a traffic incident occur on it, a grid-box is assigned based on the specific location of the incident (the incident that occurred is contained within the assigned grid-box). Each grid-box is assigned a zip code by using the latitude and longitude of the center of the box, then using a geocoding API to find the zip code of that coordinate point. By doing this, we now have the ability to sum total severities for a specific segment of a specific road at a certain day and time.



#	Name	Type	Collation	Attributes	Null	Default	Extra	Action
1	id	int(11)			No	None	AUTO_INCREMENT	Change Drop Primary Unique Index Spatial Fulltext Distinct values
2	freq_clear	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
3	freq_snow	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
4	freq_cloudy	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
5	freq_rain	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
6	freq_fog	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
7	zipRegion	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
8	hour	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
9	day	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
10	observedAt	datetime			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values

Figure 10: Frequency Table

The above frequency table is responsible for storing the frequency of all weather conditions for each grid-box for every hour, each day of the week. In order to query the Weather Underground API to get weather information, a zip code is needed. To make sure that the API query limit was not hit, we developed a strategy to efficiently collect weather. Based on an assumption that most neighboring zip codes will have similar weather, we decided to only collect weather once for a region of zip codes. For example, instead of checking the weather for 07853, 07854, and 07867, we simply collect the weather for 07853, then generalize that weather to all zip codes starting with ‘078’. This lets us collect weather data for a much larger range without exhausting our API key. Ultimately, for each hour of each day of the week, a running total of the weather conditions are stored in this table. By collecting data each hour instead of only when an incident occurs, we are able to include the null case in our average severity calculation.

Network Protocol

A HTTP request will be used to access “Google Maps” for the map images displayed on the website. The HTTP request was suggested because it is the most commonly used protocol and it is a standard protocol used in any browser space.

Global Control Flow

Execution order:

Our system is mostly procedure-driven. The navigation of the website as well as inputting information are the only 'asynchronous' events which occur. The system is mostly procedure-driven because each object does its job in a chronological order. That is to say that each object's job begins after the previous object's job ends. For example, once the TrafficCollector finishes querying the DB, the controller signals to the MapCommunicator to collect the map.

Time Dependency:

This traffic monitoring system will have parts that will operate in real time, and those that are not time sensitive. Many pieces of data are collected in real-time from API calls. Other pieces of information are calculated and stored ahead of time, so they are not time sensitive. The map will not dynamically update if the user changes the parameters. Instead, they will have to click "View HeatMap/Route" again, and the new image will be calculated with the new parameters.

It should be noted that there will be a small time delay between when the user request is entered and when the final processed images are displayed. We do, however, expect to return our map to the user within 2 minutes. The 2 minutes is an upperbound, not a target time.

Concurrency:

The system is not concurrent, and does not use multiple threads in the user interface or the data collection systems. We assume single threads for each system.

Hardware Requirements

Client:
-We recommend user to have any of the following:

- The current version of Microsoft Edge (Windows)
- Internet Explorer 10 and 11 (Windows)
- The current and previous version of Firefox (Windows, macOS, Linux)
- The current and previous version of Chrome (Windows, macOS, Linux)
- The current and previous version of Safari (macOS)

These are the web browsers supported by Google Maps, which is what will be used for the map images on the platform.

- Screen Resolution of at least 1024x768 is required
- Internet connection with minimum bandwidth of 56Kbps
- Javascript functionality must be enabled in the web browser

Server:

- Internet connection with minimum bandwidth of 56Kbps per concurrent client.
- 2 GB of free disc space per month for storing historical data
- The server must provide the following services: PHP, MYSQL, and Apache HTTP server.

Section 4: Algorithms and Data Structures

Algorithms

The algorithms for our database are very important to our project, and we explain the algorithms in great detail below. We believe that it should be distributed more than the 4 points currently allotted to this section.

There were many algorithms that needed be to implemented in order to realize the mathematical models previously described for our database collection. Recall that the mathematical model (equation 1) was designed to determine the average traffic severity of each road segment for a given time, day of week, and weather condition. Below is the mathematical model previously described:

$$A_{x,t} = \frac{\sum_{i=0}^{N} \tau(x,t)}{M} \qquad (1)$$

In this model, the total sum of severities for each road segment at a given time, day of week, and weather condition, was divided by M, the total frequency for the set of time, day, and weather conditions. For example, if the database shows 5 instances of snowy weather on a Tuesday at 5:00pm, and the total sum of traffic severities along a given stretch of US-206 that matches those conditions is 12, then the average severity is 12/5 = 2.4.

There were three different algorithms that needed to be implemented in order to realize this equation. (A) First, an algorithm was needed to collect the traffic data. (B) Second, an algorithm was needed to record the frequency of each weather condition for all the zip codes in our bounding box every hour. And finally, (C) there had to be an algorithm that sums the severities of each road segment and calculates the average severity for said road segment for each weather condition. Below are activity diagrams for A, B, and C, that will be used to describe each of these algorithms.

A. *Traffic Collection:*

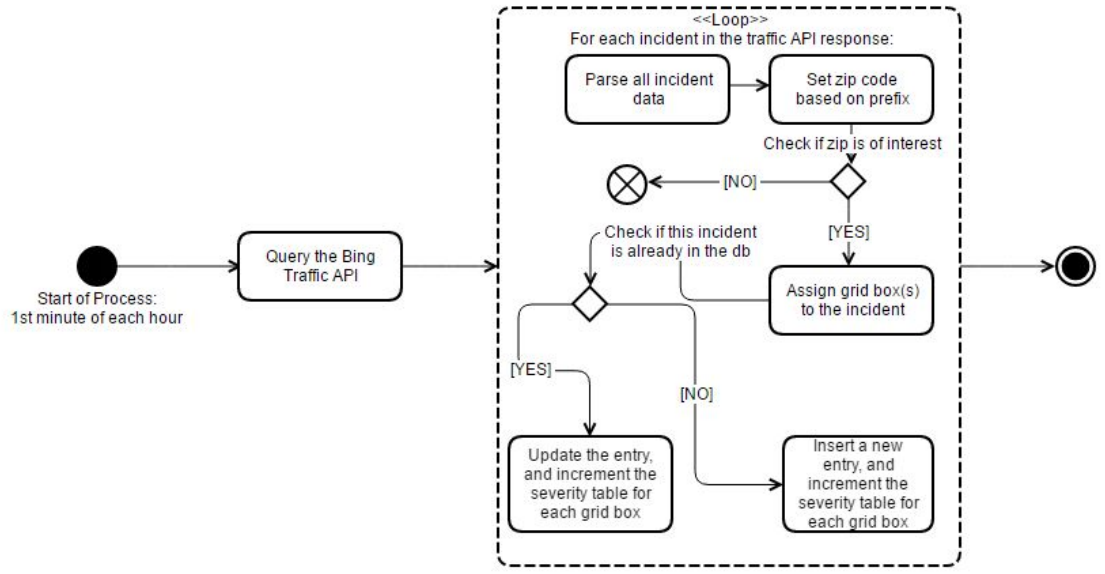

<<Loop>>
For each incident in the traffic API response:
Parse all incident data
Set zip code based on prefix
Check if zip is of interest
[NO]
[YES]
Start of Process: 1st minute of each hour
Query the Bing Traffic API
Check if this incident is already in the db
Assign grid box(s) to the incident
[YES]
[NO]
Update the entry, and increment the severity table for each grid box
Insert a new entry, and increment the severity table for each grid box

Figure 11: Traffic Collection Algorithm

In this algorithm, each traffic incident from the Bing Traffic API is parsed for its data. Then, if the zip code is within our service coverage area, we check and see if it's in the traffic collection database already. If it is, the incident info gets updated. If not, it gets inserted into the database. Additionally, for every hour the incident lasts, the severity of traffic for the specific road segment (contained in each grid box) is incremented for the current hour and day.

B. ***Weather Frequency Collection:***

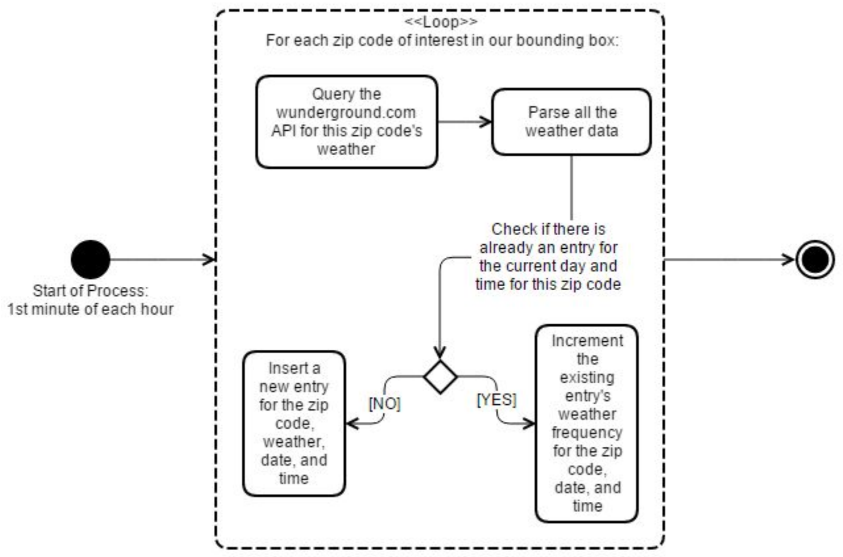

<<Loop>>
For each zip code of interest in our bounding box:
Query the wunderground.com API for this zip code's weather
Parse all the weather data
Check if there is already an entry for the current day and time for this zip code
Start of Process: 1st minute of each hour
[NO]
[YES]
Insert a new entry for the zip code, weather, date, and time
Increment the existing entry's weather frequency for the zip code, date, and time

Figure 12: Weather Collection Algorithm

In this algorithm, the weather of each zip code of interest for our bounding box is collected from the wunderground.com API. The weather data for each zip code is parsed and inserted into the weather frequency table if there currently does not exist an entry for the current day, hour, and zip code, or updates by incrementing the weather frequency if there is already an entry that matches this criteria. The wunderground API provides a multitude of weather conditions, but to simplify the number of conditions in our database we add a switch case that assigns any weather condition either to “Clear”, “Snow”, Cloudy”, “Rain” or “Fog”. This also benefits the user since these 5 conditions are the conditions the user can choose from, so that they are not overwhelmed with multiple overly similar conditions.

C. ***Total Severity Calculation:***

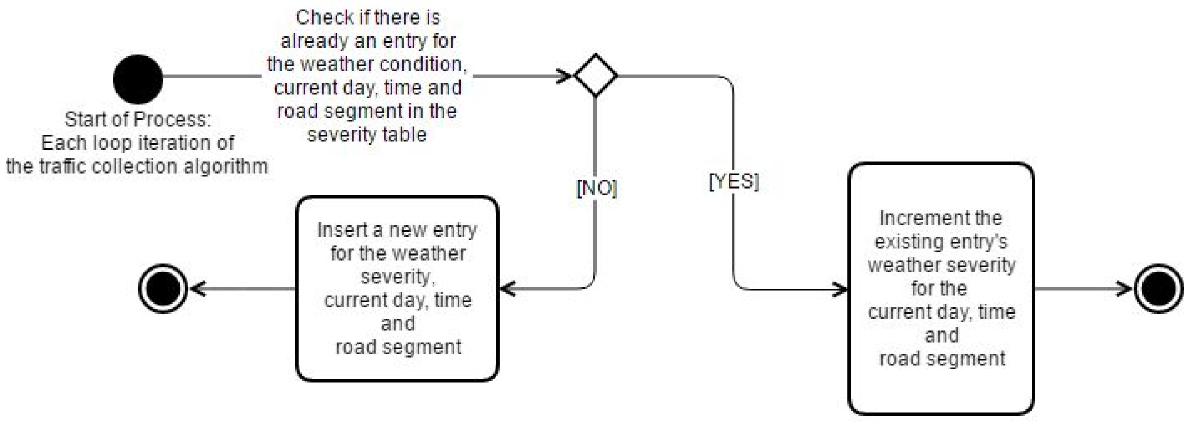


Figure 13: Total Severity Calculation Algorithm

For each incident in the traffic API response, this algorithm is run to increment the total severity for the road segment of the incident for the day and hour if there already exists an entry in the table, or inserts one.

Finally, after the severities and weather frequencies have all been updated/inserted for each road segment at the specific hour and day, the averages for each can be calculated simply using the equation (1). Moreover, in order to realize the persistent data storage described in the *Persistent Data Storage* section of this report, a cron job was enabled on our server that calls all these algorithms on the top of every hour to ensure that data is collected continuously.

In this section we explained the algorithms associated with our Data Tier. The Logic Tier and Presentation Tier do not have complex algorithms and therefore there is no description of algorithms associated with those tiers.

Data Structures

Our system does not rely on complex data structures. Most of our data needs are handled by our relational databases. Though many objects interface with API's, all XML or JSON responses are parsed into arrays before being returned, so as to make sure that formatting is consistent. This also helps abstract away the API interfacing from the logic tier.

Section 5: User Interface Design and Implementation

Compared to the original screen mock-ups from Report 1, the current user interface has not changed much. This is thanks to the fact that we carefully designed our user interfaces from the beginning keeping our use cases in mind at all times. There have been some minor changes that we will show and discuss below. The mock-ups have been moved into the implementation phase, as can be seen in the figures below. We show the parts of the user interface for UC-1, UC-2, and UC-6, that have been implemented and mention what is still on the backlog for development.

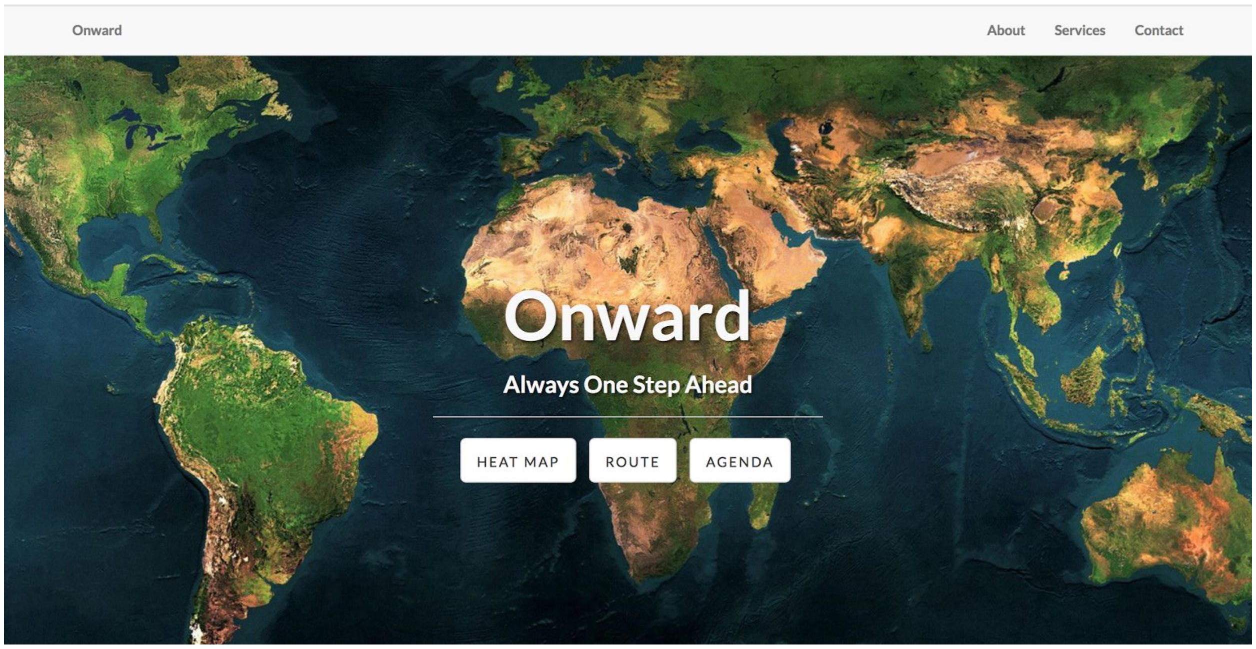
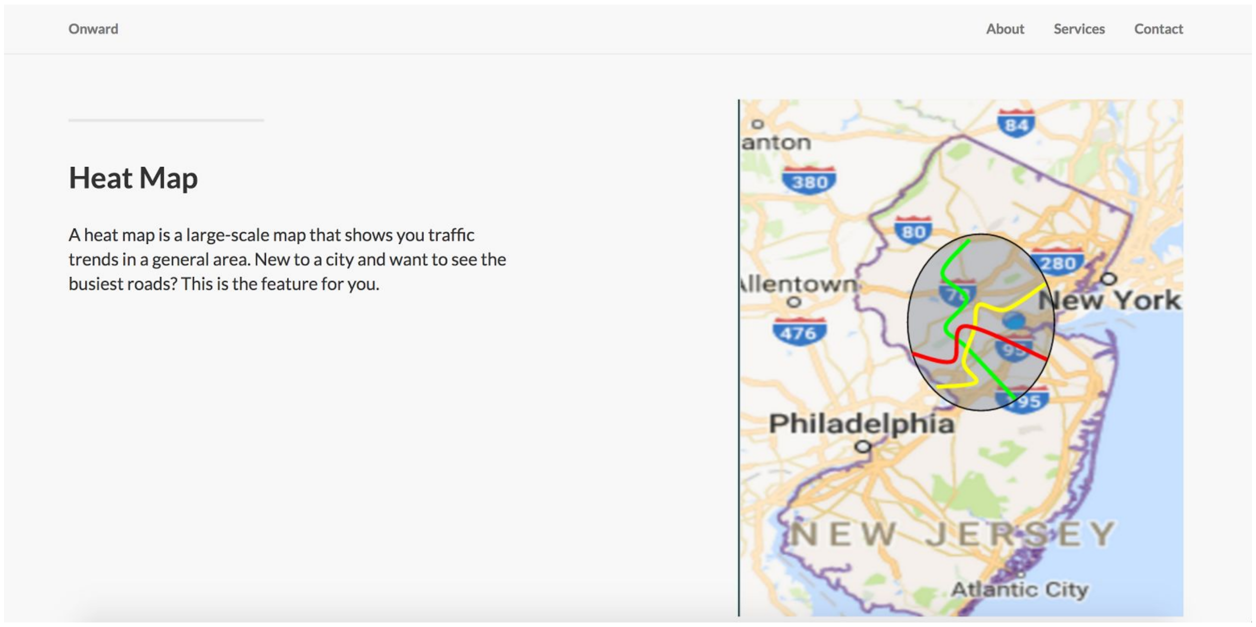


Figure 14: Landing Page (1)

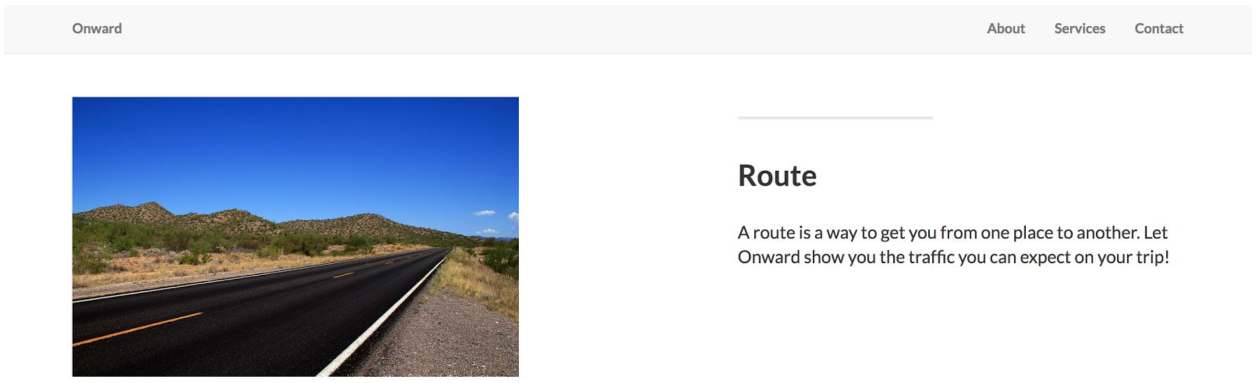
Figure 14 is our current Landing Page, where a user will be directed when they go to http://www.onwardtraffic.com/. There are some visual differences on the Landing Page, but it is functionally the same as the mockup. Instead of having the description of the features with the “feature buttons” Heat Map, Route and Agenda we have decided to place these feature descriptions below, as shown in Figures 15 and 16.



Heat Map

A heat map is a large-scale map that shows you traffic trends in a general area. New to a city and want to see the busiest roads? This is the feature for you.

Figure 15: Landing Page (2)



Route

A route is a way to get you from one place to another. Let Onward show you the traffic you can expect on your trip!

Figure 16: Landing Page (3)

On the same Landing Page, if the user scrolls down, they will see Figure 15 and Figure 16 consecutively. We decided to describe the features below the Figure 14 screen, as opposed to on it, for a few reasons. The first is that having the descriptions in the feature buttons would be cluttered and might overload a new user with new information. The other being that a returning user would not want to see the overview of the features every time they visit our site, but rather just select one of the features right away. This does not increase user effort, but rather declutters and better organizes our Landing Page.

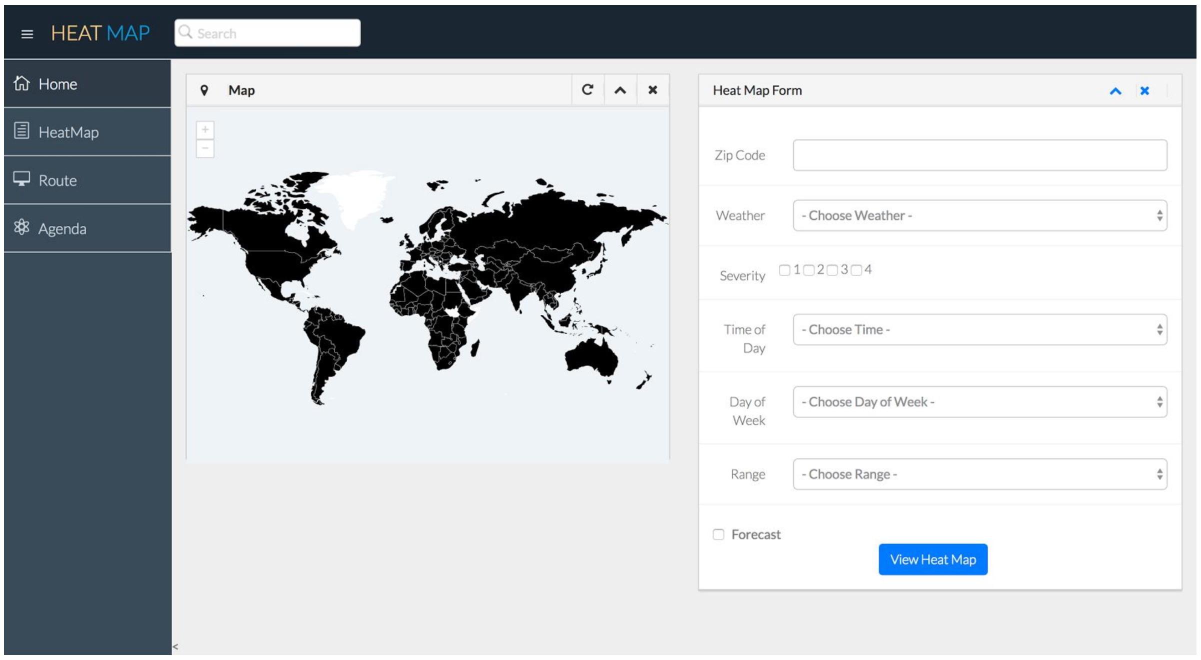


Figure 17: Heat Map Feature Page

Figure 17 shows the user interface of the Heat Map feature. The user would navigate to this page by clicking on the "Heat Map" button shown in the center of Figure 14. The search bar at the top of the pages will be removed from all pages during final implementation. The user can switch between features using the menu on the left.

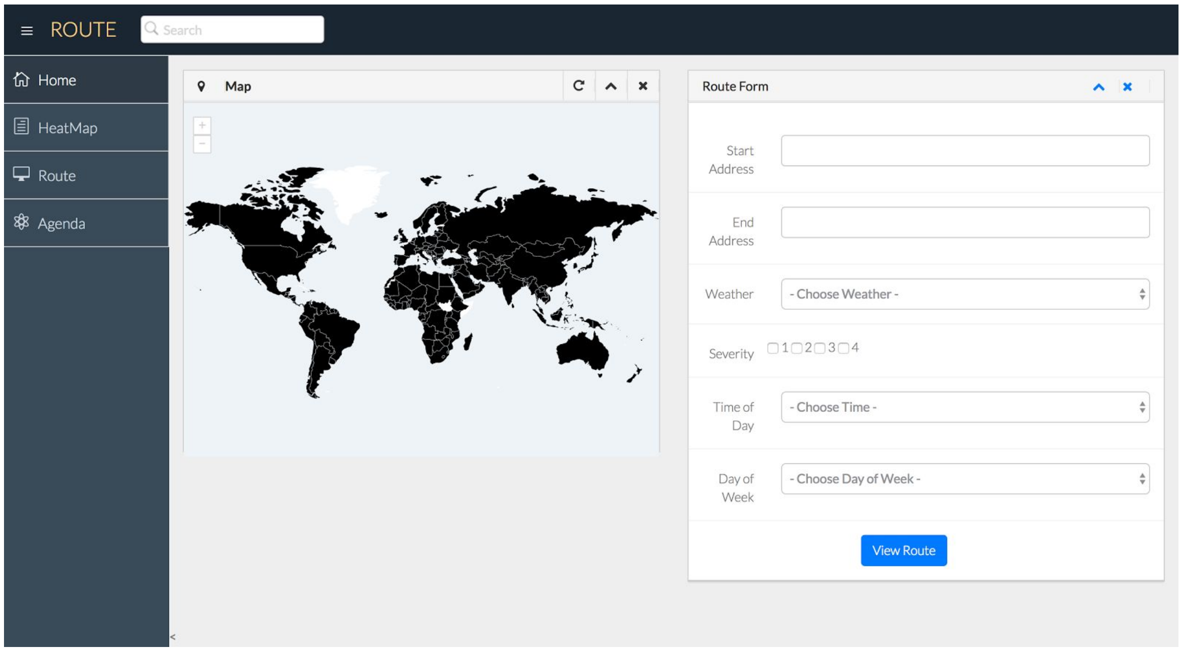


Figure 18: Route Feature Page

Figure 18 is the user interface of the Route Feature. The user can navigate to this page by clicking on the “Route” button shown in the center of Figure 14.

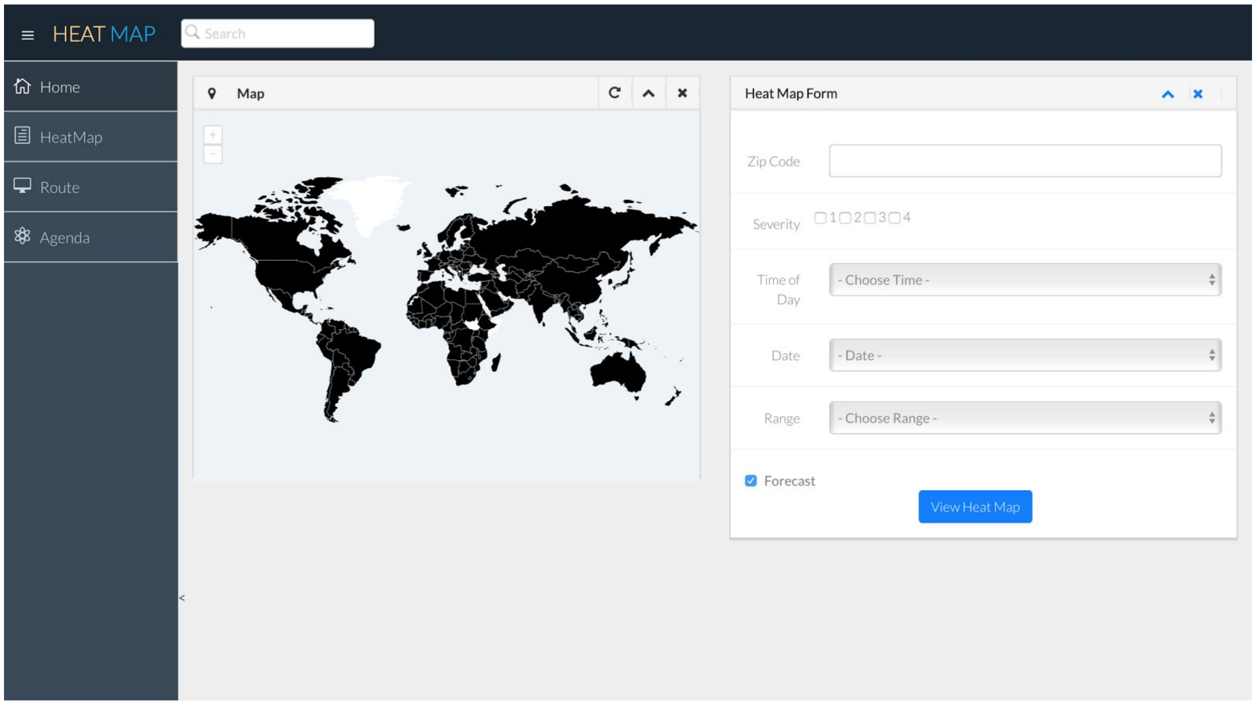


Figure 19: Forecasted Weather implemented on Heat Map

Figure 19 shows our user interface for when a user selects the checkbox “Forecast” at the bottom of the form in Figure 17. When that checkbox is clicked, either for Heat Map or Route, the form will dynamically change to that shown in Figure 19. The difference is that now the user no longer enters a type of weather manually, and instead of day of the week, they get a drop down to choose a “Date” that is in the next 10 days. Ten days was selected because it is the farthest date that the weather API can forecast.

To implement the web pages, Bootstrap[5] templates were used as a starting point, and modified to match our initial design. We used the template “Landing Page”[6] for the Landing Page and the “Nice Admin”[7] template for the other feature pages. A change made to the user interface is that there is a side menu where the user can navigate to other features as well as the home page. This side menu will be present on all pages and can be shown or hidden by pressing the hamburger button on the top left. The map that is shown in Figures 17, 18 and 19 is not the map that will be shown in final implementation, it is just a placeholder. We will be using a Google Maps GUI for our map interface. A change from the original mock-up design of the Heat Map feature is that the “Radius” specifier is now called “Range” and is implemented by a drop down instead of a slider. In addition, the choosing of day of the week is now a drop down menu rather than button selection. These changes were made to keep the form entries consistent and streamlined.

There will also be a “Data” section to the left of the map which will be implemented at a later point in time. This was included in the original mock-up of both the Heat Map and Route features, but has not yet been implemented. These data graphs will show the time on the x-axis (for that day) and the number of incidents reported on the y-axis. Incidents reported will be based on what we have collected in our database.

The user effort estimation has not drastically changed since the original estimation. The user simply navigates to the landing page, clicks on the feature they wish to use, fills out the information needed, and clicks the view button. The updated and detailed user effort estimations, with corrections for what was not included in Report 1, are shown below.

UC-1 (Assuming that the user has successfully navigated to the home page of the website)

1. Navigation: Total 2 mouse clicks as follows:
    a. Click “Heat Map”
    --*after completing data entry as shown below*--
    b. Click “View Heat Map”
2. Data Entry: Total 10 mouse clicks and up to 5 keystrokes as follows:
    a. Click cursor to “Zip Code/City” text field
    b. Press five keys which correspond to the zip code i.e. “08925”
    c. Click the dropdown button next to weather to display weather options
    d. Click the desired weather option i.e. “Snow”
    e. Click on the severity of the traffic incidents displayed i.e. “1”
    f. Click the dropdown menu next to Time
    g. Click the desired time of interest to display the Heat Map i.e. 2:00 PM
    h. Click the dropdown button next to Day of the Week to choose day.
    i. Click the desired day of the week from dropdown i.e “Sunday”
    j. Click the dropdown button next to Range to choose range
    k. Click the desired range in miles around city or zip code i.e 10

Note: If a user inputs a city, then depending on the city name (ie. the number of letters in the city’s name) the keystroke count will be different. Click count will be different if they choose to see multiple severities.

UC-2 (Assuming that the user has successfully navigated to the home page of the website)

1. Navigation: Total 2 mouse clicks as follows:
    a. Click “Route”
    --*after completing data entry as shown below*--
    b.Click “View Route”
2. Data Entry: Total 9 mouse clicks and 22 keystrokes as follows:
    a. Click cursor to “Start address” text field
    b. Press the keys which correspond to the address i.e. 101 Main St
    c. Click cursor to “End address” text field
    d. Press the keys which correspond to the address i.e. 500 Park Pl
    e. Click the dropdown button next to weather to display weather options
    f. Click the desired weather option i.e. “Snow”
    g. Click on the severity of the traffic incidents displayed i.e. “1”
    h. Click the dropdown menu next to Time
    i. Click the desired time of interest to display the Route i.e. 2:00 PM
    j. Click the dropdown button next to Day of the Week to choose day
    k. Click the desired day of the week from dropdown i.e “Sunday”

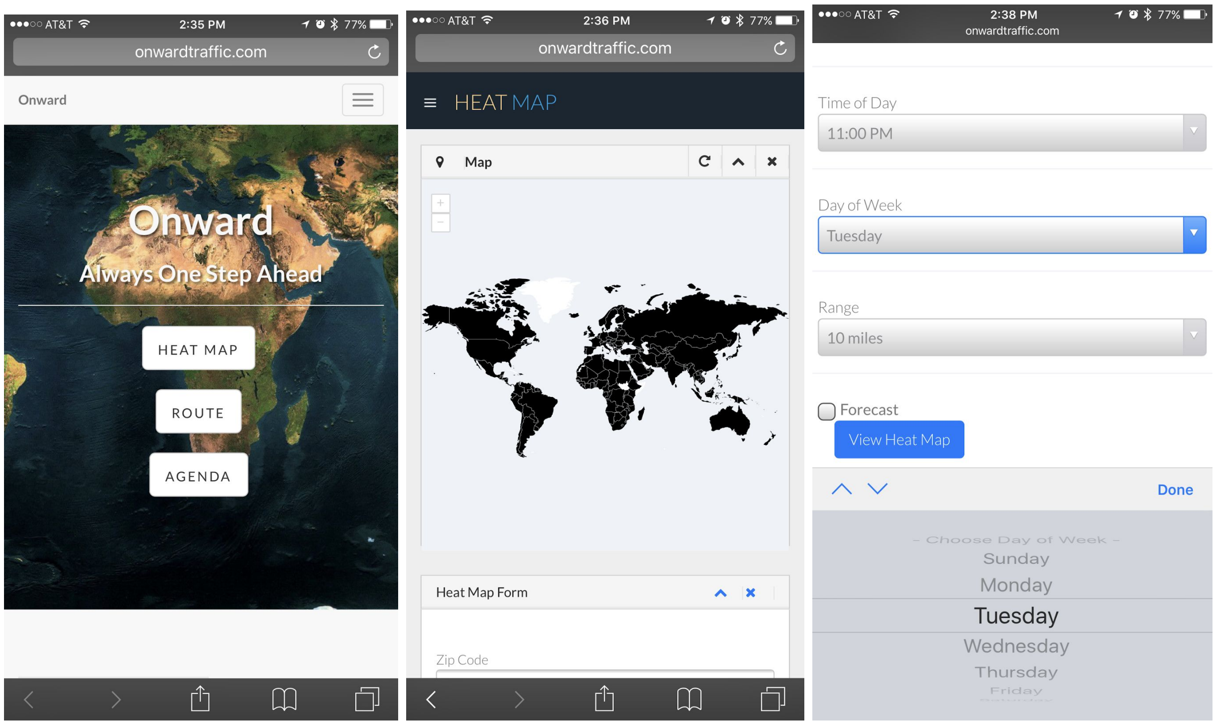
Note: Keystroke count depends on the address inputted. Click count will be different if they choose to see multiple severities.

UC-6 (Assuming that the user has successfully navigated to the home page of the website)

1. Navigation: Total 2 mouse clicks as follows:
    a. Click “Heat Map”
    --*after completing data entry as shown below*--
    b. Click “View Heat Map”
2. Data Entry: Total 9 mouse clicks and 5 keystrokes as follows:
    a. Click cursor to “Zip Code/City” text field
    b. Press five keys which correspond to the zip code i.e. “08925”
    c. Click on the severity of the traffic incidents displayed i.e. “1”
    d. Click on the “Forecast” Check Box
    e. Click the dropdown menu next to Time
    f. Click the desired time of interest to display the Heat Map i.e 2:00 PM
    g. Click the dropdown menu next to Date
    h. Click the desired date to display i.e. “Wed, April 5”
    i. Click the dropdown button next to Range to choose range
    j. Click the desired range in miles around city or zip code i.e 10

Note: For the purpose of illustrating the UC-6 we showed the usage with the UC-1, Heat Map. But it could also be used with the “Route” and “Agenda” features, UC-2 and UC-3 in a similar manner. If a user inputs a city, then depending on the city name the keystroke count will be different. Click count will be different if they choose to see multiple severities.

For the effort estimations, the worst case scenario is that the user would re-enter their zip code/city or addresses which would increase keystrokes.



Time of Day: 11:00 PM

Day of Week: Tuesday

Range: 10 miles

Figure 20: Mobile Interface

Figure 20 is some sample images showing our efforts to make our user interface mobile responsive, so people can use Onward on the go. As can be seen, the web pages are well formatted for mobile use.

Section 6: Design of Tests

We designed test cases that would reflect possible expected and unexpected behavior on the part of the user and participating actors. This is to ensure that our system can handle, for example, bad user-inputs or an HTTP request to an API returning an error.

We separated these test cases by the object that they apply to.

Unit Test Cases

InfoEntry:

Test-Case Identifier: TC-1

Function Tested: receiveInput(formInputs): bool

Pass/Fail Criteria: The test passes if the controller receives and saves the parameters.

Test Procedure	Expected Results
Call function with valid form entries. Function is called through the HTML form on the website. (Pass)	Controller receives the form entries and saves them for continued use.
Call function with invalid form entries. Function is called through the HTML form on the website. (Fail)	The function will notify the user which entry is incorrect and ask them to reenter.

Test-Case Identifier: TC-2

Function Tested: isValid(receivedInput): bool

Pass/Fail Criteria: The test passes if the receivedInput parameters are all valid.

Test Procedure	Expected Results
receiveInput() function calls this function with valid input. (Pass)	Function returns true and the receivedInput() function sends the form inputs to the controller.
receiveInput() function calls this function with invalid input. Invalid input could be a location outside of the bounding box we allow. (Fail)	Function returns false and the receivedInput() function notifies user of invalid entries, prompts for new entries.

ParamStorage:

Test-Case Identifier: TC-3

Function Tested: store(allParams): void

Pass/Fail Criteria: The test passes if the parameters are saved in the conditionParams and locationParams attributes.

Test Procedure	Expected Results
The controller calls the function and passes to it the parameters for desired location, weather, severity level, and time of day. (Pass)	The parameters for weather conditions and severity levels are stored in conditionParams. The parameters about the setting are stored in locationParams.
The controller calls the function and either passes too few or too little parameters. (Fail)	The parameters are not stored.

Test-Case Identifier: TC-4

Function Tested: getParams(String): Array

Pass/Fail Criteria: The test passes if the params are returned.

Test Procedure	Expected Results
After already having conditionParams and locationParams stored, call the function. (Pass)	The parameters are returned.
Without previously storing conditionParams and locationParams, call the function. (Fail)	The first index in the returned array will contain a flag to indicate an error.

ScreenDisplay:

Test-Case Identifier: TC-5

Function Tested: sendMap(mapImage): void

Pass/Fail Criteria: The test passes if the map image is received and ready to be displayed.

Test Procedure	Expected Results
Call function with a valid map image. (Pass)	An HTML element for a map with highlighted roads and markers is received and saved by the class. It is then ready to be displayed by the displayScreen() function.
Call function with invalid map image. (Fail)	The HTML element received is not a proper map and is not saved. The function exits without doing anything.

Test-Case Identifier: TC-6

Function Tested: sendGraph(graphImage): void

Pass/Fail Criteria: The test passes if the graph image is received and ready to be displayed.

Test Procedure	Expected Results
Call function with a valid graph image. (Pass)	An HTML element for a graph with proper statistics is received and saved by the class. It is then ready to be displayed by the displayScreen() function.
Call function with invalid map image. (Fail)	The HTML element received is not a proper graph and is not saved. The function exits without doing anything.

Test-Case Identifier: TC-7

Function Tested: displayScreen(): void

Pass/Fail Criteria: The test passes if the highlighted map and updated graph is displayed to the user.

Test Procedure	Expected Results
Call function with mapImage and graphImage attributes saved. (Pass)	A graph and map with highlighted roads and markers is displayed to the user.
Call function without mapImage and graphImage attributes preloaded. (Fail)	The function cannot display them and does nothing.

WeatherCollector:

Test-Case Identifier: TC-8

Function Tested: getForecast(allParams): Array

Pass/Fail Criteria: The test passes if the function returns a weather forecast for the time and day entered by the user.

Test Procedure	Expected Results
Call function with day and time within the 10-day forecast window. (Pass)	The function will call the callWeatherService() method to obtain a weather forecast for the date and time specified.
Call function with day and time outside the 10-day forecast window. (Fail)	The function will not be able to forecast the weather and will do nothing.

Test-Case Identifier: TC-9

Function Tested: callWeatherService(allParams): Array

Pass/Fail Criteria: The test passes if the function successfully calls the weather forecasting API and returns the forecasted weather to the getForecast() method.

Test Procedure	Expected Results
getForecast() method calls this function with valid input. API key has not passed its usage limit. (Pass)	Function returns the forecasted weather to the calling method.
getForecast() method calls this function with valid input. API key has passed its usage limit. (Fail)	API will indicate that no more calls are available. The weather will not be forecasted.

MapCommunicator:

Test-Case Identifier: TC-10

Function Tested: getRoute(location): Array

Pass/Fail Criteria: The test passes if a route to get from starting point to destination is returned.

Test Procedure	Expected Results
Call function with valid location for starting and ending points. (Pass)	A route to get from starting point to ending point is returned.
Call function with invalid location for starting and ending points. Invalid point could be a point on an ocean. (Fail)	The function will not be able to find a route and exit.

Test-Case Identifier: TC-11

Function Tested: routeQueryMapServ(location): Array

Pass/Fail Criteria: The test passes if the mapping API successfully returns a route to get from starting location to ending location.

Test Procedure	Expected Results
getRoute() method calls this function with valid input. API key has not passed its usage limit. (Pass)	Function returns a route to get from starting point to ending point to the calling method.
getRoute() method calls this function with valid input. API key has passed its usage limit. (Fail)	API will indicate that no more calls are available. The route will not be found.

Test-Case Identifier: TC-12

Function Tested: getMap(traffic): HTML Element

Pass/Fail Criteria: The test passes if an HTML element for a map with the traffic statistics highlighted on it is generated and returned.

Test Procedure	Expected Results
Call function with valid traffic information. (Pass)	ImageQueryMapServ() function will be called. That function will generate a map HTML element that shows traffic statistics and return it.
Call function with invalid traffic information. (Fail)	The function will not be able to generate a map with traffic information displayed on it.

Test-Case Identifier: TC-13

Function Tested: imageQueryMapServ(traffic): HTML Element

Pass/Fail Criteria: The test passes if the mapping API successfully returns an HTML element for the map.

Test Procedure	Expected Results
getMap() method calls this function with valid input. API key has not passed its usage limit. (Pass)	Function returns a map HTML element with traffic information overlayed on it to the calling method.
getMap() method calls this function with valid input. API key has passed its usage limit. (Fail)	API will indicate that no more calls are available. The map HTML element will not be generated.

TrafficCollector:

Test-Case Identifier: TC-14

Function Tested: getHeatTraff(location): Array

Pass/Fail Criteria: The test passes if traffic information for the desired location is returned.

Test Procedure	Expected Results
Call function with valid location. This function will call the method queryHeatDB() to get traffic information for that location. (Pass)	Traffic information for the desired location is obtained and returned.
Call function with invalid location. The invalid location could be a location outside of our project’s bounding box. (Fail)	The function will not be able to find traffic information and do nothing.

Test-Case Identifier: TC-15

Function Tested: getRouteTraff(route): Array

Pass/Fail Criteria: The test passes if traffic information for the desired route is returned.

Test Procedure	Expected Results
Call function with valid route. This function will call the method queryRouteDB() to get traffic information along that route. (Pass)	Traffic information along the desired route is obtained and returned.
Call function with invalid route. The invalid route could include streets outside of our project's bounding box. (Fail)	The function will not be able to find traffic information and do nothing.

Test-Case Identifier: TC-16

Function Tested: queryHeatDB(location, conditionParams): Array

Pass/Fail Criteria: The test passes if there is traffic information for the given location and conditions in the database and it is successfully queried.

Test Procedure	Expected Results
getHeatTraff() method calls this function with valid input. Database contains traffic information for this input. (Pass)	Function queries the traffic table database to obtain all traffic information for desired location and conditions. Returns information to calling method.
getHeatTraff() method calls this function with valid input. Database does not contain traffic information for this input. (Fail)	The database shows that no traffic incidents were reported for this location and condition, returns nothing.

Test-Case Identifier: TC-17

Function Tested: queryRouteDB(route, conditionParams): Array

Pass/Fail Criteria: The test passes if there is traffic information for the given route and conditions in the database and it is successfully queried.

Test Procedure	Expected Results
getRouteTraff() method calls this function with valid input. Database contains traffic information for this input. (Pass)	Function queries the traffic table database to obtain all traffic information for desired route and conditions. Returns information to calling method.
getRouteTraff() method calls this function with valid input. Database does not contain traffic information for this input. (Fail)	The database shows that no traffic incidents were reported for this route and condition, returns nothing.

LocConverter:

Test-Case Identifier: TC-18

Function Tested: getHeatLoc(locationParams): Array

Pass/Fail Criteria: The test passes if location params are successfully converted into latitude and longitude coordinates and returned to the caller.

Test Procedure	Expected Results
Call function with valid locationParams (zip code). This function will call the method callLocServHeat() to do the geocoding. (Pass)	A set of coordinates that correspond to the inputted location are returned to the caller.
Call function with invalid locationParams. (Fail)	The function will not be able to do the geocoding and do nothing.

Test-Case Identifier: TC-19

Function Tested: getRouteLoc(locationParams): Array

Pass/Fail Criteria: The test passes if locationParams are successfully converted into latitude and longitude coordinates and returned to the caller.

Test Procedure	Expected Results
Call function with valid locationParams (starting and ending address). This function will call the method callLocServRoute() to do the geocoding. (Pass)	Two sets of coordinates that correspond to the inputted starting and ending point are returned to the caller.
Call function with invalid locationParams. (Fail)	The function will not be able to do the geocoding and do nothing.

Test-Case Identifier: TC-20

Function Tested: callLocServHeat(locationParams): Array

Pass/Fail Criteria: The test passes if the function successfully converts location parameters for zip code into a set of coordinates using a geocoding API.

Test Procedure	Expected Results
getHeatLoc() method calls this function with valid input. API key has not passed its usage limit. (Pass)	Function uses geocoding API to convert the location into a set of coordinates and returns them to calling method.
getHeatLoc() method calls this function with valid input. API key has passed its usage limit. (Fail)	The function will not be able to do the geocoding and will do nothing.

Test-Case Identifier: TC-21

Function Tested: callLocServRoute(locationParams): Array

Pass/Fail Criteria: The test passes if the function successfully converts location parameters for starting and ending addresses into two sets of coordinates using a geocoding API.

Test Procedure	Expected Results
getRouteLoc() method calls this function with valid input. API key has not passed its usage limit. (Pass)	Function uses geocoding API to convert the starting and ending points into two sets of coordinates and returns them to calling method.
getRouteLoc() method calls this function with valid input. API key has passed its usage limit. (Fail)	The function will not be able to do the geocoding and will do nothing.

Controller:

Test-Case Identifier: TC-22

Function Tested: sendInput(userParams): void

Pass/Fail Criteria: The test passes if the controller successfully saves all the user's parameters.

Test Procedure	Expected Results
Call function with valid userParams. (Pass)	The params are saved in the controllers attributes and also sent to the ParamStorage class for storage.
Call function with invalid userParams. (Fail)	The function will not be able to save the params and will report an error.

Test Coverage

Although user interaction in Onward is limited, the system does not necessarily behave in a deterministic way. That is because our system frequently interacts with participating actors through the use of API calls and SQL queries. There are several reason why these interactions may fail, such as a server outage, or the expiration of an API key. For this reason, we decided to develop a state-based coverage which outlines all the possible transitions and events the system may encounter while attempting to process a user request. We initially didn't consider these failure edge cases as meaningful enough to include. After experiencing a server outage which cut off access to our SQL database, we ended up realizing that they were, in fact, important enough to consider. By testing to ensure that all transitions and states are encountered, we effectively cover all requirements of this project. Furthermore, by unit testing each function with good and bad inputs, we fully cover each individual module.

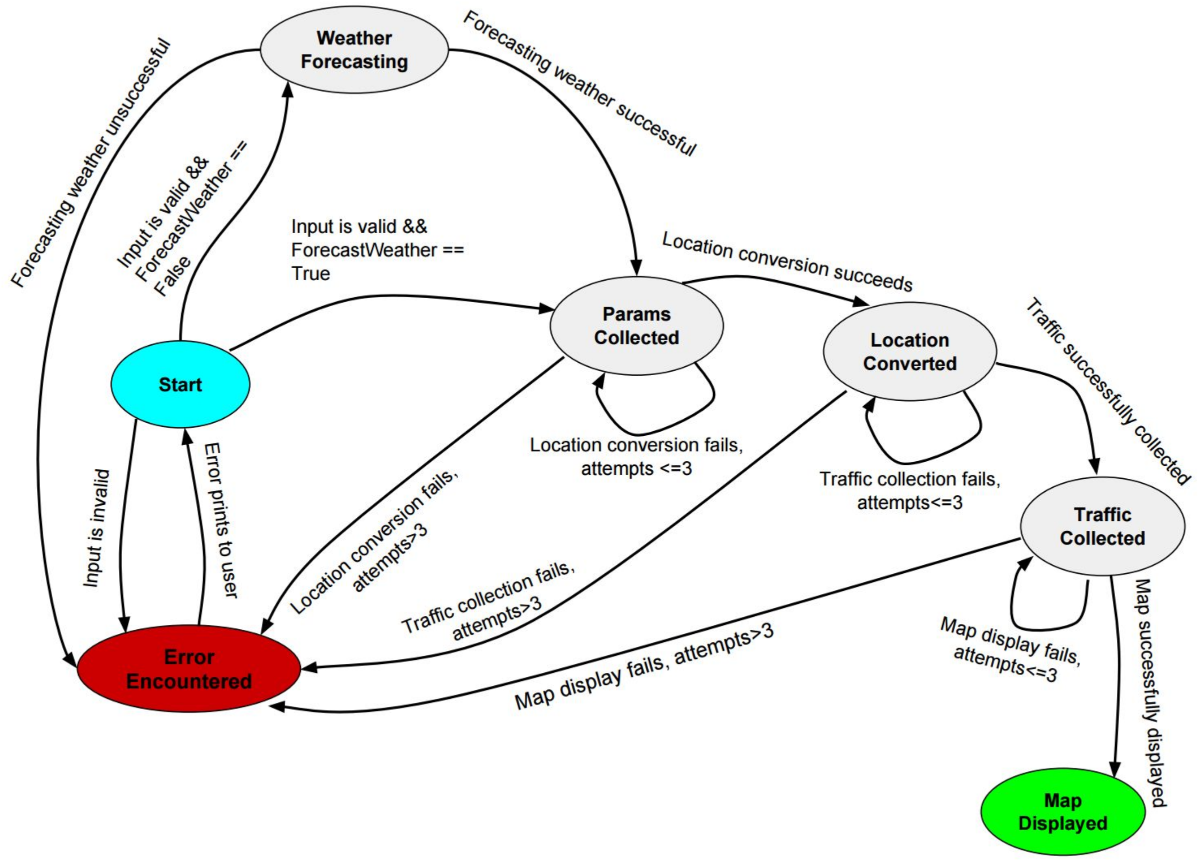

Weather Forecasting
Forecasting weather successful
Forecasting weather unsuccessful
Input is valid && ForecastWeather == False
Input is valid && ForecastWeather == True
Params Collected
Location conversion succeeds
Location Converted
Traffic successfully collected
Start
Location conversion fails, attempts <=3
Traffic collection fails, attempts<=3
Traffic Collected
Input is invalid
Error prints to user
Location conversion fails, attempts>3
Traffic collection fails, attempts>3
Map display fails, attempts<=3
Map successfully displayed
Error Encountered
Map display fails, attempts>3
Map Displayed

Figure 21: State Diagram for UC-1 and UC-6

Figure 21 shows the state diagram for UC-1 and UC-6. UC-6 is the weather forecasting use case. This is shown in the diagram in that if a user chooses to forecast weather, they must enter the “Weather Forecasting” stage before going to the “Params Collected” stage. The “Map Displayed” stage also includes displaying the graph of statistical data.

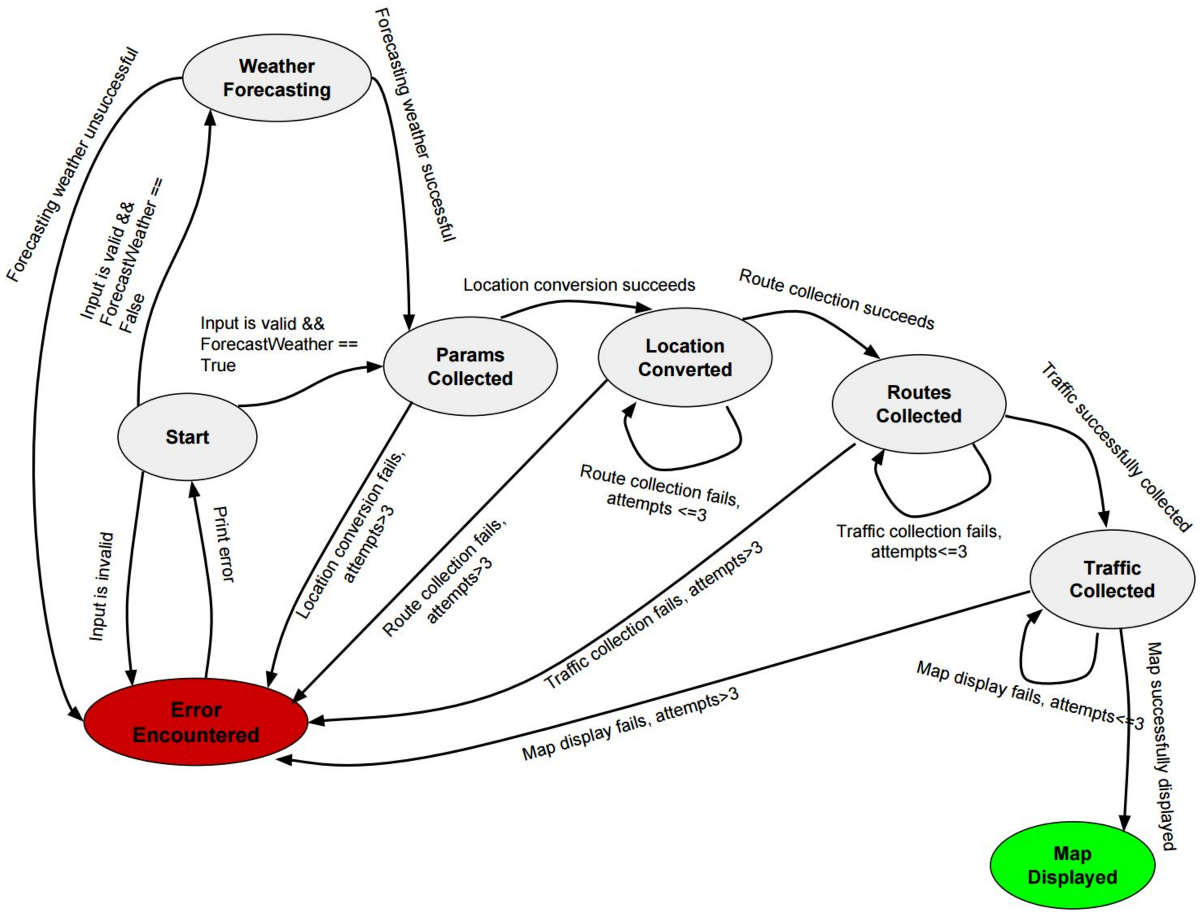

Weather Forecasting
Forecasting weather unsuccessful
Forecasting weather successful
Input is valid && ForecastWeather == False
Input is valid && ForecastWeather == True
Start
Params Collected
Location conversion succeeds
Location Converted
Route collection succeeds
Routes Collected
Traffic successfully collected
Traffic Collected
Route collection fails, attempts <=3
Traffic collection fails, attempts<=3
Input is invalid
Print error
Location conversion fails, attempts>3
Route collection fails, attempts>3
Traffic collection fails, attempts>3
Map display fails, attempts>3
Map display fails, attempts<=3
Map successfully displayed
Error Encountered
Map Displayed

Figure 22: State Diagram for UC-2 and UC-6

Figure 22 shows the state diagram for UC-2 and UC-6. UC-6 is the weather forecasting use case. This is shown in the diagram in that if a user chooses to forecast weather, they must enter the “Weather Forecasting” stage before going to the “Params Collected” stage. This diagram differs from Figure 21 in that this time the “Routes Collected” stage must be entered before going to “Traffic Collected”. Also note that that although in both figures you conclude with the “Map Displayed” stage, the actual map displayed varies by feature. The “Map Displayed” stage also includes displaying the graph of statistical data.

Integration Testing

After unit testing is complete, we will perform our integration testing based around the state-based diagrams above. We will make sure that every state is reached at least once, and that every transition, both valid and invalid, will be encountered at least once. This will ensure that our units can function together for both expected and unexpected behavior. This also ensures that our testing is systematic.

Acceptance Testing

Once integration testing is complete, we will move onto acceptance testing. The goal of our acceptance tests is more holistic, and is to basically ensure that the system performs as expected when put into the hands of a regular user. The user will then report whether or not the desired functionality was achieved. The user's feedback will also help us to determine if the non-functional and UI requirements have been met. The UI requirements are binary, such as "will display a map" (it either does display a map, or it does not), so it is fairly easy to determine if they have been met. In order to quantify if the non-functional requirements have been met, we will collect keystroke and mouse click data in order to determine final user effort. By comparing that against our estimation, we can determine whether or not the end user was able to easily use our software.

Additionally, these user acceptance tests will allow us to gather qualitative feedback from an end user on how our system can improve. This feedback could involve a desired UI change to improve ease-of-use, a suggestion for a future feature, etc. Our design process is iterative, so we would be able to incorporate feedback into future releases if so desired.

Section 7: Project Management

Merging the Contributions from Individual Team Members

Our team uses Google Docs to create our reports so everyone can simultaneously work on the same document and add their contributions. This minimizes the issue of collecting everyone's work and putting all the work into one document. Nevertheless, while collaborating on the report it sometimes was difficult to blend writing styles of all six group members. Often times several different styles of writing would conflict and interrupt the coherence of the report. In order to incorporate all of the ideas and work of the group, it was often more beneficial to let all members write their respective parts and then later work to edit the document so that it would smoothly transition from one response to the next.

Another problem that we encountered was keeping the naming and labeling conventions for diagrams consistent. We dealt with this issue by coming to a consensus about naming conventions. By doing this we were able to ensure consistency with how we communicated our ideas to those reading our report, as well as ensuring uniform formatting and appearance.
We are also using Gliffy[8] for our UML Diagrams, as this is an online based UML tool that all our team members can access and edit the diagrams accordingly.

Project Coordination and Progress Report

Currently the group is implementing UC-1,UC-2, and UC-6 and the conceptual objects associated with them. The database used for data collection has already been created and is functional in terms of properly collecting data for weather and traffic incidents along with calculating average severities. This database was planned out, developed and debugged by all members of our team. Setting up the database has taken up much of our development work time up until now, but we consider this okay because the database construction was one of the most complicated parts of our system. In addition, the basic user interface templates have been created. Some relevant project management activities include keeping a schedule that accounts for the report submission dates, the demo dates, and necessary implementation due dates. In addition we have scheduled weekly meetings as well as additional meetings as needed. This schedule allows for every group member to aware of the specific work that needs to be completed by a specific date so that each group member can plan his/her schedule and complete the work by the designated time.

Plan of Work

Our team is continuing to work according to the Gantt Chart below and will have the development and testing of our Heat Map, Route and Forecasted Weather features (UC-1, UC-2 and UC-6) by our demo which is now scheduled for March 28th. Further details can be seen in the Gantt Chart.

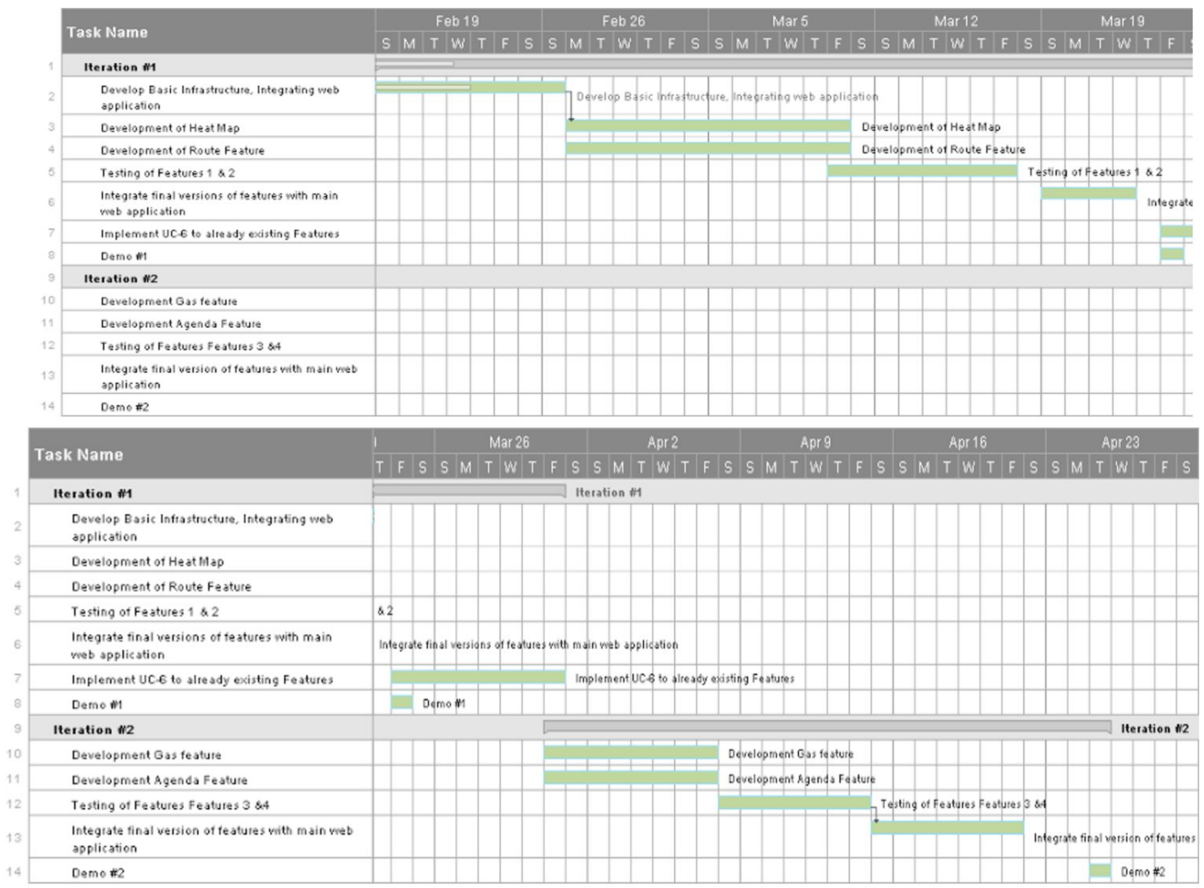


Figure 23: Gantt Chart

Breakdown of Responsibilities

Since our proposal and Report 1, we have developed a much better understanding of our project as well as how the development will proceed. We have created our conceptual objects and named their attributes and operations. We have decided that the development should now be oriented towards the development of the classes rather than feature specific. Since both of these features have similar implementations, and as a result have overlapping concepts, it makes the most sense to have team members develop each conceptual object such that it works with the Heat Map as well as Route feature. This methodology allows us to avoid many future integration conflicts and implementation issues. Additionally, it allows each team member to have ownership of all the features we are delivering, as opposed to just one.

Sean and Mhammed are working on the development of the Map Communicator. The Map Communicator is the most complex class in our project which is why it is the only object they are constructing. They will also be responsible for the unit testing of this class.

Shubhra and Lauren are working on the development of the InfoEntry, TrafficCollector and WeatherCollector classes of the project.

Brian and Ridwan are working on the development of the ParamStorage, LocConverter and MapDisplay classes of the project. They will also be responsible for the unit testing of these classes.

All members will contribute equally to the construction of the Controller class, which is effectively our “main” object. All members will contribute to the user interface and to the construction of the database.

Once these classes are functional and well tested, the next step will be the integration of these classes. Sean will coordinate the integration as he is the one who approves pull requests and merges branches. Here, the original sub teams will be responsible for the integration such that their respective features are functional. That is, Mhammed Brian and Sean will work on the integration of the classes for the Heat Map feature and similarly Lauren Shubhra and Sean will work on the integration of the classes for the Route feature. The Heat Map and Route feature will simultaneously be integrated with the Forecasted Weather Feature (UC-6).

Section 8: References

1. Bing Traffic API - *Bing Developer Network.* 2017. <https://msdn.microsoft.com/en-us/library/hh441725.aspx>
2. Current Weather Data - *Weather Underground API*. 2017. <https://www.wunderground.com/weather/api>
3. GeoCoding API - *Google. 2017.* <https://developers.google.com/maps/documentation/geocoding/start>
4. Cron Job - *Wikipedia. 2017.* <http://www.adminschoice.com/crontab-quick-reference>
5. Bootstrap - *Start Bootstrap. 2017.* <http://getbootstrap.com>
6. Bootstrap Landing Page - *Start Bootstrap. 2017.* <https://startbootstrap.com/template-overviews/landing-page>
7. Bootstrap Nice Admin Template- *Start Bootstrap. 2017.* <https://bootstraptaste.com/nice-admin-bootstrap-admin-html-template>
8. Gliffy - *Gliffy Inc. 2017.* <https://www.gliffy.com>
9. PHP graphing - *JPGraph*. 2017<http://jpgraph.net>

The use of these references are used throughout this report, and there is a superscript matching the number of the reference the first time any of these references are mentioned.