

## **Report 3: My Traffic Wizard**

Professor: Dr. Mike Mireku Kwakye  
CSCI441: Software Engineering

Group: Team A  
Apr 28, 2024

Repository URL: <https://github.com/fhsu-csci441-team-a/myTrafficWizard>

Render App URL\*: <https://mytrafficwizard.onrender.com>

*\*Note: Render App sleeps so may need time to load. Also, due to rate limits, scheduled trips are only processed once every hour*

### **Team A Group Members:**

Nicole-Rene Newcomb  
Tyler Anderson  
Philip Baldwin  
Jacob Spalding

### Individual Contributions Breakdown

We feel that all members have contributed evenly to the project. A more precise breakdown of contributions can be seen in the table below:

Sections	Nicole-Rene	Tyler	Philip	Jacob
Cover Page	x			
Individual Contributions Breakdown	x	x	x	x
Table of Contents	x			
Work Assignment	x	x	x	x
1a. Problem Statement	x			
1b. Decomposition into Sub-Problems	x			
1c. Glossary of Terms		x		
2a. Business Goals		x		
2b. Enumerated Functional Requirements	x	x	x	x
2c. Enumerated Nonfunctional Requirements				x
2d. User Interface Requirements			x	
3a. Stakeholders	x	x	x	x
3b. Actors and Goals	x	x	x	x
3c. Use Cases	x	x	x	x
3d. System Sequence Diagrams		x		
4a. Preliminary Design			x	
4b. User Effort Estimation			x	
5a. Identifying Subsystems		x		
5b. Architecture Styles			x	

5c. Mapping Subsystems to Hardware				X
5d. Connectors and Network Protocols				X
5e. Global Control Flow			X	
5f. Hardware Requirements				X
6. Project Size Estimation	X			
7a. Conceptual Model	X	X	X	X
7b. System Operation Contracts	X	X	X	X
7c. Data Model and Persistent Data Storage		X	X	
8. Interaction Diagrams		X		
9a. Class Diagram		X		
9b. Data Types and Operation Signatures	X	X	X	X
9c. Traceability Matrix	X		X	
10. Algorithms and Data Structures				X
11. User Interface Design and Implementation			X	
12a. Test Cases	X	X	X	X
12b. Test Coverage	X			
12c. Integration Testing	X			
Project Management and Plan of Work		X		
a. Merging the Contributions	X			
b. Project Coordination and Progress Report		X		
c. Plan of Work		X		
d. Breakdown of Responsibilities		X		
References	X			

Summary of Changes	X	X		
API Research	X	X	X	X
Web Host Research	X	X		
GitHub Task Management		X		
Creating SlackBotController, SlackBotModel, HomeController, NotificationController, Routes, and NotificationController and Slack interactive GUI tests	X			
Creating DiscordBotController, InputValidation, WeatherController, and WeatherModel				X
Creating User Interface Web Page	X		X	
Creating TravelController, TravelIncidentModel, TravelTimeMode, databaseConnection, ScheduledTripsModel, and BaseFetchRetry		X		
Creating AddressModel, MessageModel, and GmailController			X	
Creating Unit Tests and Integration Testing Suite		X		
Demo #1 Slides and Video Presentation	X	X	X	X
Demo #1 Technical Documentation	X	X		
9d. Design Patterns			X	
9e. Object Constraint Language (OCL) Contracts	X			
History of, Current, and Future Work	X			
Demo #2 Slides and Video Presentation	X	X	X	X
Demo #2 Technical Documentation & E-Archive	X	X		



## Table of Contents

<u>Section</u>	<u>Page #</u>
Cover Page.....	1
Individual Contributions Breakdown.....	2
Table of Contents.....	5
Summary of Changes.....	8
Work Assignment.....	10
<b>1. Customer Problem Statement.....</b>	<b>11</b>
a. Problem Statement.....	11
b. Decomposition into Sub-Problems.....	14
c. Glossary of Terms.....	15
<b>2. Goals, Requirements, and Analysis.....</b>	<b>16</b>
a. Business Goals.....	16
b. Enumerated Functional Requirements.....	17
c. Enumerated Nonfunctional Requirements.....	18
d. User Interface Requirements.....	20
<b>3. Use Cases.....</b>	<b>22</b>
a. Stakeholders.....	22
b. Actors and Goals.....	23
c. Use Cases.....	24
i. Casual Description.....	24
ii. Use Case Diagram.....	25
iii. Traceability Matrix.....	26
iv. Fully-Dressed Description.....	28
d. System Sequence Diagrams.....	32
<b>4. User Interface Specification.....</b>	<b>39</b>
a. Preliminary Design.....	39
b. User Effort Estimation.....	44
<b>5. System Architecture and System Design.....</b>	<b>46</b>
a. Identifying Subsystems.....	46

b.	Architecture Styles.....	48
c.	Mapping Subsystems to Hardware.....	49
d.	Connectors and Network Protocols.....	50
e.	Global Control Flow.....	51
f.	Hardware Requirements.....	52
<b>6.</b>	<b>Project Size Estimation Based On Use Case Points.....</b>	<b>53</b>
<b>7.</b>	<b>Analysis and Domain Modeling.....</b>	<b>57</b>
a.	Conceptual Model.....	57
i.	Concept Definitions.....	58
ii.	Association Definitions.....	59
iii.	Attribute Definitions.....	60
iv.	Traceability Matrix.....	62
b.	System Operation Contracts.....	65
c.	Data Model and Persistent Data Storage.....	67
<b>8.</b>	<b>Interaction Diagrams.....</b>	<b>68</b>
<b>9.</b>	<b>Class Diagram and Interface Specifications.....</b>	<b>76</b>
a.	Class Diagram.....	76
b.	Data Types and Operation Signatures.....	85
c.	Traceability Matrix.....	105
d.	Design Patterns.....	109
e.	Object Constraint Language (OCL) Contracts.....	111
<b>10.</b>	<b>Algorithms and Data Structures.....</b>	<b>122</b>
a.	Algorithms.....	122
b.	Data Structures.....	122
c.	Concurrency.....	122
<b>11.</b>	<b>User Interface Design and Implementation.....</b>	<b>123</b>
<b>12.</b>	<b>Design of Tests.....</b>	<b>125</b>
a.	Test Cases.....	125
b.	Test Coverage.....	134
c.	Integration Testing.....	135
d.	System Testing.....	136

Project Management and History of Work.....	137
a. Merging the Contributions from Individual Team Members.....	137
b. Project Coordination and Progress Report.....	138
c. History of Work.....	140
d. Breakdown of Responsibilities.....	143
e. Current Status.....	145
f. Future Work.....	146
References.....	147

## Summary of Changes (Updated)

- Removed REQ14 (the system should display a map of your trip) and REQ24 (a user should be able to see a map of their route) as they were not being mapped to a use case for current work. These can be implemented as part of future work, along with REQ10 and REQ11, which weren't able to be completed within the project timeline.
- Expanded plan of work to include table with detailed entries and switched from using a Kanban board to a Gantt chart to increase readability of project timeline.
- Added to the flow of events for the main success scenario of use case 7: "4. Database updated to indicate trip notification sent".
- Changed name of trafficIncidentModel to travelIncidentModel and trafficController to travelController to improve naming consistency; all references to former names in images and text references have been updated.
- Added new subsystem components: scheduleController and messageModel, geocodeModel, and locationModel.
- Removed subsystem components: geocodeConverter.
- Updated Attribute Definitions Table to include homeController, remove notificationStatus from scheduleController, and remove userID and formattedMessage from slackBotController.
- Removed locationModel and geocodeModel from the system; the responsibility of providing a geocode has been assigned to the addressModel.
- Updated TC-7 to test getUpcoming trips instead of getTripsByDateRange, the latter has been deprecated.

- Update TC-11/TC-12, instead of throwing an exception, an error message is returned for testing failed procedures.
- Add test procedure on TC-10 which will test the scenario in which no incidents have been reported on the route.
- Update the return type for travelController's getTravelMessage from string to object
- Updated PF to be equal to 28 instead of 30, based on Report 3 instructions
- Removed checkAddress function in class InputValidation.
- Remove externalScheduler and inputValidation from all class diagrams and testing suites as they were not implemented as classes
- Remove test cases for scheduleController (TC-9), the scheduleController will not be passed a timestamp, instead it will create this internally in the class instance
- Removed homeController ↔ addressModel association as the addressModel is now called directly from the user interface
- Updated the UI and Effort Estimation Updates details
- Added updated user interface screenshots for desktop and mobile versions
- Added updates to various sections denoting changes to work assignments, work history, etc.

## **Work Assignment** (Updated)

Nicole-Rene and Jacob have been assigned to the chatbot integrations. Philip and Tyler are working on the user interface and notification integrations. All members have worked to select appropriate APIs to achieve optimum functionality of the project and added to the report.

**\*\*Update:** All tasks were completed successfully, with all members helping each other as needed.

### **Individual Student Competences**

**Nicole-Rene Newcomb:** Strong collaborator, brainstorming facilitator, and troubleshooter with moderate project management and organization skills. Primary programming languages include: Python, Java, C++, JavaScript, SQL, HTML, and CSS. Familiar with Kotlin, as well as WordPress and Android App development.

**Tyler Anderson (Team Leader):** Skill set includes Python, T-SQL, Java, and Azure Data services, complemented by a solid foundation in data modeling, database design, and data integration (APIs). Proficient in technical writing - documenting and communicating complex technical concepts via diagrams and text.

**Philip Baldwin:** In my current role, I field daily questions from customers about errors in the reports and services we provide. To answer, I must interpret gps and location data for vehicles, and cross-reference it against map data to determine the cause. Some of the technical knowledge I must employ on a daily basis includes the use of SQL, direct data retrieval via API calls, and virtual private server (VPS) administration. As a senior at FHSU, I have made use of programming languages (C++, Java, Python) as well as web development technologies (HTML, CSS, Javascript) to complete projects.

**Jacob Spalding:** Before going to FHSU I got a trade school degree in computers and networking. I work at the University of Kansas Health System as a PACS Admin. In my current role I support over 135 workstations for the Radiologists at the Hospital, satellite locations, and in their homes. At Fort Hays State University, I have used C++, JavaScript, Python, SQL, HTML and Java.

## **1. Customer Problem Statement**

### **1a. Problem Statement**

As drivers get used to driving familiar routes, they are less likely to use web mapping apps to map their routes every trip. This can cause drivers to be unaware of emerging incidents that could impact their travel. Unexpected weather and traffic conditions can result in a driver's standard commute taking much longer than anticipated. When this situation occurs, there can be several negative outcomes that tend to arise.

Driving at unsafe speeds for the conditions is one major risk that drivers may take when they realize they may not arrive at their destinations on time. Wet, snowy, and icy conditions can increase chances of accidents when speeds are not markedly decreased, so pressure to drive too fast can put drivers in danger. Drivers may also face negative consequences due to arriving at work, school, or other destinations late. Company drivers may also risk their business' reputation when transports or shipments are delayed.

Using web mapping apps for live directions can streamline navigation, but constantly entering trip details to account for changing conditions or to ensure timely arrivals can be cumbersome for drivers. Many web mapping apps also assume that drivers intend to start their trip immediately and can cause distraction due to verbal commands repeating while expecting the trip to start. Using these apps also results in high device battery consumption due to the use of GPS. These drawbacks indicate that a more convenient, automated notification system that offers multiple channels (email, SMS, Discord, Slack) could be a valuable solution to route planning.

To better understand how these struggles can impact drivers, some potential customer stories are provided below:

#### **Mary**

Mary is a hard-working account executive. She lives on the rural outskirts of a suburban area and commutes into the city center every day for work. In ideal conditions, her commute takes about 65 minutes, but traffic accidents often cause her commute to be closer to 80 minutes.

She wishes she could live closer to work, but she feels that she is better able to afford a home with sufficient backyard space for her kids in her current neighborhood.

Unfortunately, the start of her route has some low-lying dirt roads that are prone to flooding in rainy conditions and developing a thick ice coating in the winter. Since Mary doesn't have 4-wheel drive to avoid getting bogged down in the mud or avoid skidding out of control, unsafe conditions in this area means she has to go along a much longer alternate route that adds an extra 20 minutes to her commute when the roads are flooded or iced over. There are also two bridges along her route, and traffic accidents on them can severely delay traffic in that area.

Although Mary tries to pay attention to the evening weather reports, she is often distracted between her hectic job requiring a lot of take-home projects and catching up with her dearly beloved family. She has three children under the age of six who are watched by her parents during the day. After a hard day at work, she doesn't want to miss out on what little time she has to spend with them in the evenings before their bedtime. At the end of the day, she is usually exhausted by all her hard work and spending time with her rowdy youngsters. This makes it easy for her to miss the weather forecast segment of the local evening news.

She is also chronically underslept and has very little time available in the morning to check the news for possible traffic alerts too. Hold-ups on the bridges aren't known until an incident occurs, so there isn't a way to prepare for this the evening before. If a bridge delay isn't reported on the morning news during the brief time she's watching or she misses the announcement, then she won't be able to take an alternative route around the bridge. This detour adds an extra 12m to her drive, but it is preferable to the time delays that can occur from incidents on the bridge.

Mary is eager to find a way to make her commute safer that won't take so much time. Her current method of trying to pay attention to the local news reports while spending time with her children often results in missing traffic alerts. In addition, while the local news sometimes covers the roads on the outskirts of town, they tend to focus on the major city roads with heavier traffic. If Mary always planned to take the longer route whenever the reporters mentioned a likelihood of rain or freezing conditions, she would often end up wasting an extra 20 minutes of her time unnecessarily. Mary wants to drive safely, but she also needs a quicker, more accurate way to plan her trips to avoid wasting her valuable time.



## **Global Logistics**

Global Logistics LLC is a local parcel and freight company that promises its customers on-time delivery at a fair price no matter the weather. Global Logistics is strongly dedicated to doing its best to always deliver on time. However, dynamic traffic conditions and weather often result in unexpected delays, especially during times when the company is managing a heavy backlog of pending deliveries. The Christmas and end-of-year holiday season tends to be the most difficult time, as the volume of deliveries is extremely high right when weather conditions can make roads the most hazardous.

The company employs dispatching agents who collaborate with the logistics control center to plan and map out the next day's delivery routes the evening prior to the delivery. Since demands can fluctuate day to day, the exact delivery routes change frequently. While the experienced agents eventually become familiar with certain areas that may be prone to hazards, it is still hard for them to foresee every potential area, especially along less common routes, that may have a higher risk of traffic delays.

As Global Logistics is entering an expansion phase, they have recently purchased more trucks and vans and are in the process of training new personnel. The newer dispatching agents have been finding it very difficult to anticipate where traffic delays due to bad weather or other incidents may occur. This has resulted in more deliveries arriving late and a higher rate of customer complaints.

Global Logistics is eager to find a way to help both new and experienced dispatch agents plan better daily routes to avoid delivery delays and poor customer satisfaction ratings. They are hoping to minimize the overhead costs of needing to assign a dedicated employee to monitor the local news for the weather and traffic conditions. The company is also determined to avoid burdening their already busy drivers with needing to recheck their assigned route directions frequently while making deliveries, as they want their drivers to drive safely and focus on the deliveries assigned to them. Global Logistics hopes to find a simple, automated way to highlight potential delays. When drivers need to be alerted, they want notifications to be sent in a way that is convenient for the drivers.

## **1b. Decomposition into Sub-Problems**

### **Individual Customers Like Mary:**

1. Customers want to avoid being late to their trip destinations.
2. Customers want to know about potential traffic delays ASAP to adjust their departure times.
3. Customers want to minimize the time spent checking their routes prior to departure.
4. Customers want to automate the process of receiving trip traffic and condition updates.
5. Customers want to be alerted to travel updates through a convenient channel.

### **Commercial Customers Like Global Logistics:**

1. Customers want to avoid traffic delays that result in late deliveries.
2. Customers want to know about potential traffic delays ASAP to alter driver routes.
3. Customers want to minimize the work required to plan and update driver routes.
4. Customers want to automate the process of receiving route traffic and condition updates.
5. Customers want their drivers notified through a channel convenient for their drivers.

## 1c. Glossary of Terms

**Administrator:** An individual with the authority and responsibility to configure and manage a system.

**API (Application Programming Interface):** An API is a set of rules that enable different applications to communicate with each other. In this system's context, an API is used for data transfer and to utilize the functionality of external third-party systems.

**Application:** A software package that performs a specific function for an end user or another application.

**Chatbot:** A computer program that simulates human conversation.

**Cronjob:** A task automated using cron, a job scheduler on Unix-like operating systems.

**Database:** An organized collection of structured information, or data, stored electronically in a computer system. It is usually accessed and controlled via a database management system (DBMS).

**Developer:** An individual that creates software using code.

**Geocode:** A set of latitude and longitude coordinates that represent a specific geographical place such as a landmark, street address, or location.

**Graphical User Interface (GUI):** An interface that allows users to interact with a computer or electronic device.

**Traffic API:** An API that allows users to request and receive data for traffic incidents and travel times. The request can be customized with parameters such as date and location.

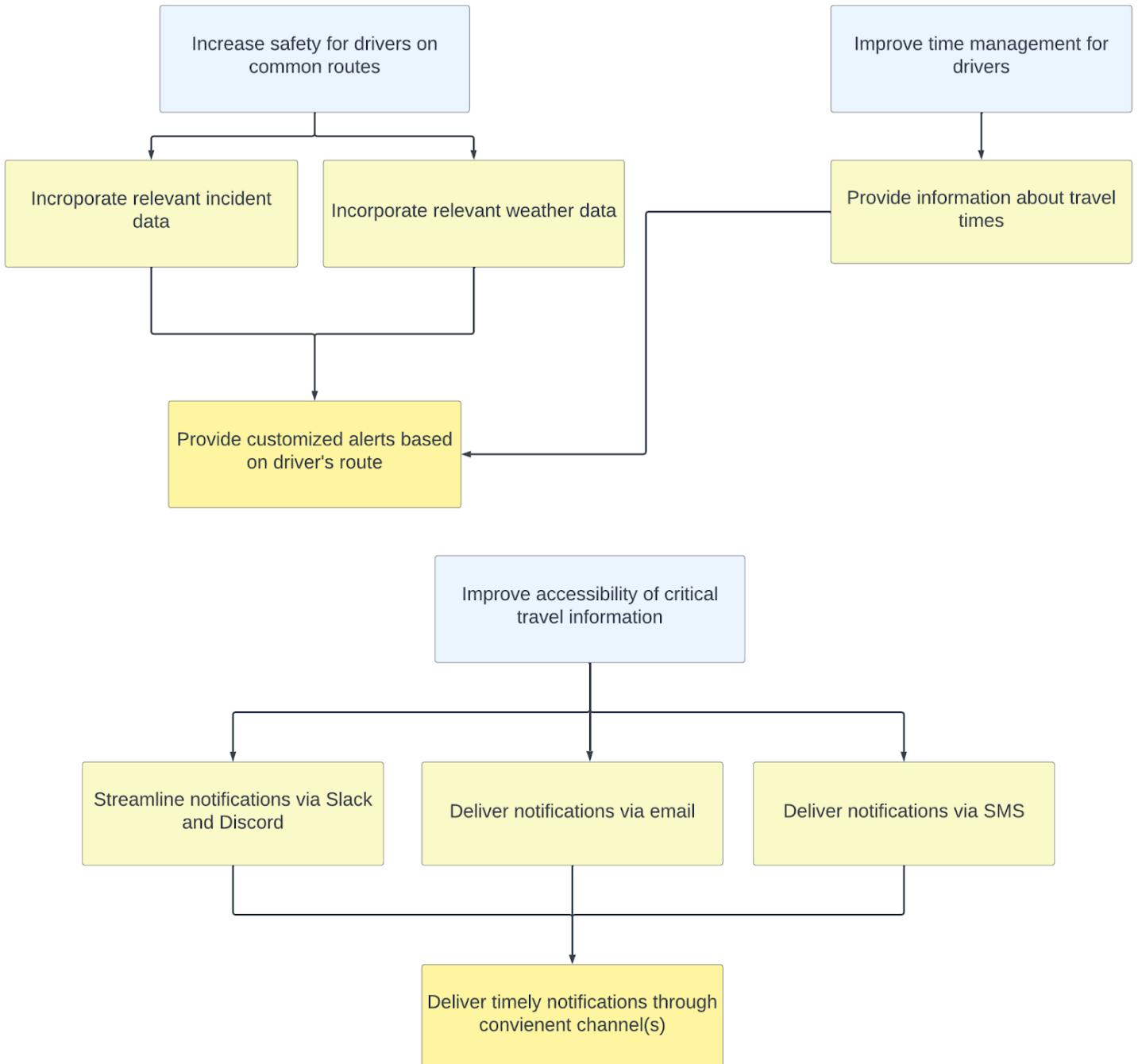
**Weather API:** An API that allows users to request and receive data for weather. The request can be customized with parameters such as location.

**Web App Host:** A platform that provides the resources and services necessary for running and managing web applications.

**User:** An individual that interacts with or utilizes a systems resources and services

## 2. Goals, Requirements, and Analysis (Updated)

### 2a. Business Goals



## 2b. Enumerated Functional Requirements

Identifier	PW	Requirement
REQ1	5	The system shall provide an interface for the creation of alerts accepting the following parameters: origin and destination addresses, preferred channels of notification, e-mail address, telephone number, route nickname, and recurrence. *Update: route nickname and recurrence options are no longer in the current project timeline and are part of future work.
REQ2	5	The system shall process requests for alerts and ensure their execution at scheduled times through the chosen communication channels.
REQ3	5	The system shall facilitate the transmission of notifications across multiple potential channels, according to the user's requests.
REQ4	5	The system shall transmit notifications through e-mail and Short Message Service (SMS), if requested by the user.
REQ5	3	The system should send notifications through Discord, if requested by the user.
REQ6	2	The system should send notifications through Slack, if requested by the user.
REQ7	5	The system shall send a notification to the user that includes estimated time, ideal time, travel delay time and any applicable incident information for their respective route.
REQ8	4	The system shall incorporate and leverage weather and traffic data captured at the time the message is sent for the chosen route.

**2c. Enumerated Nonfunctional Requirements** (Updated)

<b>Identifier</b>	<b>PW</b>	<b>Requirement</b>
REQ9	5	The home page shall have a fully functional and easy to use interface.
*REQ10	3	The system should provide preferences for what time the user will receive the notification.
*REQ11	5	The system shall reliably send notifications on time when the user sets up recurring notifications.
REQ12	5	The system shall allow users to choose a notification channel.
REQ13	5	The system shall have clearly labeled input boxes for user input.
*REQ14	2	The system should display a map of your trip.
REQ15	2	The system should work on most platforms.
REQ16	3	The system should provide incident details and estimated trip time for your chosen area.
REQ17	2	The system should provide weather information that may impact your planned route.
REQ18	3	The system should store user data for internal use only and will not share it with any 3 <sup>rd</sup> parties.

\* These requirements have been removed from consideration for the current project timeline and should be considered as potential future work.

**FURPS Table:**

<b>Functionality</b>	<ul style="list-style-type: none"><li>· Features a homepage that will be easy to use for our End Users.</li><li>· The home page will have input fields for starting point such as time, address, city, state, and zip code.</li><li>· The home page will have input fields for destination address, city, state, and zip code.</li><li>· The home page will give the user a choice of how they receive their notifications: SMS, E-mail, discord, slack or you can choose all of them.</li></ul>
<b>Usability</b>	<ul style="list-style-type: none"><li>· The home page will be laid out in a simple and easy to use way.</li><li>· The input fields will only accept valid inputs from the user.</li><li>· All input fields will be clearly labeled.</li></ul>
<b>Reliability</b>	<ul style="list-style-type: none"><li>· The system will have no issues within the software program itself.</li><li>· The only downtime that may occur would be from one of the systems that are out of our control.</li></ul>
<b>Performance</b>	<ul style="list-style-type: none"><li>· The system will process data and display information in a timely manner.</li><li>· The database will have enough space to hold all the information needed</li></ul>
<b>Supportability</b>	<ul style="list-style-type: none"><li>· The home page is supported on multiple browsers.</li><li>· The system will support multiple notification types: SMS, E-mail, Discord, and Slack</li></ul>

**2d. User Interface Requirements** (Updated)

Identifier	Priority	Requirement
REQ19	5	<p>A user shall be able to enter a departure date and time.</p> <div data-bbox="591 443 1040 590" style="border: 1px solid black; padding: 5px;"> <p align="center"><b>Departure Date and Time</b></p> <p align="center">02/02/2024 05:51 PM <input type="button" value="🗳"/></p> </div>
REQ20	5	<p>A user shall be able to enter a departure location.</p> <div data-bbox="591 667 1175 1230" style="border: 1px solid black; padding: 10px;"> <p align="center"><b>Departure</b></p> <p align="center">Street Address</p> <p align="center"><input type="text"/></p> <p align="center">Apt., Suite, Etc. (Optional)</p> <p align="center"><input type="text"/></p> <p align="center">City</p> <p align="center"><input type="text"/></p> <p align="center">State</p> <p align="center">KS <input type="button" value="▼"/></p> <p align="center">Zip Code (Optional)</p> <p align="center"><input type="text"/></p> </div>
REQ21	5	<p>A user shall be able to enter an arrival location.</p> <div data-bbox="591 1308 1175 1881" style="border: 1px solid black; padding: 10px;"> <p align="center"><b>Destination</b></p> <p align="center">Street Address</p> <p align="center"><input type="text"/></p> <p align="center">Apt., Suite, Etc. (Optional)</p> <p align="center"><input type="text"/></p> <p align="center">City</p> <p align="center"><input type="text"/></p> <p align="center">State</p> <p align="center">KS <input type="button" value="▼"/></p> <p align="center">Zip Code (Optional)</p> <p align="center"><input type="text"/></p> </div>



REQ22	5	<p>A user shall be able to submit a notification request.</p> <div style="text-align: center;"> <input type="button" value="Submit"/> </div>
REQ23	3	<p>A user should be able to choose their method of notification.</p> <div style="border: 1px solid black; padding: 10px;"> <p style="text-align: center;"><b>Notifications</b></p> <p><input type="checkbox"/> E-mail <input type="text"/></p> <p><input type="checkbox"/> Text Number <input type="text"/></p> <p>Cell Provider <input type="text" value="AT &amp; T"/></p> <p><input type="checkbox"/> Discord <input type="text"/></p> <p><input type="checkbox"/> Slack <input type="text"/></p> </div>
* REQ24	1	<p>A user should be able to see a map of their route.</p> <div style="border: 1px solid black; padding: 20px; text-align: center;"> <p><b>Map Image Here</b></p> </div>

\* These requirements have been removed from consideration for the current project timeline and should be considered as potential future work.

### 3. Use Cases (Updated)

#### 3a. Stakeholders

Having a convenient, reliable way to check trip times and conditions can help reduce commuter stress and discourage the tendency to go over the safe speed limit to avoid arriving late. This may ultimately reduce accidents, which is potentially beneficial to all drivers on the road via increased safety and fewer traffic delays caused by those incidents. Making frequent travel safer through an easy-to-use notification system will benefit app users, as well as society at large. However, we have identified these primary stakeholders:

**Individual Users:** use the system for individual trip planning. These users will benefit by being able to plan their departure times to match their route conditions. This will make their daily commutes smoother and less hectic. Having updated traffic and incident information about their route as part of their routine will allow them to avoid the stress and negative effects of arriving late.

**Commercial Users:** use the system for employee route planning. These users will benefit by being able to provide value to their companies while making their work tasks easier. Having a simple way to set up alerts for the company drivers will streamline their daily route planning processes. It will also benefit their companies by reducing late deliveries and poor customer service reviews.

**Developers:** are deeply invested in their app. They want to ensure their code works well and results in the project being a success. The hard-working developers on this project are devoted to creating a well-designed system that is functional and beneficial for users. This dedication is critical to ensuring the ultimate success of the program.

### 3b. Actors and Goals

Actor	Participating /Initiating	Role	Goal
User	Initiating	The User wants to use the app to ensure the trip routes are safe and efficient.	The goal of the User is to receive traffic and weather details regarding their route based on their inputs.
Scheduling Service	Initiating	The Scheduling Service ensures that user requests are processed at the correct time.	The goal of the Scheduling Service is to run at prescribed intervals to ensure delivery of notifications.
Weather API	Participating	The Weather API provides weather data.	The goal of the Weather API is to return the requested weather data.
Traffic API	Participating	The Traffic API provides traffic flow and incident data.	The goal of the Traffic API is to return the requested traffic data.
Discord Bot	Participating	The Discord Bot ensures that users are able to receive notifications via Discord.	The goal of the Discord Bot is to process and deliver notifications to users on Discord, as requested.
Slack Bot	Participating	The Slack Bot ensures that users are able to receive notifications via Slack.	The goal of the Slack Bot is to process and deliver notifications to users on Slack, as requested.
SMS Notification Service	Participating	The SMS Notification Service ensures that users are able to receive notifications via SMS.	The goal of the SMS Notification Service is to process and deliver notifications to users via SMS, as requested.
Email Notification Service	Participating	The Email Notification Service ensures that users are able to receive notifications via Email.	The goal of the Email Notification Service is to process and deliver notifications to users via Email, as requested.

### 3c. Use Cases (Updated)

#### **i. Casual Description**

**\*\*Update Note:** All use cases were implemented as of the final demo.

##### UC1: Submit Trip Alert Request

- Description: Allows users to submit trip details and email.
- Responds to Requirements: REQ1, REQ2, REQ4, REQ9, \*REQ10, REQ13, REQ15, REQ18, REQ19, REQ20, REQ21, REQ22

##### UC2: Request Scheduled Alerts via SMS

- Description: Allows users to submit their cell phone number and carrier to receive SMS notifications.
- Responds to Requirements: REQ2, REQ3, REQ4, REQ12, REQ23

##### UC3: Request Scheduled Alerts via Discord

- Description: Allows users to choose if they want to receive alerts via Discord by providing their Discord UserID.
- Responds to Requirements: REQ2, REQ3, REQ5, REQ12, REQ23

##### UC4: Request Scheduled Alerts via Slack

- Description: Allows users to choose if they want to receive alerts via Slack by providing their Slack UserID.
- Responds to Requirements: REQ2, REQ3, REQ6, REQ12, REQ23

##### UC5: Fetch Traffic API Data

- Description: The traffic API will return the selected traffic data for each user request.
- Responds to Requirements: REQ7, REQ8, REQ16

##### UC6: Fetch Weather API Data

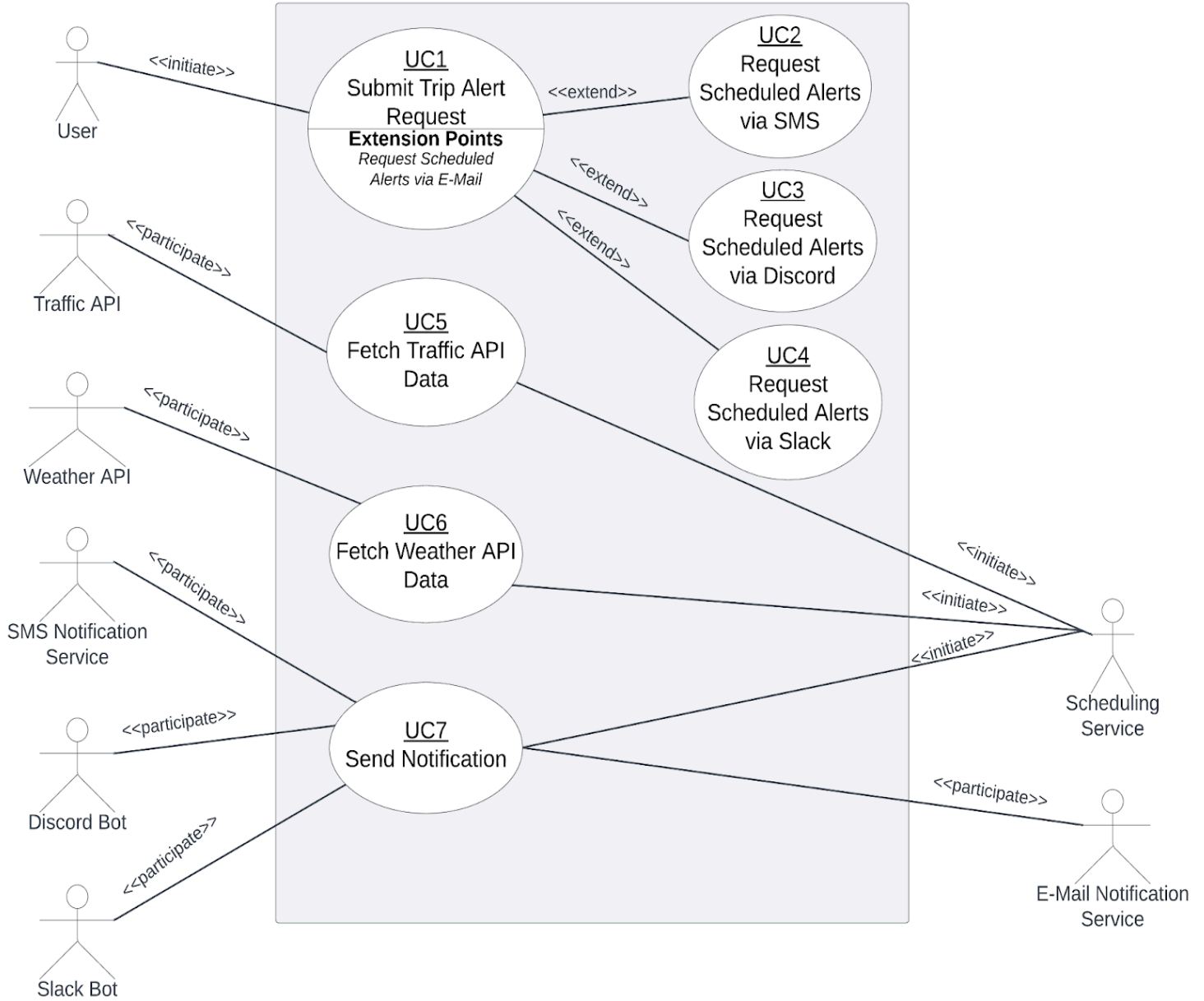
- Description: The weather API will return the selected weather data for each user request.
- Responds to Requirements: REQ7, REQ8, REQ17

##### UC7: Send Notification

- Description: The Scheduling Service triggers the trip notifications to be sent through the chosen user channels.
- Responds to Requirements: REQ2, REQ3, REQ7, REQ8, \*REQ11, REQ16

\* These requirements were removed from current work and should be considered for future work.

## ii. Use Case Diagram



### iii. Traceability Matrix (Updated)

Requirement	PW	UC1	UC2	UC3	UC4	UC5	UC6	UC7
REQ-1	5	X						
REQ-2	5	X	X	X	X			X
REQ-3	5		X	X	X			X
REQ-4	5	X	X					
REQ-5	3			X				
REQ-6	2				X			
REQ-7	5					X	X	X
REQ-8	4					X	X	X
REQ-9	5	X						
*REQ-10	3	X						
*REQ-11	5							X
REQ-12	5		X	X	X			
REQ-13	5	X						
REQ-15	2	X						
REQ-16	3					X		X
REQ-17	2						X	
REQ-18	3	X						
REQ-19	5	X						
REQ-20	5	X						
REQ-21	5	X						
REQ-22	5	X						
REQ-23	3	X	X	X	X			
Max PW		5	5	5	5	5	5	5
Total PW		53	23	21	20	12	11	27

\* These requirements were removed from current work and should be considered for future work.

### **\*\* Text Description of Traceability Matrix (Update)**

Use Case 1 (Submit Trip Alert Request) corresponds with all of the user interface requirements, as this is the GUI interface provided for users to submit their trip requests. This use case enables users to input trip details and notification preferences, ensuring the creation of alerts with their trip departure time, origin and destination addresses, and preferred notification channels. It also matches with the requirements that allow the processing of the requested notifications.

Use Cases 2-4 pertain to different notification channels - SMS, Discord, and Slack respectively, aligning with the requirements that enable users to select their preferred notification channel, providing flexibility and catering to the needs of individual users.

Use Cases 5-6 correspond to requirements that users be able to get information regarding traffic and weather conditions. Enabling this app's interaction with third-party traffic and weather APIs ensures that users are able to receive comprehensive information about their planned trips.

Use Case 7 meets the requirements that relate to integrating various components of the system to trigger and deliver notifications based on users' submitted trip details. This use case ensures timely and accurate delivery of notifications, enhancing the overall user experience.

#### iv. Fully-Dressed Description

##### Use Case UC-1: Submit Trip Alert Request

**Related Requirements:** REQ1, REQ2, REQ4, REQ9, \*REQ10, REQ13, REQ15, REQ18, REQ19, REQ20, REQ21, REQ22

**Initiating Actor:** User

**Participating Actors:** None

**Preconditions:** The user interface allows user to enter information and select options

**Postconditions:** The user's entries are submitted and processed for future use

**Flow of Events for Main Success Scenario:**

1. User submits trip details and email (default channel selection)
2. Web app (a) displays confirmation message indicating request successful, (b) processes data and stores user's entries, (c) passes details of alert scheduling to the Scheduling Service to be sent via the Email Notification Service

**Flow of Events for Extensions (Alternate Scenarios):**

2a. Input validation indicates required fields are not filled properly

1. Web app displays warning to the user that required fields aren't filled
2. User fills in required fields and resubmits
3. Web app (a) displays confirmation message indicating request successful, (b) processes data and stores user's selections for future use, (c) passes details of alert scheduling to the Scheduling Service to be sent via the Email Notification Service

2b. Selected activity may entail requesting SMS alerts (UC-2)

2c. Selected activity may entail requesting Discord alerts (UC-3)

2d. Selected activity may entail requesting Slack alerts (UC-4)



### Use Case UC-2: Request Scheduled Alerts via SMS

**Related Requirements:** REQ2, REQ3, REQ4, REQ12, REQ23

**Initiating Actor:** User

**Participating Actors:** None

**Preconditions:** User has entered trip information and email in appropriate fields

**Postconditions:** The user's entries are submitted and processed for future use

**Flow of Events for Main Success Scenario:**

1. User selects SMS Notification option
2. User enters their cell phone number
3. User selects their cell provider
4. User submits form
5. Web app (a) displays confirmation message indicating request successful, (b) processes data and stores user's selections for future use, (c) passes details of alert scheduling to the Scheduling Service to be sent via the SMS Notification Service

**Flow of Events for Extensions (Alternate Scenarios):**

2a. Input validation indicates required SMS Notification fields are not filled properly

1. Web app displays warning to the user that required fields aren't filled
2. User fills in required fields and resubmits

### Use Case UC-3: Request Scheduled Alerts via Discord

**Related Requirements:** REQ2, REQ3, REQ5, REQ12, REQ23

**Initiating Actor:** User

**Participating Actors:** None

**Preconditions:** User has entered trip information and email in appropriate fields

**Postconditions:** The user's entries are submitted and processed for future use

**Flow of Events for Main Success Scenario:**

1. User selects Discord notification option
2. User enters their Discord UserID
3. User submits form
4. Web app (a) displays confirmation message indicating request successful, (b) processes data and stores user's selections for future use, (c) passes details of alert scheduling to the Scheduling Service to be sent via the Discord Bot

**Flow of Events for Extensions (Alternate Scenarios):**

2a. Input validation indicates required Discord notification fields are not filled properly

1. Web app displays warning to the user that required fields aren't filled
2. User fills in required fields and resubmits

#### Use Case UC-4: Request Scheduled Alerts via Slack

**Related Requirements:** REQ2, REQ3, REQ6, REQ12, REQ23

**Initiating Actor:** User

**Participating Actors:** None

**Preconditions:** User has entered trip information and email in appropriate fields

**Postconditions:** The user's entries are submitted and processed for future use

**Flow of Events for Main Success Scenario:**

1. User selects Slack notification option
2. User enters their Slack UserID
3. User submits form
4. Web app (a) displays confirmation message indicating request successful, (b) processes data and stores user's selections for future use, (c) passes details of alert scheduling to the Scheduling Service to be sent via the Slack Bot

**Flow of Events for Extensions (Alternate Scenarios):**

- 2a. Input validation indicates required Discord notification fields are not filled properly
1. Web app displays warning to the user that required fields aren't filled
  2. User fills in required fields and resubmits

#### Use Case UC-5: Fetch Traffic API Data

**Related Requirements:** REQ7, REQ8, REQ16

**Initiating Actor:** Scheduling Service

**Participating Actors:** Traffic API

**Preconditions:** Scheduling Service has determined timing based on notification schedule

**Postconditions:** Current traffic and incident data has been retrieved from Traffic API

**Flow of Events for Main Success Scenario:**

1. Scheduling Service triggers based on notification schedule
2. Application middleware sends fetch request to Traffic API
3. Traffic data is returned from the Traffic API
4. Middleware processes traffic data for inclusion in notifications

**Flow of Events for Extensions (Alternate Scenarios):**

- 2a. Error occurs and fetch from Traffic API fails
1. Fetch request is unfulfilled by Traffic API
  2. Error is thrown
  3. Fetch request is processed again to retrieve Traffic API data up to max request limit

### Use Case UC-6: Fetch Weather API Data

**Related Requirements:** REQ7, REQ8, REQ17

**Initiating Actor:** Scheduling Service

**Participating Actors:** Weather API

**Preconditions:** Scheduling Service has determined timing based on notification schedule

**Postconditions:** Weather data has been retrieved from Weather API

**Flow of Events for Main Success Scenario:**

1. Scheduling Service triggers based on notification schedule
2. Application middleware sends fetch request to Weather API
3. Weather data is returned from the Weather API
4. Middleware processes weather data for inclusion in notifications

**Flow of Events for Extensions (Alternate Scenarios):**

2a. Error occurs and fetch from Weather API fails

1. Fetch request is unfulfilled by Weather API
2. Error is thrown
3. Fetch request is processed again to retrieve Weather API data up to max request limit

### Use Case UC-7: Send Notification

**Related Requirements:** REQ2, REQ3, REQ7, REQ8, \*REQ11, REQ16

**Initiating Actor:** Scheduling Service

**Participating Actors:** Any of: SMS Notification Service, Email Notification Service, Slack Bot, and/or Discord Bot

**Preconditions:** Weather and traffic data has been retrieved to create notification for user

**Postconditions:** Message is sent to the user through their chosen channels

**Flow of Events for Main Success Scenario:**

1. Scheduling Service triggers trip notification messages to be sent
2. Messages are sent to users through any of SMS Notification Service, Email Notification Service, Slack Bot, and/or Discord Bot
3. Response status indicates message sending successful
4. Database updated to indicate trip notification sent

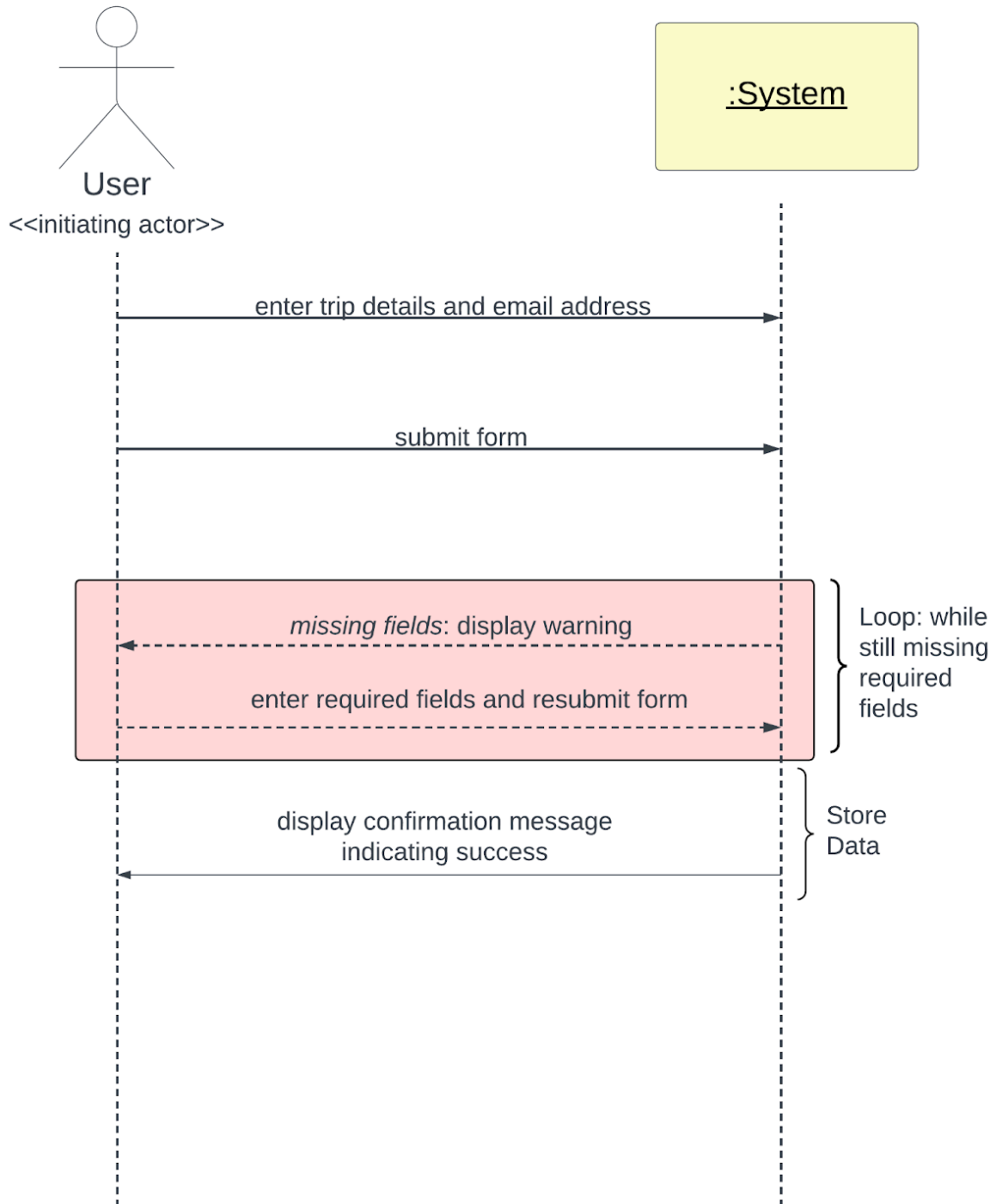
**Flow of Events for Extensions (Alternate Scenarios):**

2a. Error occurs and message doesn't send via channel

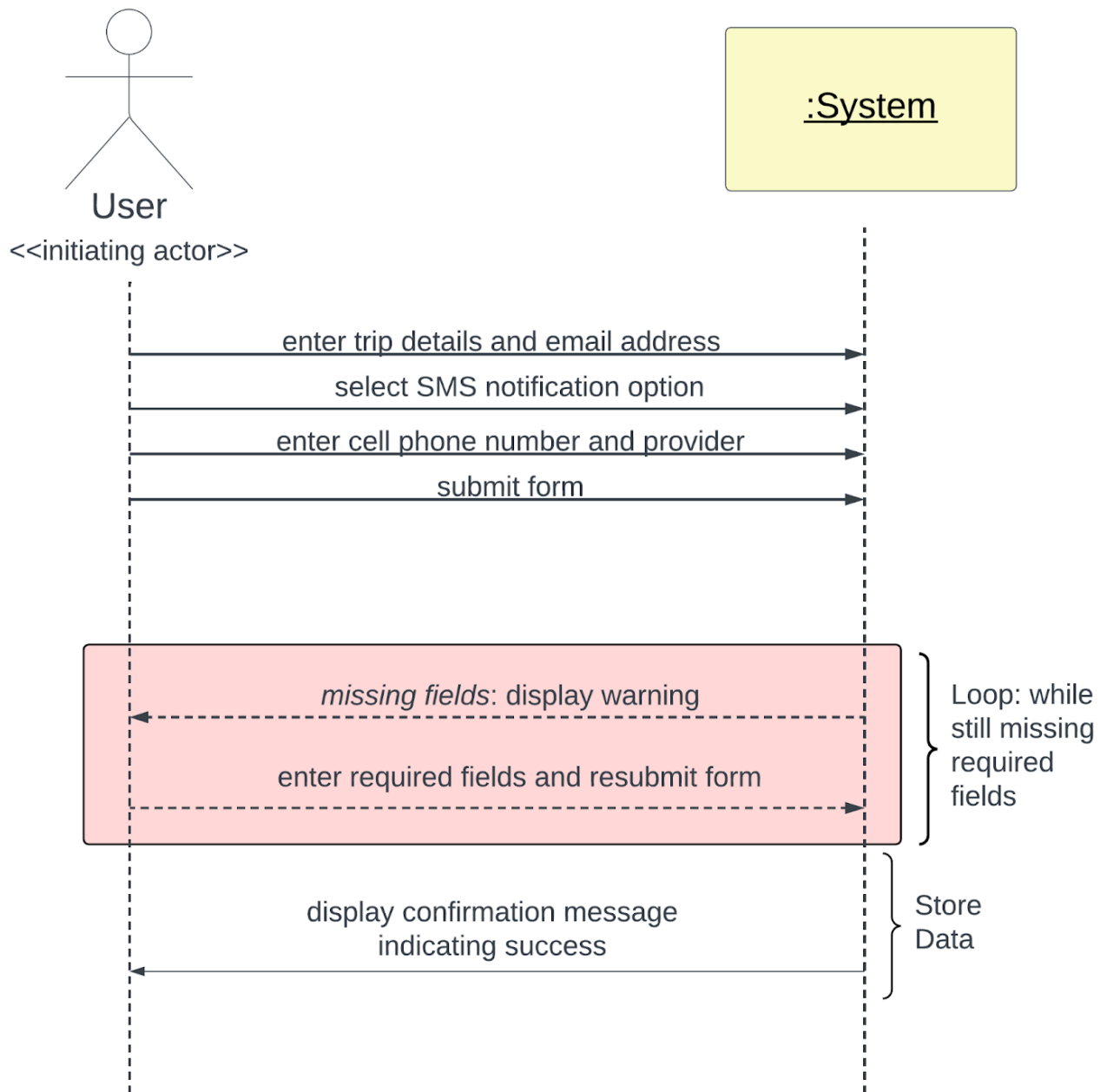
1. Response Status indicates message was not sent successfully
2. Error is thrown
3. Message is resent through the appropriate channel up to max attempt limit

### 3d. System Sequence Diagrams

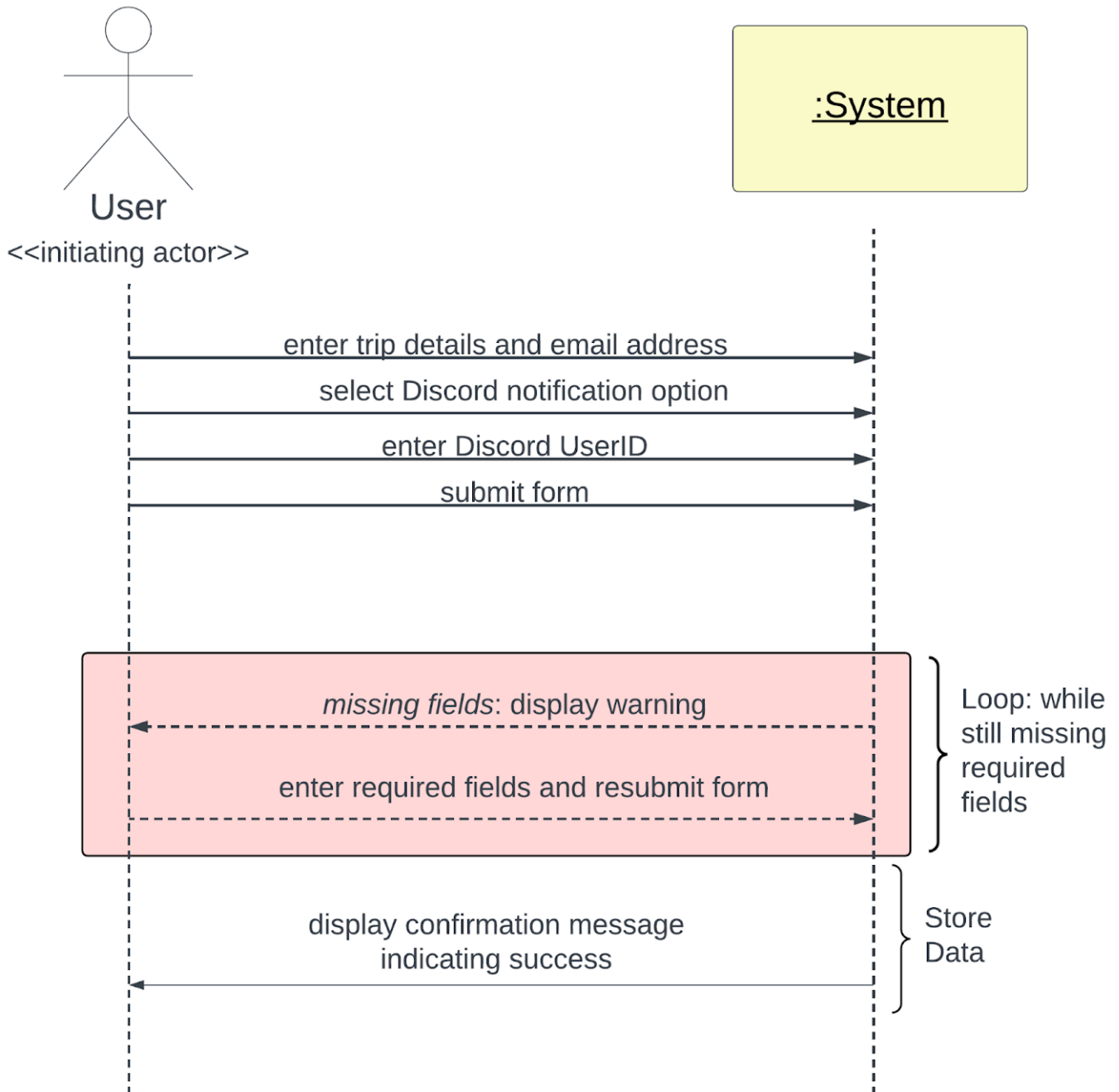
## UC-1: Submit Trip Alert Request



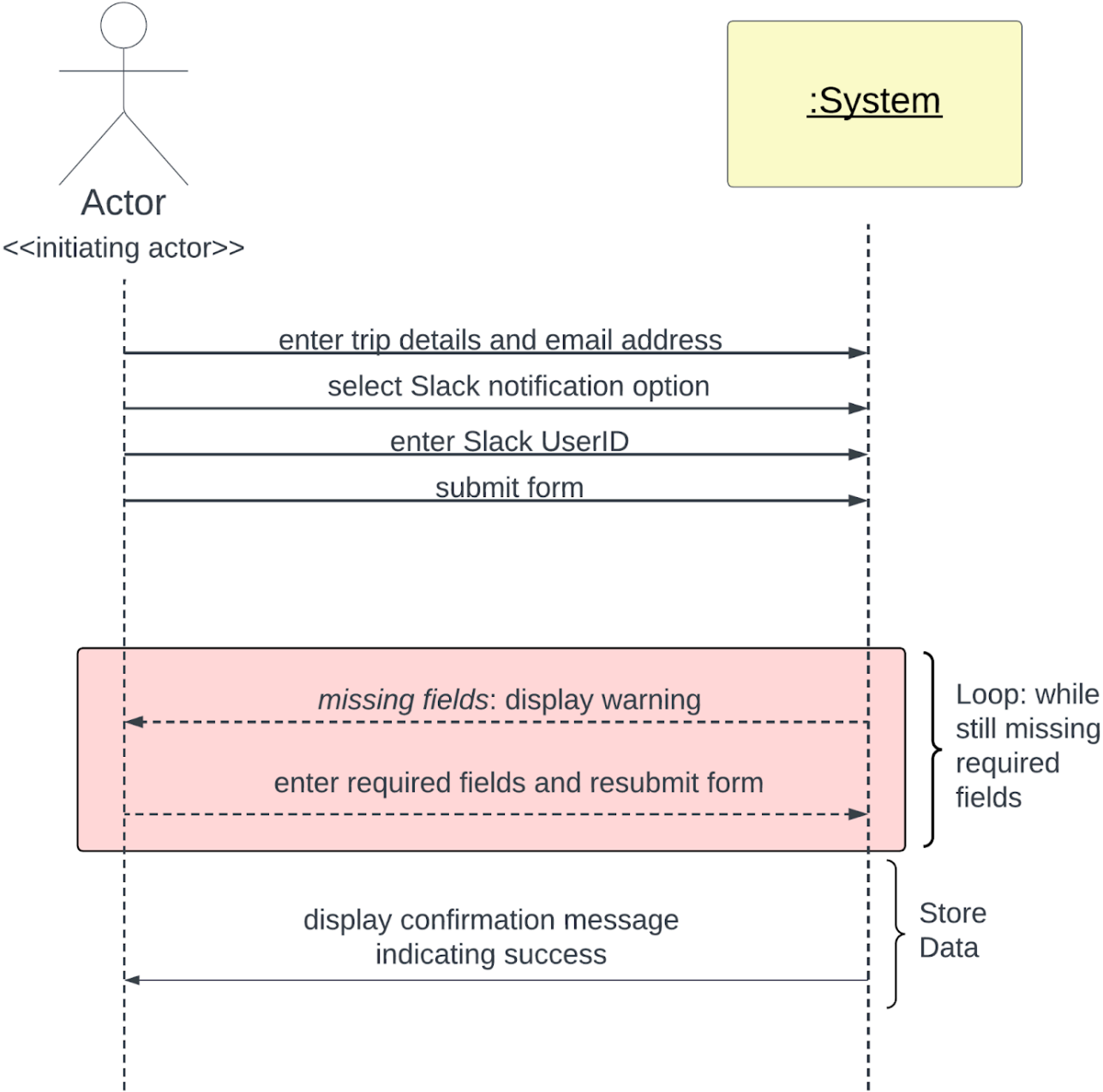
## UC-2: Request Scheduled Alerts via SMS



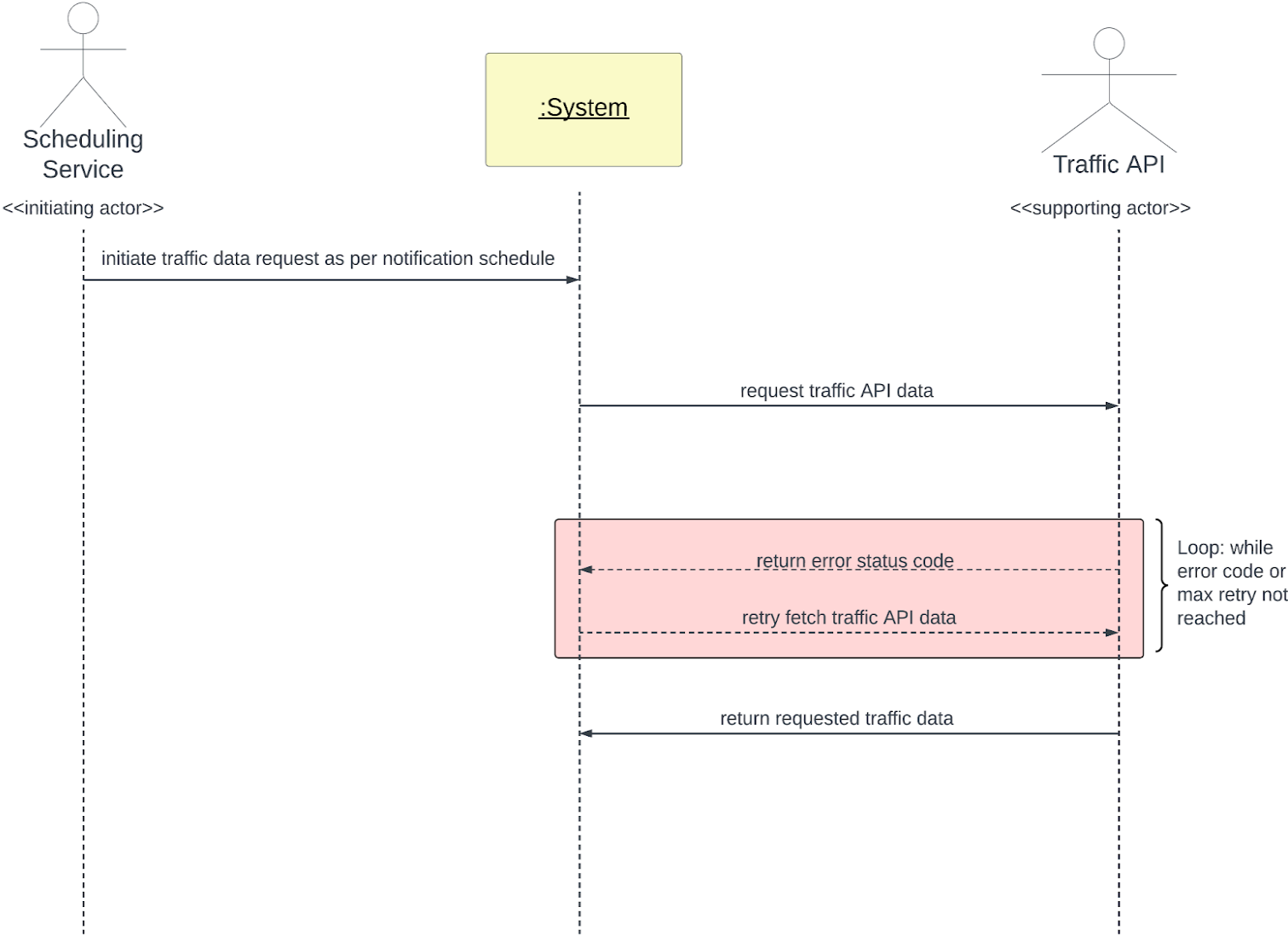
# UC-3: Request Scheduled Alerts via Discord



# UC-4: Request Scheduled Alerts via Slack

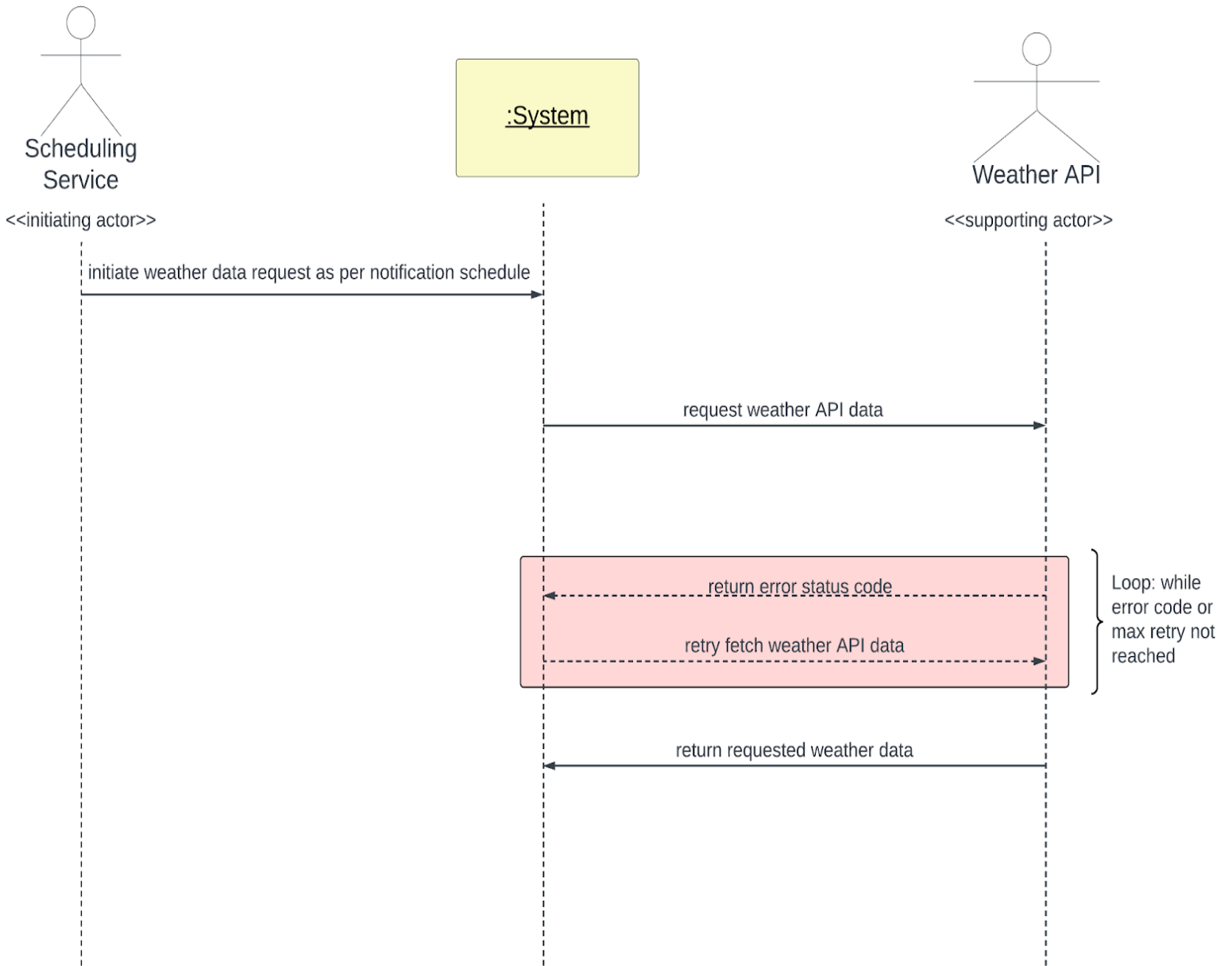


# UC-5: Fetch Traffic API Data

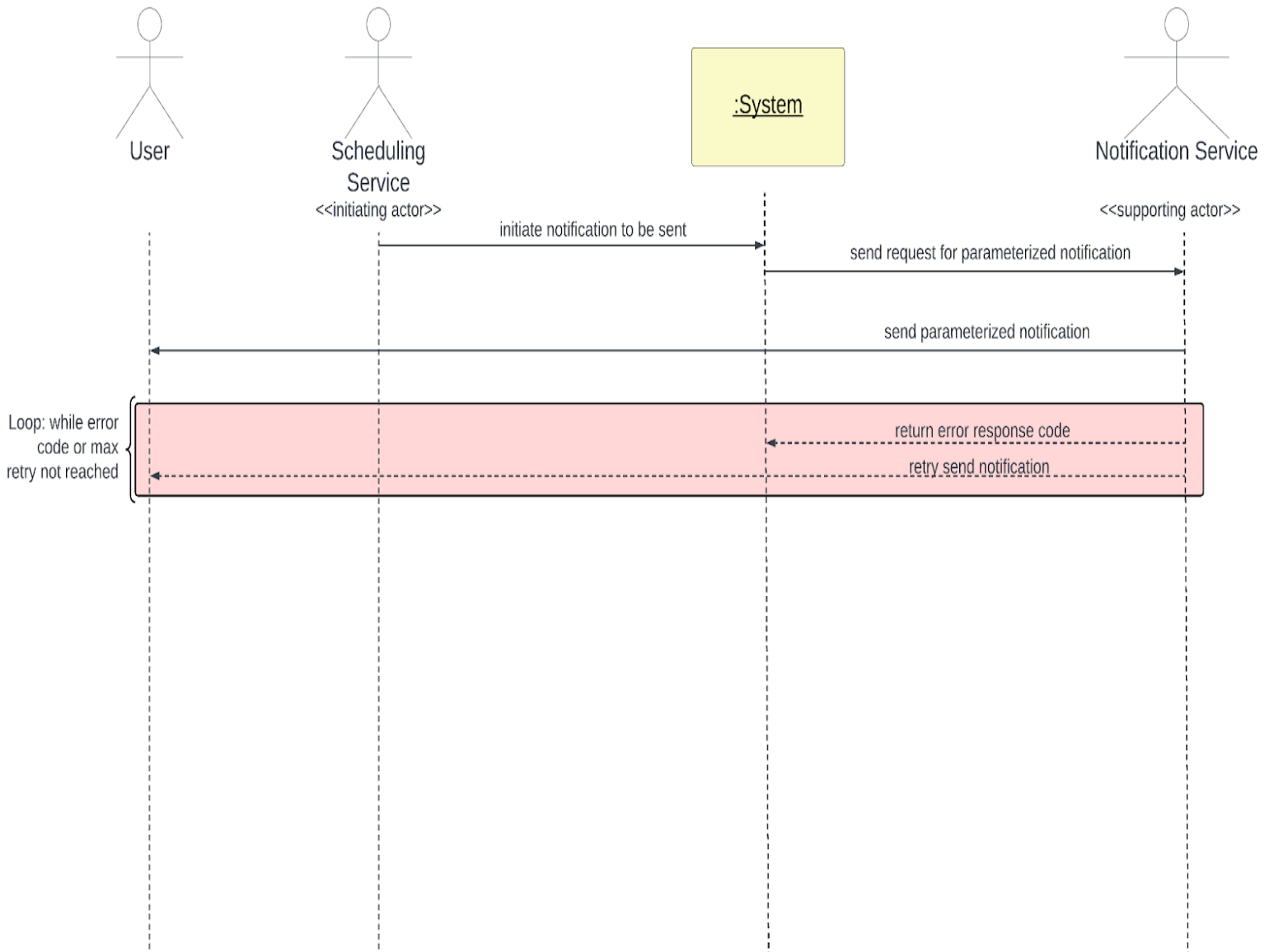




## UC-6: Fetch Weather API Data



## UC-7: Send Notification



**\*Note - Notification Service in the above sequence diagram represents notifications for all channels including E-Mail, SMS, Discord and Slack.**

## **4. User Interface Specification** (Updated)

### **4a. Preliminary Design** (Updated)

1. Enter Departure Date and Time
  - 1.1. From the main page, the user clicks on the calendar control and selects the date and time. Date and time can be selected from the calendar displayed.
2. Enter Departure Location
  - 2.1. \* From the main page, the user selects the address related fields to enter their starting location. Fields for street address, city, state, and zip code are provided.  
\*\* From the main page the user begins free-form entry of their departure address. A continuous partial address search is performed using TomTom API to provide an address list. The user then selects the correct address from the list.
3. Enter Destination Location
  - 3.1. \* From the main page, the user selects the address related fields to enter their ending location. Fields for street address, city, state, and zip code are provided.  
\*\* From the main page the user begins free-form entry of their destination address. A continuous partial address search is performed using TomTom API to provide an address list. The user then selects the correct address from the list.
4. Select Notification Channel
  - 4.1. \* From the main page, the user selects one or more notification methods (E-mail, SMS, Chatbot) using checkboxes and provides the appropriate user information for the chosen method (Email Address, Phone Number, Chatbot Username). Fields for user information are available for each method.  
\*\* From the main page, the user enters their email address, by default, then selects one or more notification methods (SMS, Chatbot) using checkboxes and provides the appropriate user information for the chosen method (Phone Number, Chatbot Username). Fields for user information are available for each method.
5. Submit Request for Notification

5.1. From the main page, the user presses the submit button to send the information entered on the form to myTrafficWizard for notification.

\* Indicates Removed Text

\*\* Indicates Added Text to Replace Removed Text

Figure: myTrafficWizard (Desktop Layout) - Original

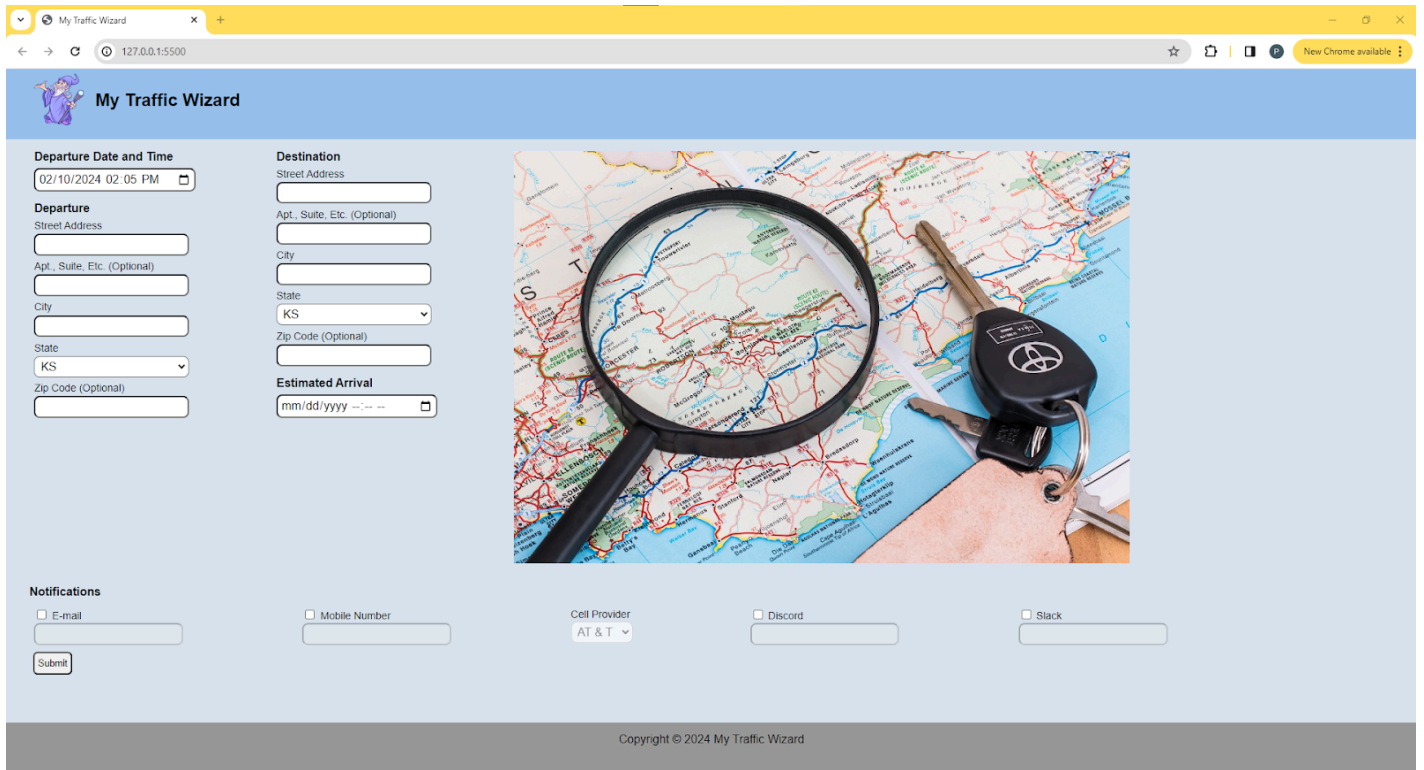


Figure: myTrafficWizard (Desktop Layout) - Updated

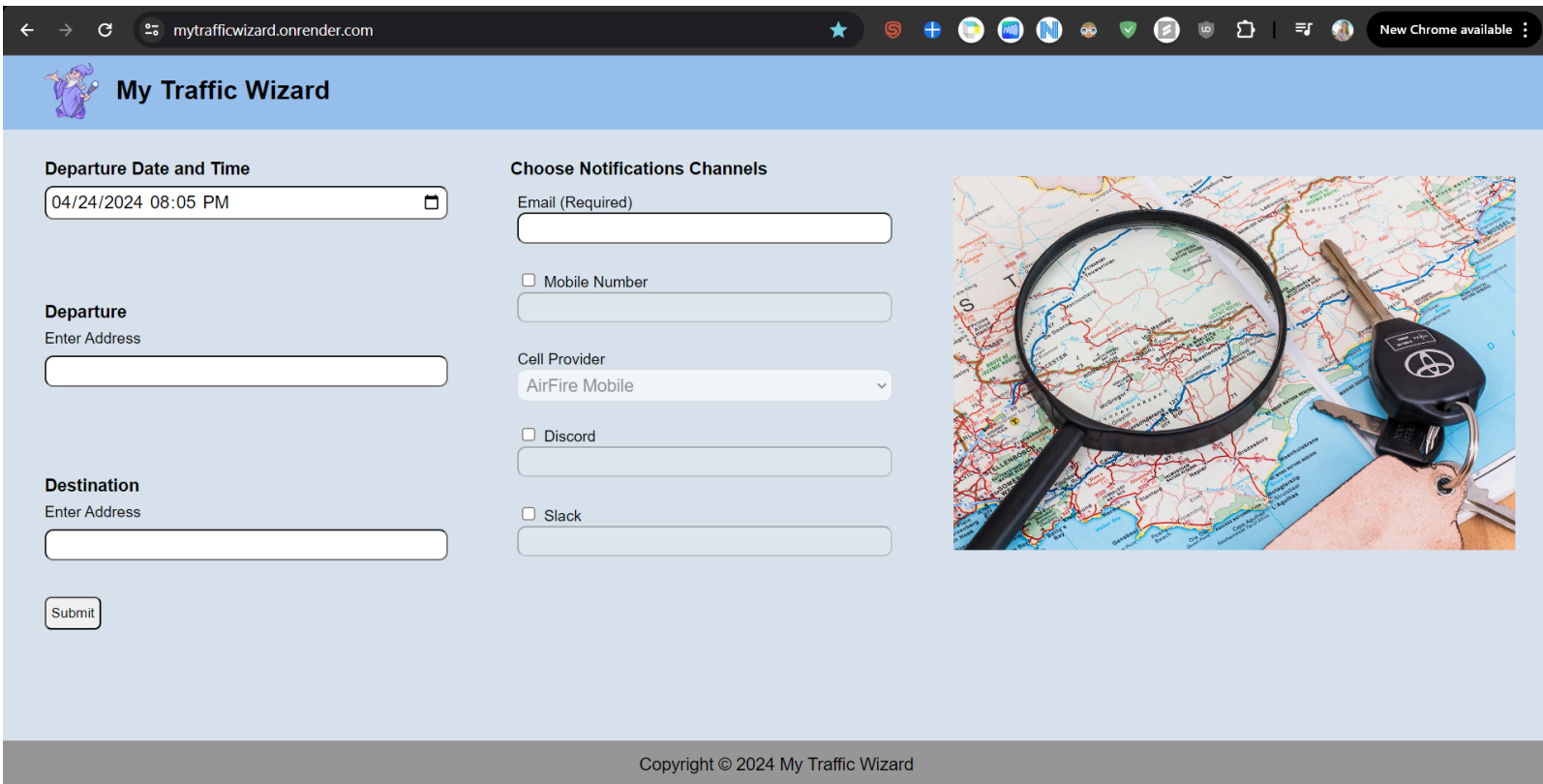




Figure: myTrafficWizard (Mobile Layout) - Original

**My Traffic Wizard**



**Departure Date and Time**

**Departure**  
Street Address  
  
Apt., Suite, Etc. (Optional)  
  
City  
  
State  
  
Zip Code (Optional)


**Destination**  
Street Address  
  
Apt., Suite, Etc. (Optional)  
  
City  
  
State  
  
Zip Code (Optional)

**Estimated Arrival**


**Notifications**  
 E-mail  
  
 Mobile Number  
  
Cell Provider  
  
 Discord  
  
 Slack

Copyright © 2024 My Traffic Wizard

**Figure: myTrafficWizard (Mobile Layout) - Updated**



# My Traffic Wizard



## Departure Date and Time

## Departure

Enter Address

## Destination

Enter Address

## Choose Notifications Channels

Email (Required)

Mobile Number

Cell Provider

AirFire Mobile

Discord

Slack

Submit

Copyright © 2024 My Traffic Wizard

#### **4b. User Effort Estimation** (Updated)

1. Enter Departure Date and Time: The sequence below shows the actions required to select the departure time February 28, 2024 08:00 AM  
Navigation: Total 1 click, as follows
  - 1.1. Click on the calendar buttonData Entry: Minimum of 4 clicks, as follows
  - 1.2. Click departure date February 28, 2024
  - 1.3. Click the departure hour 08
  - 1.4. Click the departure minutes 00
  - 1.5. Click the departure time of day AM
2. Enter Departure Address: The following sequence of actions would allow a user to enter the departure location 123 Anywhere Street, Hays KS.  
Navigation: \* Total of 3 clicks. \*\* Total of 1 click.
  - 2.1. \* Click the Street Address Text Box  
\*\* Click the Address Text Box
  - 2.2. \* Click the City Text Box
  - 2.3. \* Click the State Dropdown BoxData Entry: \* Total of 23 key presses and 1 click, as follows:  
\*\* Minimum of 3 key presses and 1 click, as follows:
  - 2.4. \* Press 19 characters to enter 123 Anywhere Street  
\*\* Press 3 characters to enter 123. A lookup to the TomTom API generates a list of matching addresses. Additional characters can be typed to narrow the list.
  - 2.5. \* Press 4 characters for the Hays  
\*\* Select the address 123 Anywhere Street, Hays KS from the dropdown list.
  - 2.6. \* Click 1 time to select KS from the dropdown list
3. Enter Destination Address: The following sequence of actions would allow a user to enter the destination location 123 Main Street, Hays KS.  
Navigation: \* Total of 3 clicks. \*\* Total of 1 click.
  - 3.1. \* Click the Street Address Text Box  
\*\* Click the Address Text Box



3.2. \* Click the City Text Box

3.3. \*\* Click the State Dropdown Box

Data Entry: \* Total of 19 key presses and 1 click, as follows:

\*\* Minimum of 3 key presses and 1 click, as follows:

3.4. \* Press 15 characters to enter 123 Main Street

\*\* Press 3 characters to enter 123. A lookup to the TomTom API generates a list of matching addresses. Additional characters can be typed to narrow the list.

3.5. \* Press 4 characters for the Hays

\*\* Select the address 123 Main Street, Hays KS from the dropdown list.

3.6. \* Click 1 time to select KS from the dropdown list

4. Enter Notification Preference: The sequence shows the actions required to choose email as the notification method for the address myemail@gmail.com.

Navigation: \* Total of 2 clicks as follows. \*\* Total of 1 click as follows

4.1. \* Click the Checkbox next to Email

4.2. Click the Email Text Box

Data Entry: Total of 17 key presses as follows

4.3. Press 17 characters to enter myemail@gmail.com

5. Submit the Notification Request: The following action would allow the user to submit the form data for a notification request.

Navigation: Total of 1 click

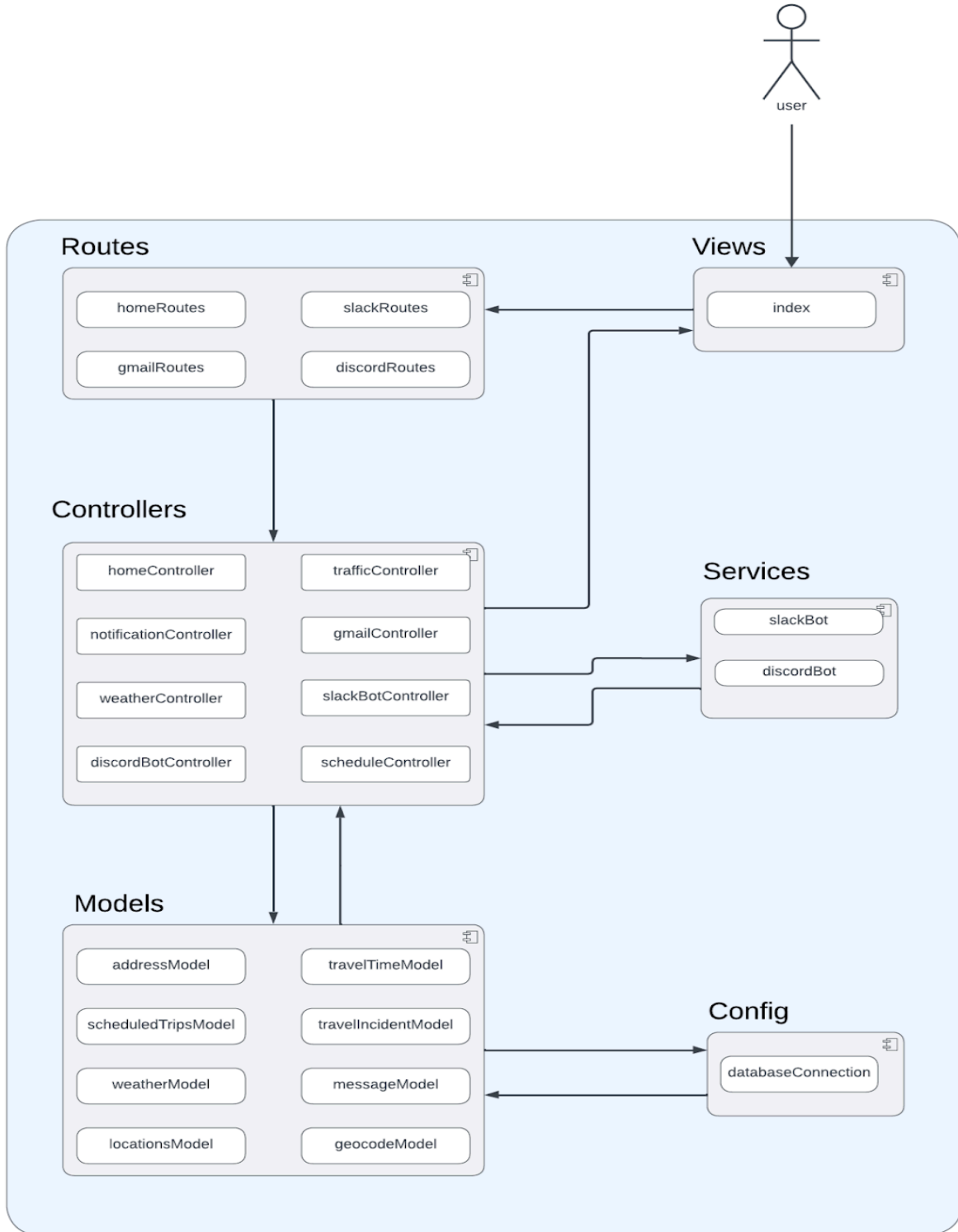
5.1. Click the Submit Button to send the data to the system

\* Indicates Removed Text

\*\* Indicates Added Text to Replace Removed Text

## 5. System Architecture and System Design

### 5a. Identifying Subsystems



The system follows an extended MVC design pattern and consists of the main subsystems: controllers, models, views and the ancillary subsystems - routes, services and a configuration component. Each subsystem can be described in the following manner:

**Models:** This subsystem serves as the application's data access layer, executing CRUD (Create, Read, Update, Delete) operations on the database and interfacing directly with third-party APIs for external data retrieval. Models are invoked by Controllers to perform data retrieval and manipulation which ensures a seamless exchange mechanism within the system..

**Views:** This subsystem is tasked with interfacing directly with the user, rendering the application's UI and capturing user inputs via forms. These inputs are subsequently routed to Controllers through defined endpoints (routes).

**Controllers:** Serving as the conduit between Models and Views, this subsystem processes user input received through Routes, leveraging Models for data operations and overseeing notification generation and dispatch. With extensive linkages to other subsystems, Controllers function as the operational core of the system, orchestrating the application's logic and user interactions.

**Routes:** This subsystem specifies the application's accessible endpoints, associating them with pertinent Controller actions. It acts as a pivotal intermediary, channeling web API requests from the user interface to the appropriate Controllers, thereby enabling structured request management.

**Services:** Encompassing a suite of reusable components, the Services subsystem enhances the application's functionality, offering utility and support to the primary subsystems.

**Config:** This subsystem is responsible for managing the application's configuration settings.

## **5b. Architecture Styles**

The team chose a Model-View-Controller (MVC) architectural style for the myTrafficWizard project. This is a common choice for web-based projects in that it incorporates inherent flexibility and scalability. With this choice, we leverage the prior knowledge of members with experience using the pattern as well.

The MVC pattern lends itself to asynchronous, parallel processing of web applications like this one. It encourages separation of concerns where objects are expert doers with high cohesion and loose coupling. By using it, components need not know how others accomplish their tasks. They need only know that a component fulfills that task. By choosing the MVC pattern, the system makes it convenient for the user to receive information about traffic and weather information in a way they prefer. There are several views of the data provided to the user including the Notification Requests web interface and communication channels serving those Notifications.

Our MVC pattern is paired with the client-server architecture. This approach maintains the interconnections necessary to retrieve information from several APIs. By doing this, we decouple the processing and handling of application logic. The methods by which input is received from the user and by which information is delivered to the user may operate independent of each other. This approach makes possible the addition of future components for Notification Requests, communication channels, and expanded functionality without a need to modify the entire code base.

The program uses an event driven architecture to send notifications based on a departure time previously determined by the user. To do this, the scheduler finds one or more notifications with desired date-time equal to the current date-time. It then initiates the notification of users by requesting the appropriate controllers retrieve, package, and send that information to a user.

### **5c. Mapping Subsystems to Hardware**

- **Database Subsystem:** This subsystem manages the storage of user information and their preferred notification method.
- **Homepage Subsystem:** This subsystem hosts the homepage of myTrafficWizard program and handles all the information input by the users. It is integrated with the database subsystem to retrieve relevant data and sends notifications via email, SMS, Slack and/or Discord to inform users about travel time, accidents, and weather updates.
- **Weather API Interface:** This subsystem connects to an external weather API to retrieve current weather conditions, forecasts, and alerts. It interfaces with the weather service provider's API over the internet to fetch relevant weather data for integration into the myTrafficWizard system and for sending weather-related notifications to users.
- **Traffic API Interface:** This subsystem interacts with an external traffic API to access real-time traffic data, incident reports, and historical traffic patterns. It communicates with the traffic service provider's API through the web to fetch traffic-related information for analysis, generating notifications about traffic incidents, and travel time estimations.
- **Communication Infrastructure:** This subsystem provides the underlying infrastructure for communication between different components of the myTrafficWizard, including database servers, homepage/notification servers, and API interfaces. It ensures reliable and secure data exchange over the network, to deliver the notifications to users.

## **5d. Connectors and Network Protocols**

- **HTTPS:** This is the protocol that our web application protocol will be running on. This is a widely used protocol for communication between web servers and clients.
- **API Connectors:** API connectors are used to integrate external APIs into the myTrafficWizard program, allowing it to fetch data from services such as weather APIs and traffic APIs. These connectors handle the authentication, request, and response processes. They usually use HTTP or HTTPS protocols for communication.
- **Notification Connectors:** These connectors enable the myTrafficWizard program to send notifications to users using various channels such as email, SMS, or push notifications to Slack and Discord. They interface with messaging services or email servers to deliver notifications based on the user preferences

### **5e. Global Control Flow**

- **Execution orderness:** The system uses event-driven architecture. In this paradigm, the program waits for user and system events to trigger handlers to perform the duties required by an event. These handlers act as experts that process the event and return the results. Events in this style may come in any order and at any time.
- **Time dependency:** While the myTrafficWizard is not a real-time system, it does have timing requirements. The scheduler waits for an appropriate time before firing events to send notifications to users. Even with that time component, the system processes events as they occur and then responds to them.

## **5f. Hardware Requirements**

- **Screen Display:** The homepage subsystem will be able to run on multiple devices such as laptops, desktops, and mobile devices. This will require it to be able to run on different resolutions because of the varying device types and display sizes.
- **Communication Network:** The myTrafficWizard program will require a stable and fast connection to the network to maintain a smooth distribution of data between all our subsystems and sending notifications to the users.
- **Database Server:** The database subsystem using PostgreSQL will need sufficient computing resources and disk space in order to store and process operations.



## 6. Project Size Estimation Based On Use Case Points (Updated)

### Actor Classification and Weights

Actor Type	Description of Actor Type	Weight
Simple	The actor is a system that interacts with the web app system through the API.	1
Average	The actor interacts through networking communication protocols or a text-based interface.	2
Complex	The actor is an app user interacting through the graphical user interface (GUI)	3

### Actor Classification for My Traffic Wizard

Actor Name	Description of Relevant Characteristics	Complexity	Weight
User	User interacts with the web app via the graphical user interface (such as when setting up a trip route).	Complex	3
Scheduling Service	Scheduling Service is interacting with the web app system through a protocol.	Average	2
Weather API	Weather API interacts with our system through the defined API.	Simple	1
Traffic API	Traffic API interacts with our system through the defined API.	Simple	1
Discord Bot	Discord Bot interacts with our system through the defined API.	Simple	1
Slack Bot	Slack Bot interacts with our system through the defined API.	Simple	1
SMS Notification Service	SMS Notification Service interacts with our system through the defined API.	Simple	1
Email Notification Service	Email Notification Service interacts with our system through the defined API.	Simple	1

### Unadjusted Actor Weight (UAW)

$$UAW = 1 * \text{Complex} + 1 * \text{Average} + 6 * \text{Simple} = 1 * 3 + 1 * 2 + 6 * 1 = 11$$

## Use Case Categories and Weights

Use Case Category	Description of Use-Case Category	Weight
Simple	Simple interface design, no more than one participating actor (in addition to the initiating actor), and/or 3 or fewer steps in the success scenario.	5
Average	Moderate interface design, 2+ participating actors (in addition to the initiating actor), and/or 4-7 steps in the success scenario.	10
Complex	Complex interface design or processing requirements, 3+ participating actors (in addition to the initiating actor), and/or 7+ steps in the success scenario.	15

## Use Case Classification for My Traffic Wizard

Use Case	Description of Use Case	Category	Weight
Submit Trip Alert Request (UC-1)	Complex interface, 4 steps main success scenario, 0 participating actors	Complex	15
Request Scheduled Alerts via SMS (UC-2)	Simple interface, 7 steps main success scenario, 0 participating actors	Average	10
Request Scheduled Alerts via Discord (UC-3)	Simple interface, 6 steps main success scenario, 0 participating actors	Average	10
Request Scheduled Alerts via Slack (UC-4)	Simple interface, 6 steps main success scenario, 0 participating actors	Average	10
Fetch Traffic API Data (UC-5)	Simple interface, 4 steps main success scenario, 1 participating actor (Traffic API)	Simple	5
Fetch Weather API Data (UC-6)	Simple interface, 4 steps main success scenario, 1 participating actor (Weather API)	Simple	5
Send Notification (UC-7)	Complex interface, 3 steps main success scenario, up to 4 participating actors (Email Notification Service, SMS Notification Service, Slack Bot, and Discord Bot)	Complex	15

### Unadjusted Use Case Weight (UUCW)

$$\text{UUCW} = 2 * \text{Complex} + 3 * \text{Average} + 2 * \text{Simple} = 2 * 15 + 3 * 10 + 2 * 5 = 70$$

### Unadjusted Use Case Points (UUCP)

$$\text{UUCP} = \text{UAW} + \text{UUCW} = 11 + 70 = 81$$

### Technical Complexity Factors for My Traffic Wizard

Technical Factor	Description of Technical Factor	Weight	Perceived Complexity	Calculated Factor (Weight x Perceived Complexity)
T1	Web-based distributed system as web app	2	5	2 x 5 = 10
T2	Performance and response times must be good to avoid calls/requests timing out	1	5	1 x 5 = 5
T3	Web app is efficient for end-users	1	3	1 x 3 = 3
T4	Internal processing is moderate	1	3	1 x 3 = 3
T5	Code designed to be modular and reusable	1	3	1 x 3 = 3
T6	Installation ease not of particular importance	0.5	0	0.5 x 0 = 0
T7	Ease of use is moderately important	0.5	3	0.5 x 3 = 1.5
T8	Portability is not a high priority	2	1	2 x 1 = 2
T9	Adding/modifying features with ease is important	1	3	1 x 3 = 3
T10	Concurrent use by multiple users is essential	1	5	1 x 5 = 5
T11	Security is a moderate concern	1	4	1 x 4 = 4
T12	No direct third-party access is needed	1	0	1 x 0 = 0
T13	Special user training is not required	1	0	1 x 0 = 0
<b>Technical Factor Total</b>				<b>39.5</b>

#### Technical Complexity Factor (TCF)

$$\text{TCF} = \text{Constant-1} + \text{Constant-2} * \text{Technical Factor} = 0.6 + (0.01 * 39.5) = 0.995$$

This results in a reduction of UCP by 0.5%

**Environmental Complexity Factors for My Traffic Wizard**

<b>Environmental Factor</b>	<b>Description of Environmental Factor</b>	<b>Weight</b>	<b>Perceived Impact</b>	<b>Calculated Factor (Weight x Perceived Impact)</b>
E1	Familiarity with development process is fair	1.5	2	1.5 x 2 = 3
E2	Application problem experience is fair	0.5	2	0.5 x 2 = 1
E3	Paradigm experience (OOP) is good	1	3	1 x 3 = 3
E4	Lead analyst capability is good	0.5	3	0.5 x 3 = 1.5
E5	Motivation is excellent	1	4	1 x 4 = 4
E6	Stable requirements are anticipated	2	4	2 x 4 = 8
E7	Part-time student staff is expected	-1	5	-1 x 5 = -5
E8	Programming language is average difficulty	-1	3	-1 x 3 = -3
<b>Environmental Factor Total</b>				<b>12.5</b>

**Environmental Complexity Factor (ECF)**

$ECF = \text{Constant-1} + \text{Constant-2} * \text{Environmental Factor Total} = 1.4 + (-0.03 * 12.5) = 1.025$

**Use Case Points (UCP)**

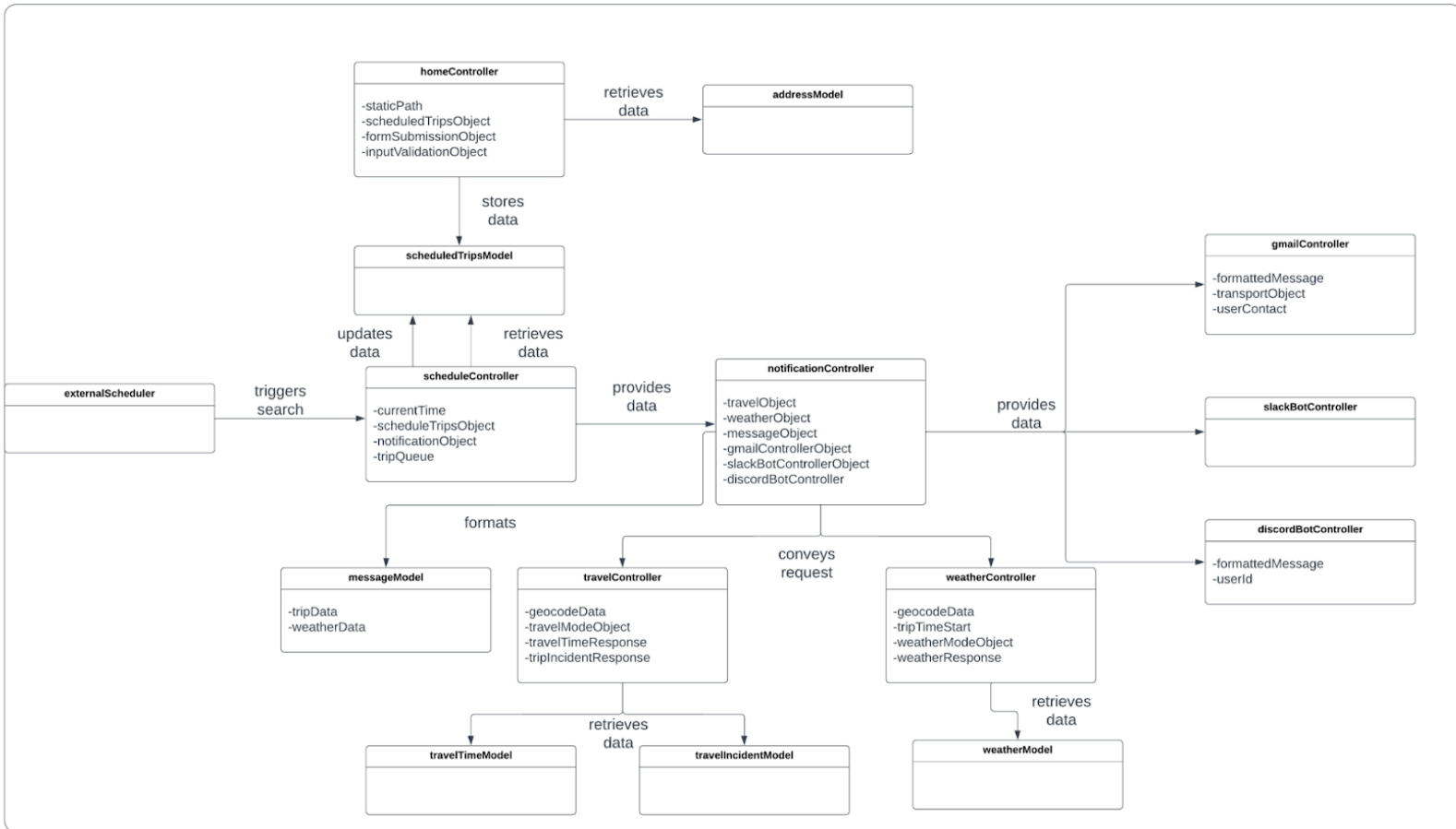
$UCP = UUCP * TCF * ECF = 81 * 0.995 * 1.025 = 82.61 \text{ Use Case Points}$

**Project Duration (\*\*Updated to use PF = 28 hours)**

$\text{Duration} = UCP * PF = 82.61 * 28 = 2313.08 \text{ person-hours}$

## 7. Analysis and Domain Modeling (Updated)

### 7a. Conceptual Model



## i. Concept Definitions

Responsibility Description	Type	Concept Name
Validate user input from form submission and display error messages if input is missing.	D	inputValidation
Accept alert requests from users and deliver confirmation of submissions to users.	D	homeController
Deliver alert requests to the database.	D	scheduledTripsModel
Store trip details, notification history, and geocode details.	K	scheduledTripsModel
Keep track of current time and trigger notification checks at regular intervals.	D	externalScheduler
Check for upcoming scheduled alerts when triggered.	D	scheduleController
Look up user notification details and delegate lookup of third-party traffic and weather data to respective controllers.	D	notificationController
Fetch data from Traffic API.	D	travelController
Fetch data from Weather API.	D	weatherController
Handle any failed requests by initiating another request.	D	baseFetchRetry
Call message formatter once data is received.	D	notificationController
Format notifications to be sent.	K	messageModel
Send notifications through email and/or SMS.	D	gmailController
Send notifications through Slack.	D	slackBotController
Send notifications through Discord.	D	discordBotController

## ii. Association Definitions

Concept Pair	Association Description	Association Name
homeController ↔ scheduledTripsModel	homeController submits user entries to the database through the scheduledTripsModel.	stores data
scheduleController ↔ scheduledTripsModel	scheduleController calls scheduledTripsModel to find upcoming scheduled trips in the database.	retrieves data
scheduleController ↔ notificationController	scheduleController calls the notificationController when a scheduled notification is present.	provides data
notificationController ↔ travelController	notificationController calls the travelController to reach out to the traffic models for API data.	conveys request
travelController ↔ travelTimeModel	travelController calls the travelTimeModel for API data on travel time.	retrieves data
travelController ↔ travelIncidentModel	travelController calls the travelIncidentModel for API data on incidents along the route.	retrieves data
notificationController ↔ weatherController	notificationController calls the weatherController to reach out to the weatherModel for API data.	conveys request
weatherController ↔ weatherModel	weatherController calls the weatherModel for API data about the weather.	retrieves data
notificationController ↔ messageModel	notificationController calls the messageModel to properly format a message based on the data.	formats
notificationController ↔ gmailController	notificationController calls the gmailController to send a notification via the gmail channel.	provides data
notificationController ↔ slackBotController	notificationController calls the slackBotController to send a notification via the Slack channel.	provides data
notificationController ↔ discordBotController	notificationController calls the discordBotController to send a notification via the Discord channel.	provides data
scheduleController ↔ scheduledTripsModel	scheduleController calls the scheduledTripsModel to update the notification status as sent.	updates data

### iii. Attribute Definitions

Concept	Attributes	Attribute Description
homeController	staticPath	Used to hold the path to the static file folder.
	scheduledTripsObject	Is an instance of the scheduledTripsModel.
	formSubmissionObject	Holds the user input from the home page form submission.
	inputValidationObject	Is an instance of the inputValidation.
scheduleController	currentTime	Used to track the timing of messages and order the queue.
	scheduledTripsObject	Is one or more instances of the scheduledTripsModel.
	notificationObject	Is one or more instances of the notificationController.
	tripQueue	Contains the upcoming scheduled trip notificationObjects.
notificationController	travelObject	Is an instance of the travelController.
	weatherObject	Is an instance of the weatherController.
	messageObject	Is an instance of the messageModel.
	gmailControllerObject	Is an instance of the needed controller to send to the email and/or SMS channel.
	slackBotControllerObject	Is an optional instance of the needed controller to send to the Slack channel, if the user requests it.
	discordBotController	Is an optional instance of the needed controller to send to the Discord channel, if the user requests it.
travelController	geocodeData	The geocodes used to represent start/end trip locations.
	travelModelObject	Is one instance each created for travelTimeModel and travelIncidentModel.
	travelTimeResponse	The object returned from the Traffic API with travel time.
	tripIncidentResponse	The object returned from the Traffic API with incidents.
weatherController	geocodeData	The geocodes used to represent start/end trip locations.
	tripTimeStart	The time the trip is meant to start.
	weatherModelObject	Is an instance of the weatherModel.



	weatherResponse	The object returned from the Weather API.
messageModel	tripData	Includes the relevant trip data pulled from tripResponse.
	weatherData	Includes the relevant weather data pulled from weatherResponse.
gmailController	formattedMessage	A message formatted for the email or SMS channel.
	transporterObject	Collect details needed to connect to gmail.
	userContact	Email to which email will be sent and/or contact number and carrier for SMS messages.
discordBotController	formattedMessage	A message formatted for the Discord channel.
	userID	Used to direct the message to the correct user in Discord.

**iv. Traceability Matrix (Updated)**

Domain Concepts	Use Case	UC1	UC2	UC3	UC4	UC5	UC6	UC7
	PW	53	23	21	20	12	11	27
homeController		x	x	x	x			
scheduledTripsModel		x	x	x	x			x
externalScheduler						x	x	
scheduleController						x	x	x
notificationController						x	x	x
travelController						x		
weatherController							x	
baseFetchRetry						x	x	
messageModel								x
gmailController								x
slackBotController								x
discordBotController								x
inputValidation		x	x	x	x			

## **\*\* Text Description of Traceability Matrix (Update)**

Our project uses the MVC design pattern. We attempted to thoroughly divide the responsibilities (separation of concerns) in the domains that would be used to fulfill our use cases. As we were ultimately able to implement all of our use cases, our division of domains seemed to fulfill the functional needs of the My Traffic Wizard app.

### **Use Case 1 - Submit Trip Alert Request:**

- homeController: manages user interface display and form submission.
- scheduledTripsModel: model for data transfer to the database.
- inputValidation: checks if user input is valid before accepting input.

### **Use Case 2 - Request Scheduled Alerts via SMS:**

- homeController: manages user interface display and form submission.
- scheduledTripsModel: model for data transfer to the database.
- inputValidation: checks if user input is valid before accepting input.

### **Use Case 3 - Request Scheduled Alerts via Discord:**

- homeController: manages user interface display and form submission.
- scheduledTripsModel: model for data transfer to the database.
- inputValidation: checks if user input is valid before accepting input.

### **Use Case 4 - Request Scheduled Alerts via Slack:**

- homeController: manages user interface display and form submission.
- scheduledTripsModel: model for data transfer to the database.
- inputValidation: checks if user input is valid before accepting input.

**Use Case 5 - Fetch Traffic API Data:**

- externalScheduler: triggers the endpoint of the scheduleController at specified intervals.
- scheduleController: manages upcoming scheduled notifications and manages the queue.
- notificationController: handles the process of creating/sending scheduled notifications.
- travelController: manages traffic API interactions.
- baseFetchRetry: automates repeat requests to REST APIs when error is returned.

**Use Case 6 - Fetch Weather API Data:**

- externalScheduler: triggers the endpoint of the scheduleController at specified intervals.
- scheduleController: manages upcoming scheduled notifications and manages the queue.
- notificationController: handles the process of creating/sending scheduled notifications.
- weatherController: manages weather API interactions.
- baseFetchRetry: automates repeat requests to REST APIs when error is returned.

**Use Case 7 - Send Notification:**

- scheduledTripsModel: model for data transfer to the database.
- scheduleController: manages upcoming scheduled notifications and manages the queue.
- notificationController: handles the process of creating/sending scheduled notifications.
- messageModel: creates messages with data from tripController and weatherController.
- gmailController: handles the sending of messages through the Gmail email account.
- slackbotController: handles the sending of messages to the Slack channel.
- discordController: handles the sending of messages to the Discord channel.

## 7b. System Operation Contracts

Operation	Submit Trip Alert Request
Use Case	UC1
Preconditions	<ul style="list-style-type: none"> <li>● The user interface allows user to enter information and select options</li> <li>● Departure and destination addresses must be valid</li> <li>● Email address is a valid format</li> </ul>
Postconditions	<ul style="list-style-type: none"> <li>● The user's entries are submitted and processed for future use</li> <li>● The notification_status is set to not sent</li> </ul>

Operation	Request Scheduled Alerts via SMS
Use Case	UC2
Preconditions	<ul style="list-style-type: none"> <li>● User has entered trip information and email in appropriate fields</li> <li>● Departure and destination addresses must be valid</li> <li>● Phone number is a valid format</li> </ul>
Postconditions	<ul style="list-style-type: none"> <li>● The user's entries are submitted and processed for future use</li> <li>● The notification_status is set to not sent</li> </ul>

Operation	Request Scheduled Alerts via Discord
Use Case	UC3
Preconditions	<ul style="list-style-type: none"> <li>● User has entered trip information and email in appropriate fields</li> <li>● Departure and destination addresses must be valid</li> <li>● User has joined the My Traffic Wizard server</li> <li>● User has provided their Discord UserID</li> </ul>
Postconditions	<ul style="list-style-type: none"> <li>● The user's entries are submitted and processed for future use</li> <li>● The notification_status is set to not sent</li> </ul>

Operation	Request Scheduled Alerts via Slack
Use Case	UC4
Preconditions	<ul style="list-style-type: none"> <li>● User has entered trip information and email in appropriate fields</li> </ul>

	<ul style="list-style-type: none"> <li>• Departure and destination addresses must be valid</li> <li>• User has joined the My Traffic Wizard workspace</li> <li>• User has provided their Slack UserID</li> </ul>
Postconditions	<ul style="list-style-type: none"> <li>• The user's entries are submitted and processed for future use</li> <li>• The notification_status is set to not sent</li> </ul>

<b>Operation</b>	<b>Fetch Traffic API Data</b>
Use Case	UC5
Preconditions	<ul style="list-style-type: none"> <li>• Scheduling Service has determined timing based on notification schedule</li> <li>• Number of retries, maxRetries &lt; 5</li> <li>• Valid geocodes are available, geoCodeData</li> </ul>
Postconditions	<ul style="list-style-type: none"> <li>• Current traffic and incident data has been retrieved from Traffic API</li> <li>• travelTimeResponse contains data returned from API</li> <li>• tripIncidentResponse contains data returned from API</li> </ul>

<b>Operation</b>	<b>Fetch Weather API Data</b>
Use Case	UC6
Preconditions	<ul style="list-style-type: none"> <li>• Scheduling Service has determined timing based on notification schedule</li> <li>• Number of retries, maxRetries &lt; 5</li> <li>• Valid geocodes are available, geoCodeData</li> </ul>
Postconditions	<ul style="list-style-type: none"> <li>• Weather data has been retrieved from Weather API</li> <li>• weatherResponse contains data returned from API</li> </ul>

<b>Operation</b>	<b>Send Notification</b>
Use Case	UC7
Preconditions	<ul style="list-style-type: none"> <li>• Weather and traffic data has been retrieved to create notification for user</li> <li>• tripData contains the relevant trip data for message</li> <li>• weatherData contains the relevant weather data for message</li> </ul>
Postconditions	<ul style="list-style-type: none"> <li>• Message is sent to the user through their chosen channels</li> <li>• formattedMessage for specific channel is sent</li> <li>• The notification_status is set to sent</li> </ul>

### 7c. Data Model and Persistent Data Storage

The application uses a relational database management system (RDBMS) to store notification details, location data, and contact information specific to the communication channels chosen by the user. We selected PostgreSQL as the RDBMS because it is offered by default by our hosting provider (Render) and provides all the database capabilities needed by our application.

Furthermore, SQL is standardized and known for fast, efficient access. It will allow us to quickly store and retrieve the relevant information we use to craft and send alerts from myTrafficWizard. We are also able to leverage the experience the team has with SQL queries and RDBMS systems.

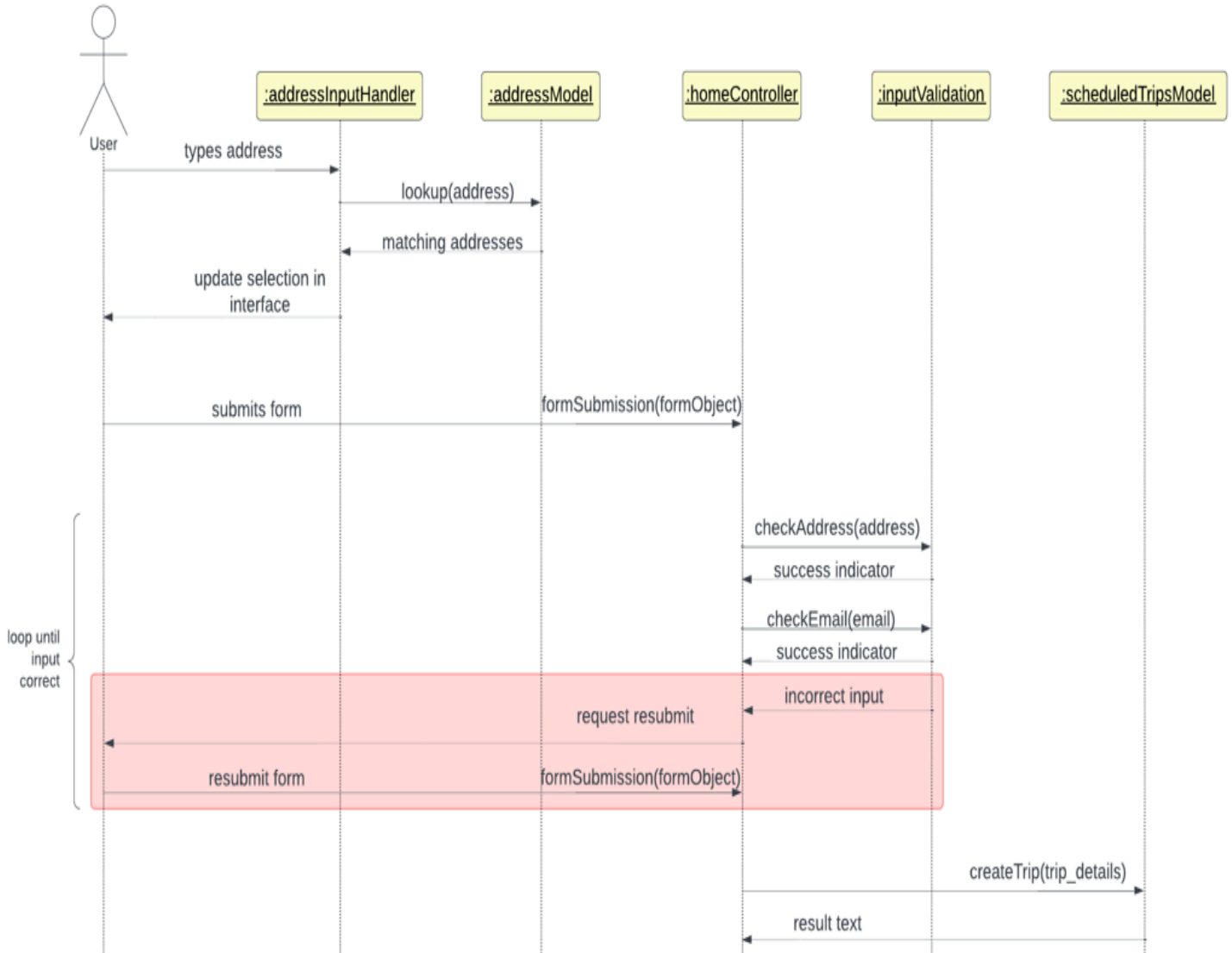
Adhering to the KISS design principle, we decided our data could be modeled in a single monolithic table, shown below:

#### Database Schema

scheduledTrips		
PK	trip_id	varchar
	email_address	varchar
	departure_latitude	varchar
	departure_longitude	varchar
	destination_latitude	varchar
	destination_longitude	varchar
	departure_date	timestamp
	mobile_number	varchar
	mobile_provider	varchar
	user_id_discord	varchar
	user_id_slack	varchar
	created_at	timestamp
	notification_status	varchar

## 8. Interaction Diagrams

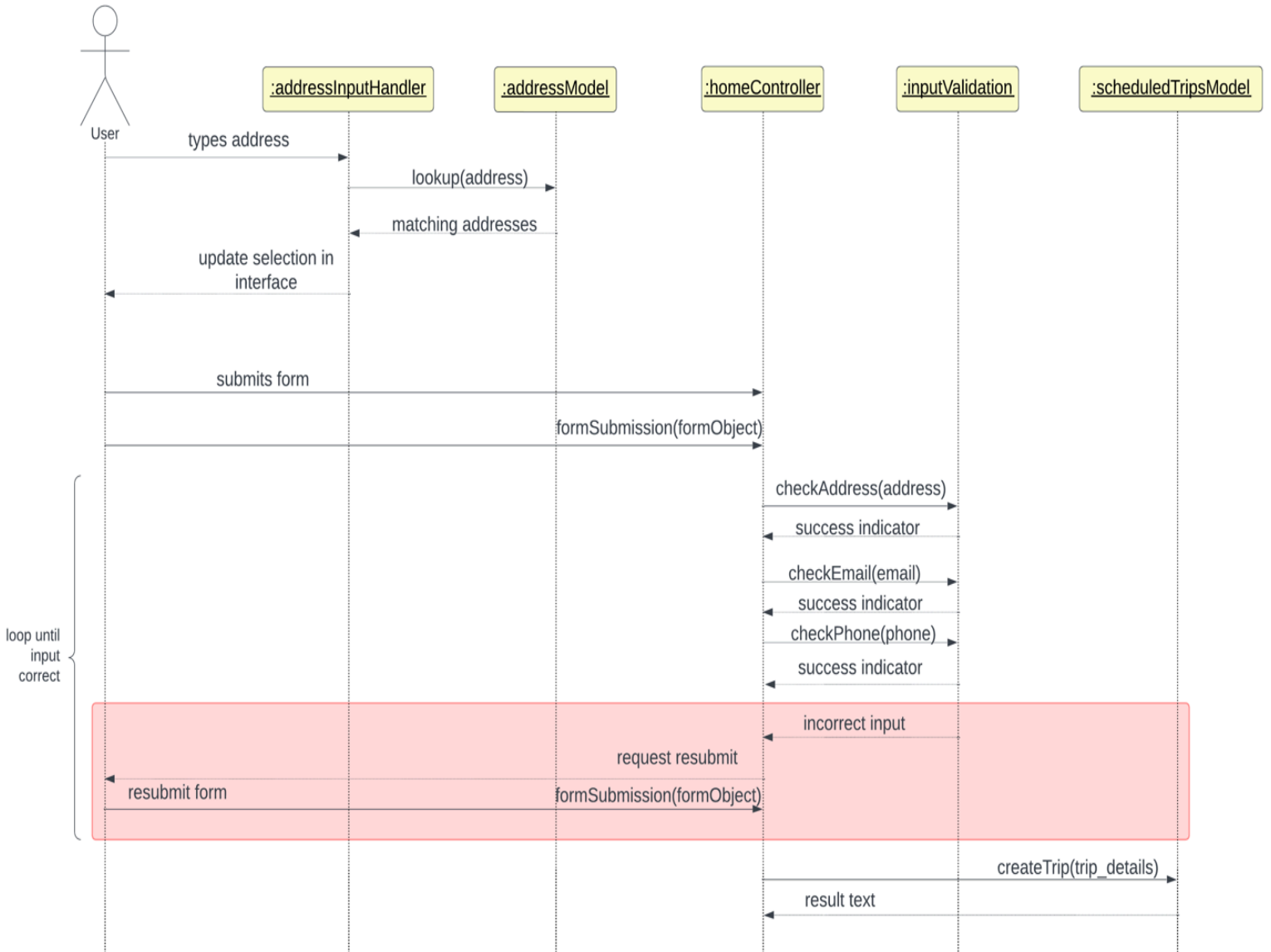
### UC1: Submit Trip Alert Request



The assignment of responsibilities for this use case centers around high cohesion - each component has a well defined role. For example, input validation is done separately from the homeController and the address model focuses on providing the user with an autocomplete-like experience for address inputs.

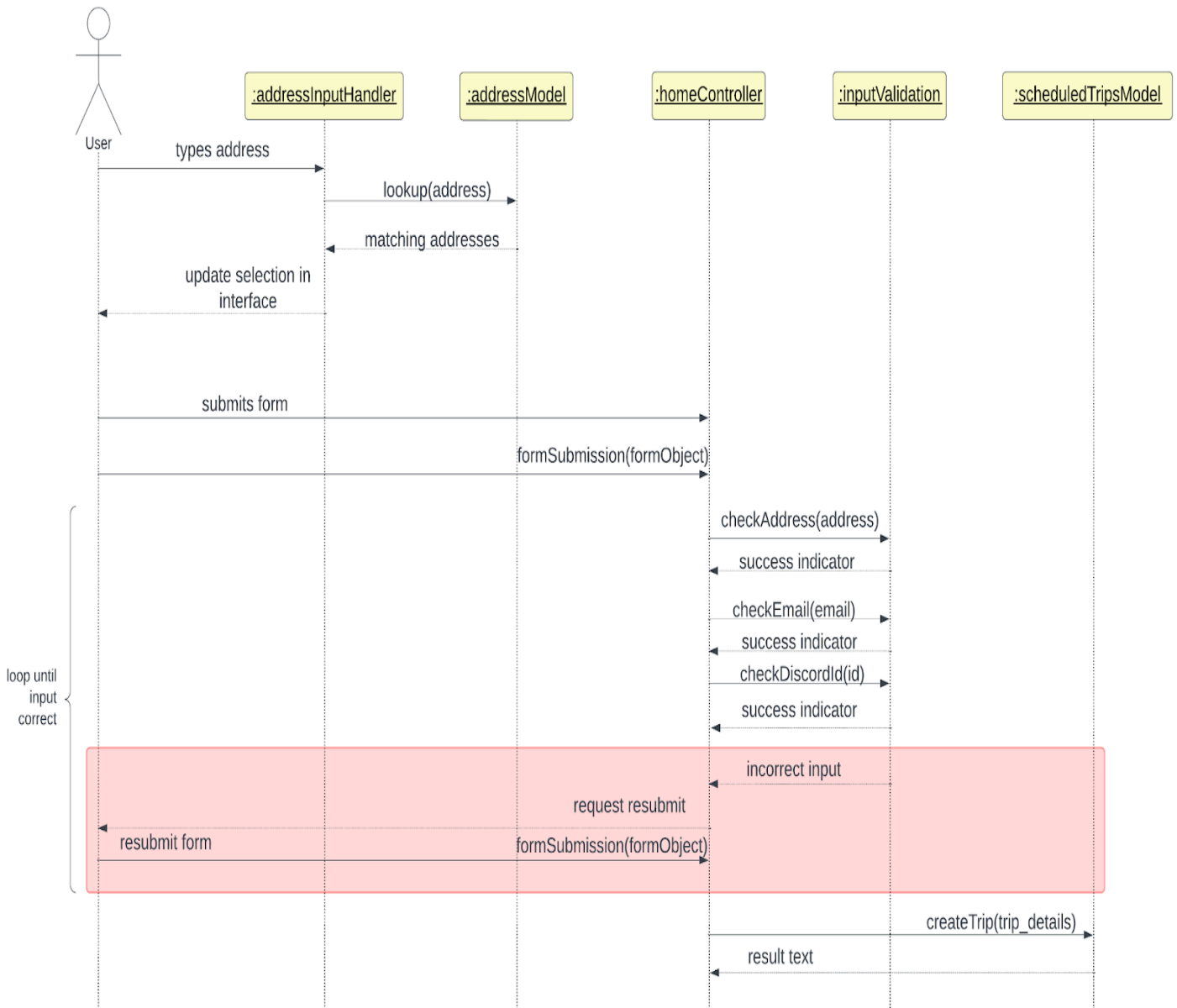


## UC2: Request Scheduled Alerts Via SMS



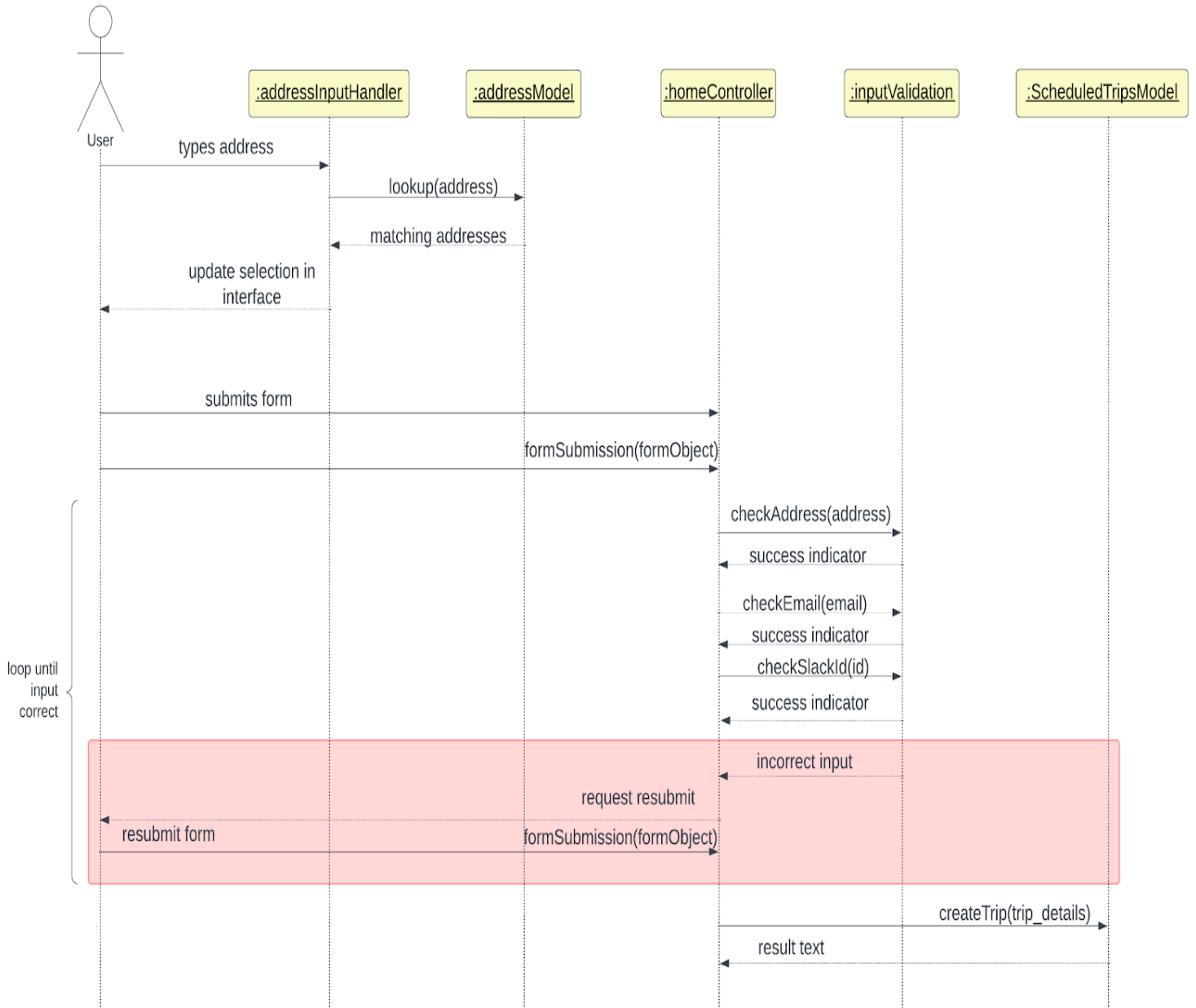
For this sequence, we can highlight the design principle of low coupling. The homeController was assigned duties (collect form data and submit form data to the database) that allows it to act as a mediator between the user interface and the database. This makes the system more modular and easier to extend as requirements progress.

## UC3: Request Scheduled Alerts Via Discord



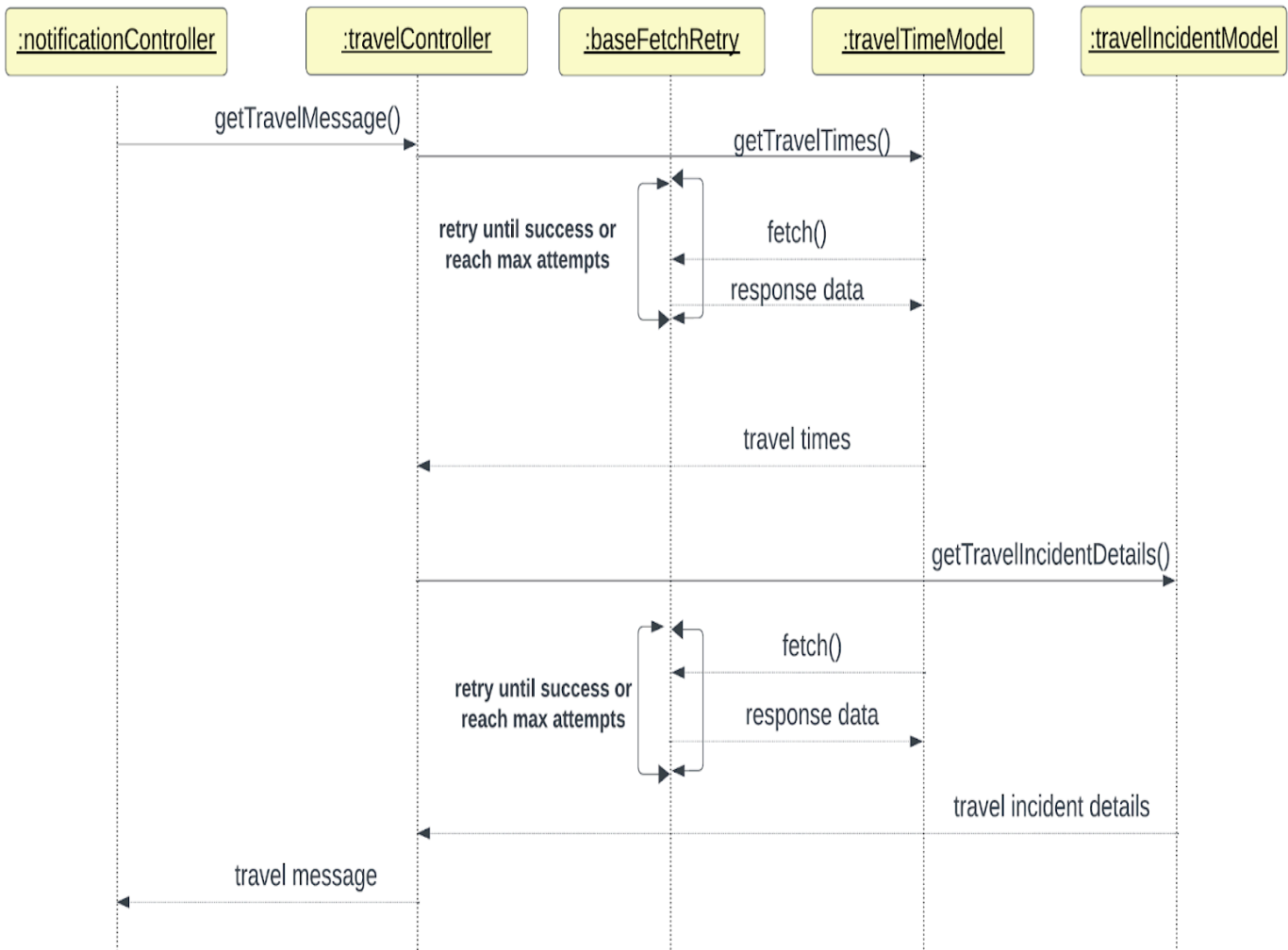
Another important design decision was to include an event handler `:addressInputHandler` to support the address autocomplete feature. Originally, we thought that responsibility may live with the `homeController`, but we decided that this responsibility would be given to an event handler that was tied to the user interface. This decision is rooted in the Expert principle, since the autocomplete workflow is heavily tied to the user interface, it seemed valuable to handle the required events via an event handler that communicates directly with our address model.

# UC4: Request Scheduled Alerts Via Slack



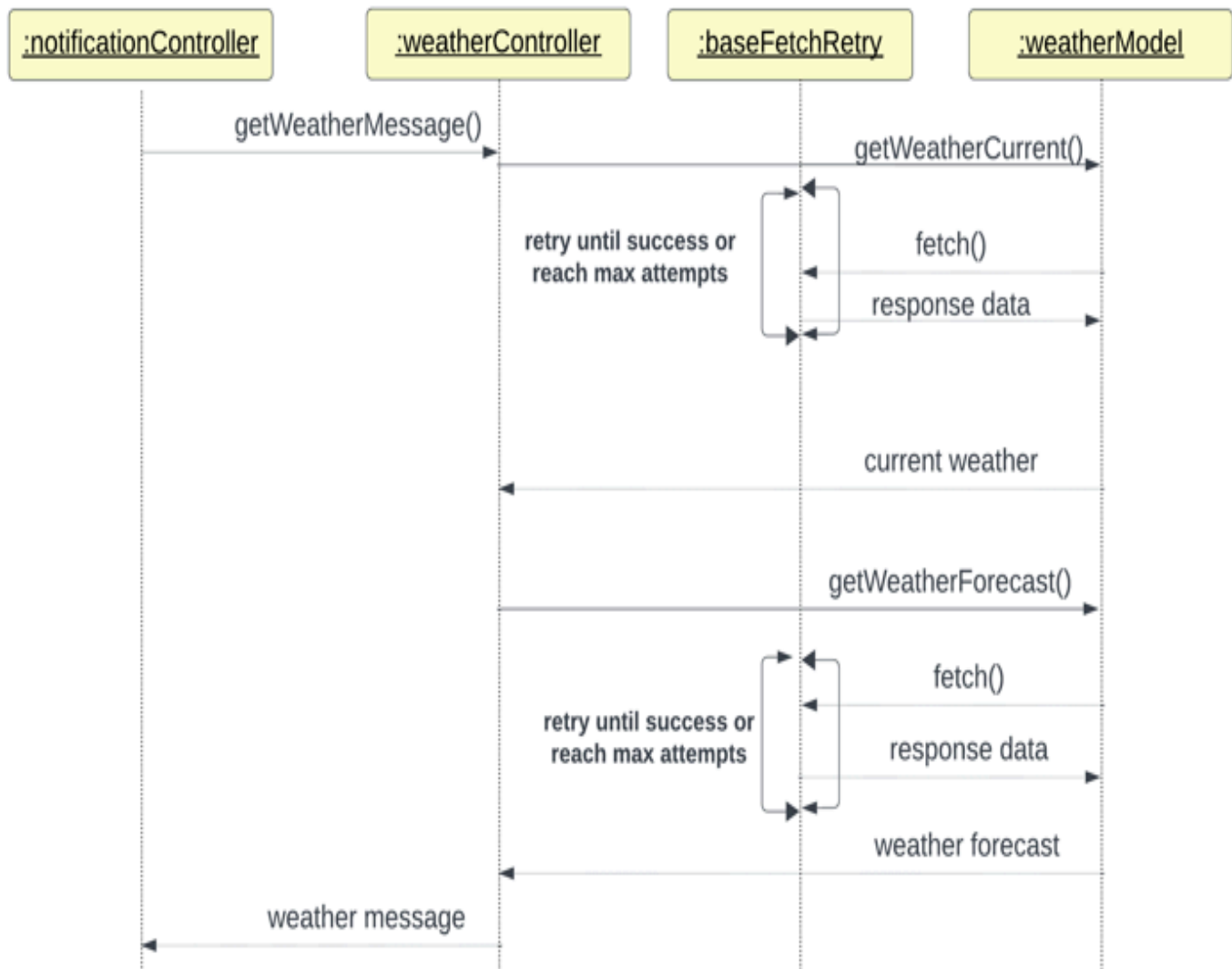
In this sequence, the homeController was given the responsibility of creating ScheduledTripsModel and inputValidation. As the mediator between the database and the user input, it was the ideal candidate to uphold the Creator design principle.

# UC5: Fetch Traffic API Data



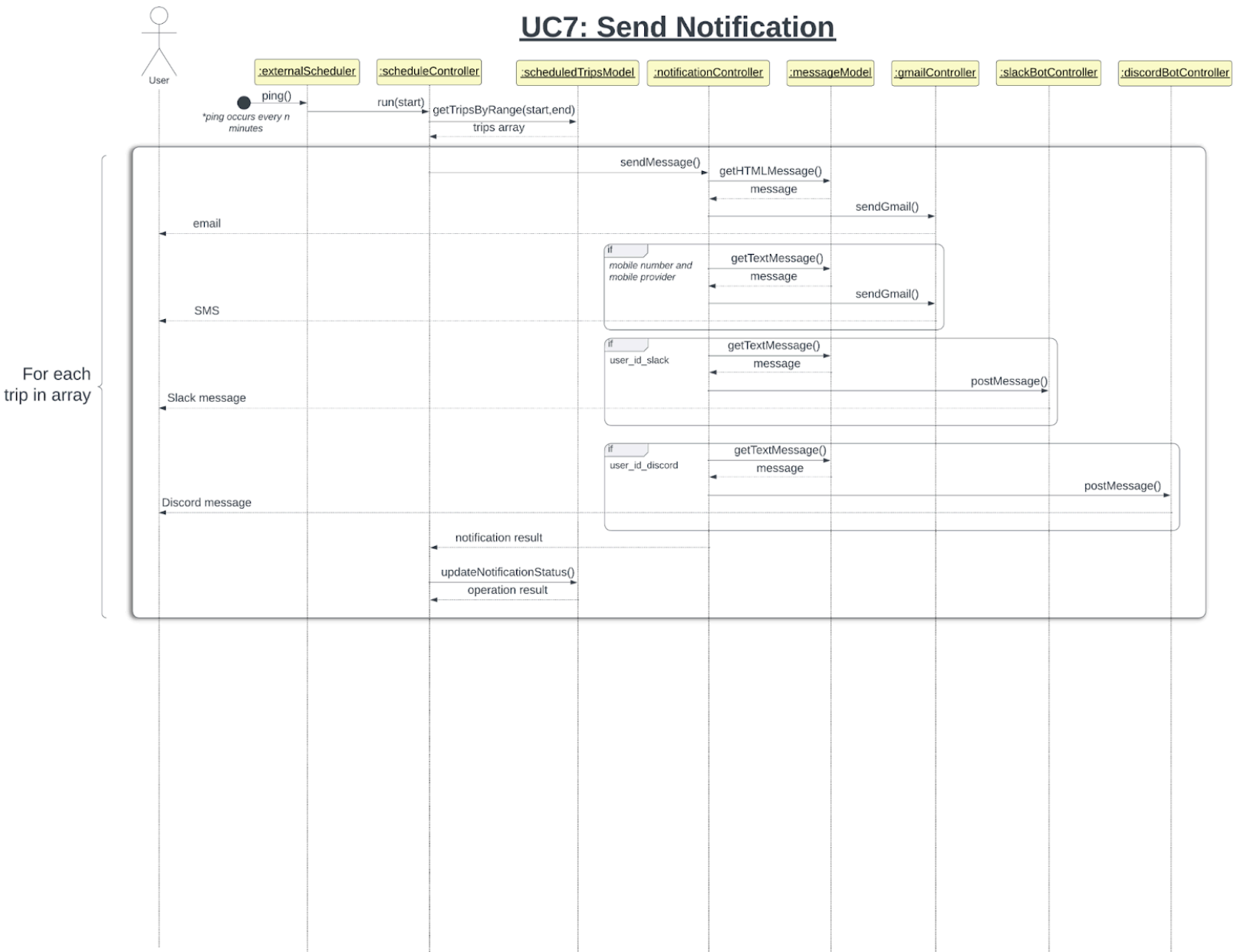
For this sequence, we decided to house the responsibility of requesting API to the models (travel times and travel incidents). Additionally, since we had a need for retries, we assigned any base API requests to the baseFetchRetry class, so the model files could inherit those capabilities without repeating the same logic (DRY principle). The responsibility of formulating the travel message was assigned to the travelController, who can focus on delivering the proper content to the notificationController (who acts as the overall Creator and initiator of the fetch). These assignments were made to increase cohesion; although some dependencies were introduced, we added value by allowing reuse of common components.

## UC6: Fetch Weather API Data



The assignment of responsibilities for this sequence were influenced by our choice to use the MVC software architecture. We assigned the responsibility of coordinating the overall creation of the message to the notificationController via the Creator principle. It instantiates and calls the weatherController to get the weather message. The weatherController then passes the responsibility of actually fetching the API data to the weatherModel which represents the data layer in our workflow. Overall, we chose to have each component have a sole responsibility and communicate with others when necessary. However, the communication links for this sequence are fairly short.

## UC7: Send Notification



*\*This image contains many interactions, you will need to zoom in for the best display.*

In this sequence, we assigned responsibilities based on role to align with high cohesion. For example, we created controllers for each notification channel; this allows a central notificationController to use those components to send notifications, instead of taking on those responsibilities for itself. We again rely on the Creator principle for the notificationController, which creates the other controllers. The responsibility for formatting and creating a deliverable message was given to the messageModel.

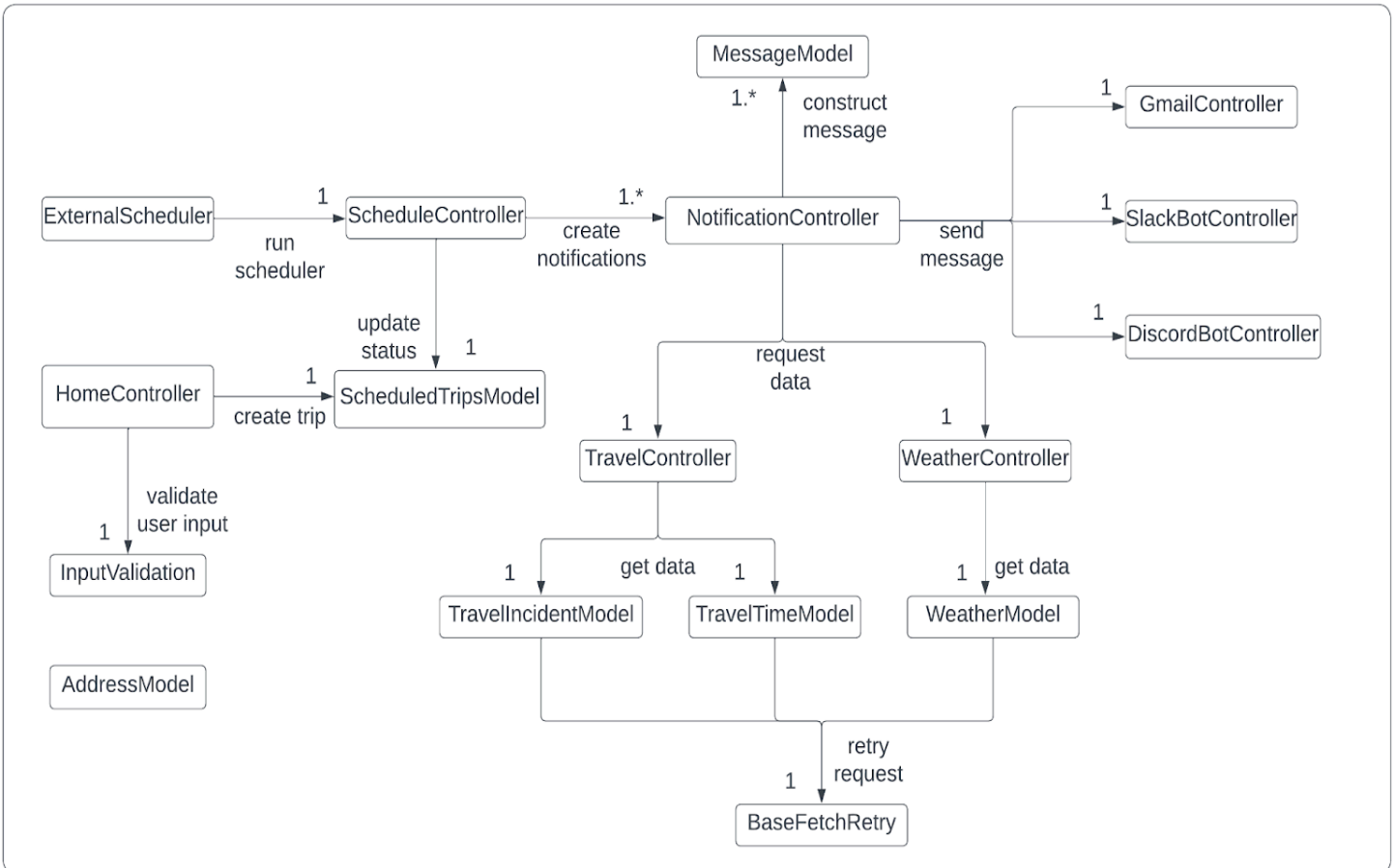
Although each channel relies on the messageModel indirectly, we were able to assign the responsibility of interacting with the messageModel to the notificationController, which decreases the messaging links in the interaction. Overall, we assigned responsibilities based on the MVC architecture and the principles: high cohesion, low coupling and Expert Doer (i.e. the controller for each channel is dedicated to that media).

## 9. Class Diagram and Interface Specification (Updated)

### 9a. Class Diagram (Updated)

Original Class Diagram:

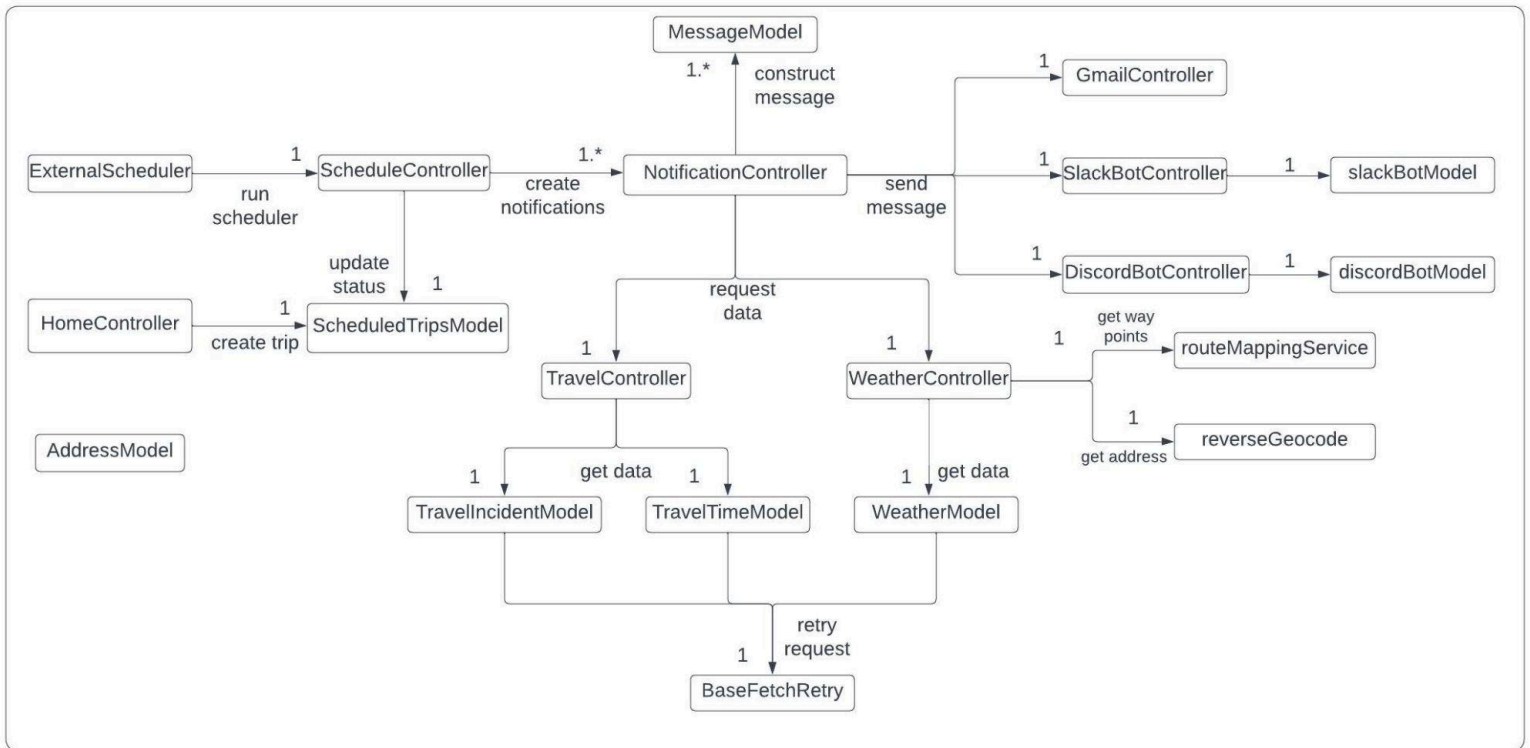
### Class Diagram: Overview





## Final Class Diagram:

### Class Diagram: Overview



### Class Diagram Explanation:

This section clarifies the architecture and design decisions that were incorporated when creating the classes in our system.

### MVC Architecture

The MVC pattern enables our team to work on the user interface, business logic, and data model independently, increasing productivity (parallel development) and decreasing coupling across our classes. When a change is made to a class, it typically impacts only part of the system and it is easy to visualize how the system works as a whole since the parts are organized.

## Key Class Responsibilities and Interactions

### HomeController:

This class serves as the system's entry point for the user, handling the user interface, processing input, and communicating with the database for data persistence.

### ScheduleController:

In our initial phase, we decided we wanted a class with the responsibility of "scheduling" the sending of the notifications. This concept evolved into the ScheduleController, which retrieves upcoming trips and delegates the task of notification management to instances of the NotificationController.

### NotificationController, GmailController, DiscordController, SlackController:

Adhering to the single responsibility principle, we allocated the task of message dispatch to specific controllers (i.e. DiscordController). The NotificationController assumes a supervisory role, managing the multichannel notification delivery process, which has significantly enhanced our system's modularity.

### TravelController and WeatherController

These classes preprocess the API data - refining raw inputs into structure formats suitable for consumption of a human. This design choice cleanly delineates data curation from retrieval and dissemination.

### SlackBotModel, DiscordBotModel, TravelIncidentModel, TravelTimeModel, WeatherModel, ScheduledTripsModel:

These models were given the main responsibility of retrieving data, storing data (in a database) and propagating data through a channel

### Address Model

An innovative addition to enhance the user experience and cleanliness of address inputs, the AddressModel is directly integrated with the user interface. It dynamically interfaces with an external API to offer address suggestions in real time, independent of backend processes.

### **Class Diagram Evolution:**

The following changes were made to the class diagram

- Remove InputValidation class - the functionality of this class was moved to the user interface
- Add RouteMappingService - this class inherited the responsibility of calculating waypoint from the weatherController
- Add ReverseGeocode - this class inherited the responsibility of finding addresses for geocodes from the weatherController
- Add slackBotModel and discordBotModel - these concepts existed in our original design but were implemented in a later iteration.

### **Class Diagram Evolution from Domain Model:**

The domain model provided an initial high-level abstraction of the core entities and their relationships within our system. It served as a blueprint, capturing the essence of the problem domain without delving into the specifics of the implementation.

Yet, as we ventured into crafting the class diagrams, our mental model began to crystallize, shifting from ideation to concrete implementations.. Noteworthy evolutions in our class diagram design process include:

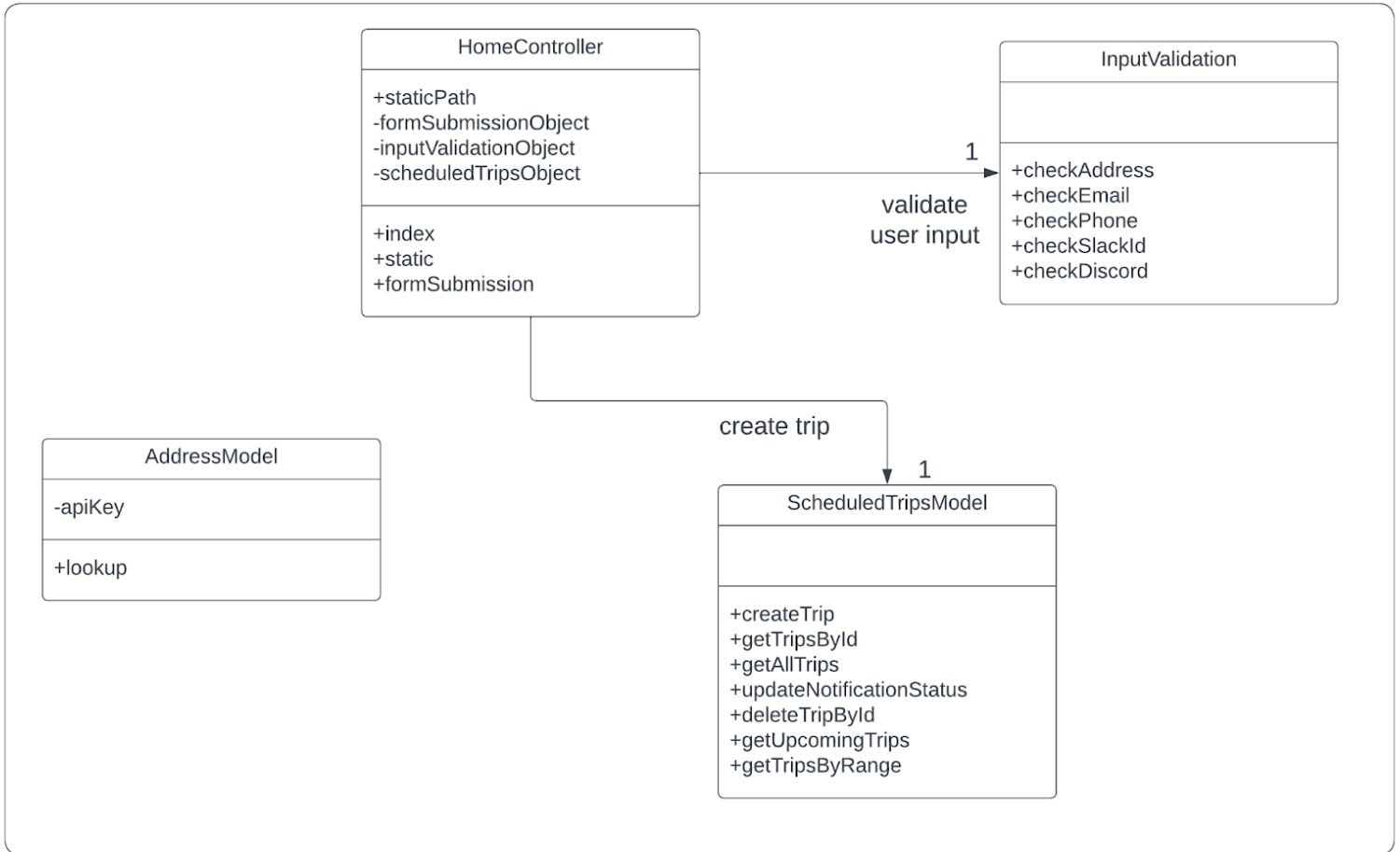
- Adding models for the DiscordBotController and SlackBotController which separates the concerns between interacting with the API and managing the sending of messages
- Changing the address model's points of activity from backend to front-end
- Creating helper services - RouteMappingService and ReverseGeocode to assist the WeatherController in carrying out its duties, but not overburdening it with responsibilities that are not core to its function
- Including a BaseFetchRetry class that provides a basic retry functionality for all models using an API to retrieve data on the backend
- Adding methods and properties to make the classes more testable

It is imperative to acknowledge that the evolution from the domain model to the class diagram was rooted in implementation rather than a transformation of core concepts or relationships. The

team's initial vision was well-conceived, allowing for a seamless transition into the system's design and architecture.

## Original Partial Class Diagram: Submit Trip

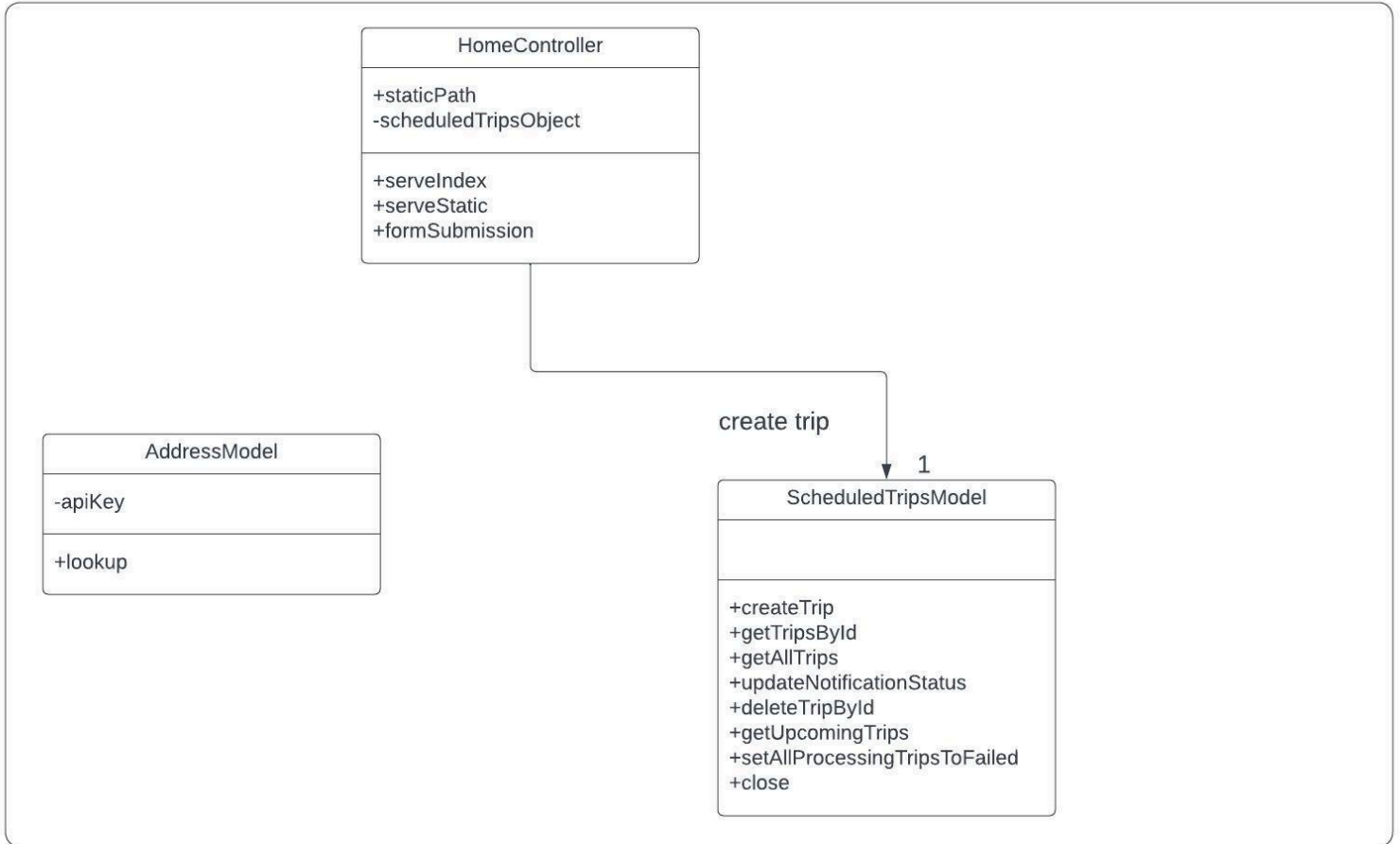
# Partial Class Diagram: Submit Trip



*\*This component represents the classes and associations involved in submitting a trip. The AddressModel appears to have no associations because it interacts directly with the an event handler tied to the user interface.*

## Final Partial Class Diagram: Submit Trip

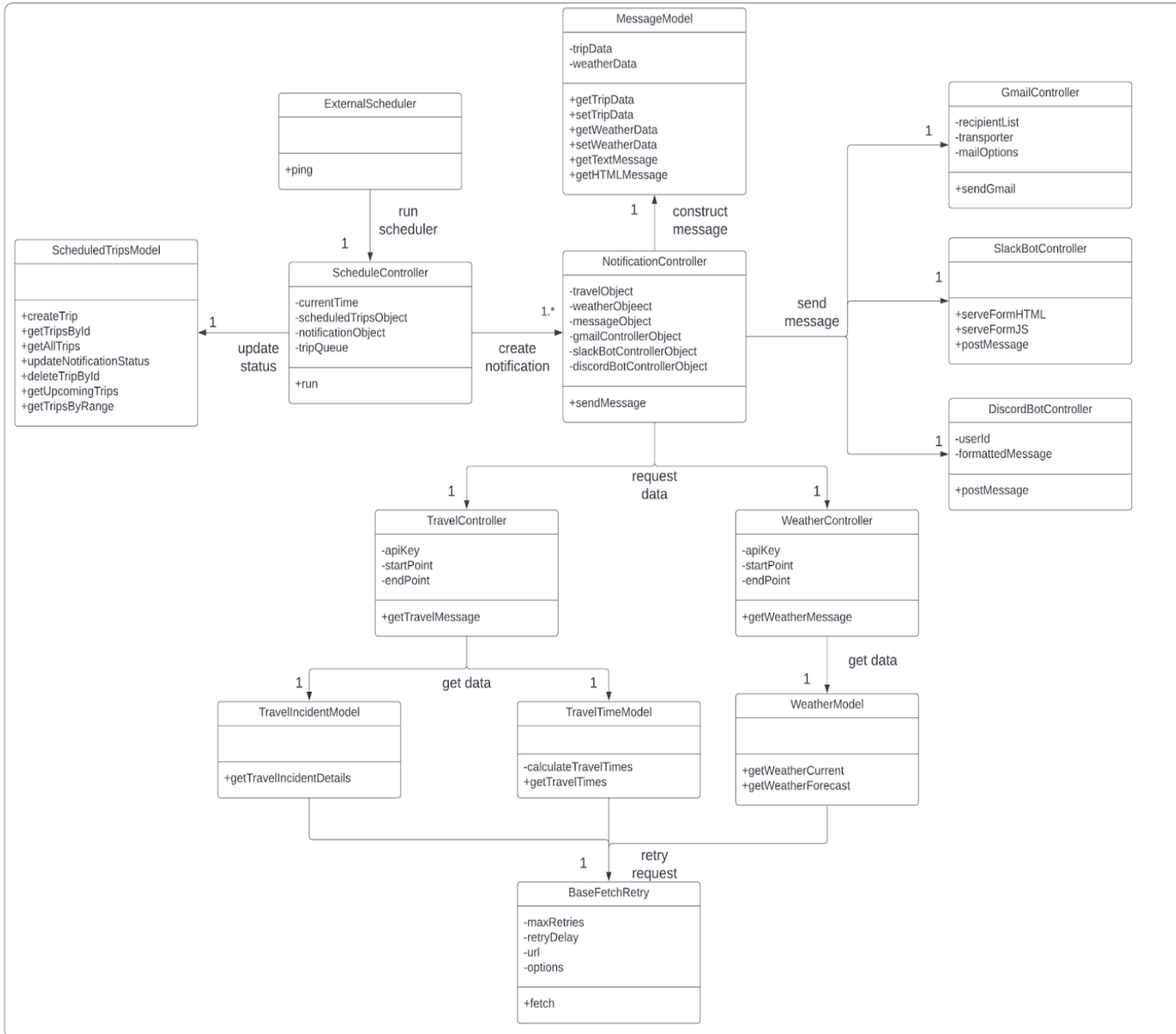
### Partial Class Diagram: Submit Trip



*\*This component represents the classes and associations involved in submitting a trip. The AddressModel appears to have no associations because it interacts directly with the an event handler tied to the user interface.*

## Original Partial Class Diagram: Send Notification

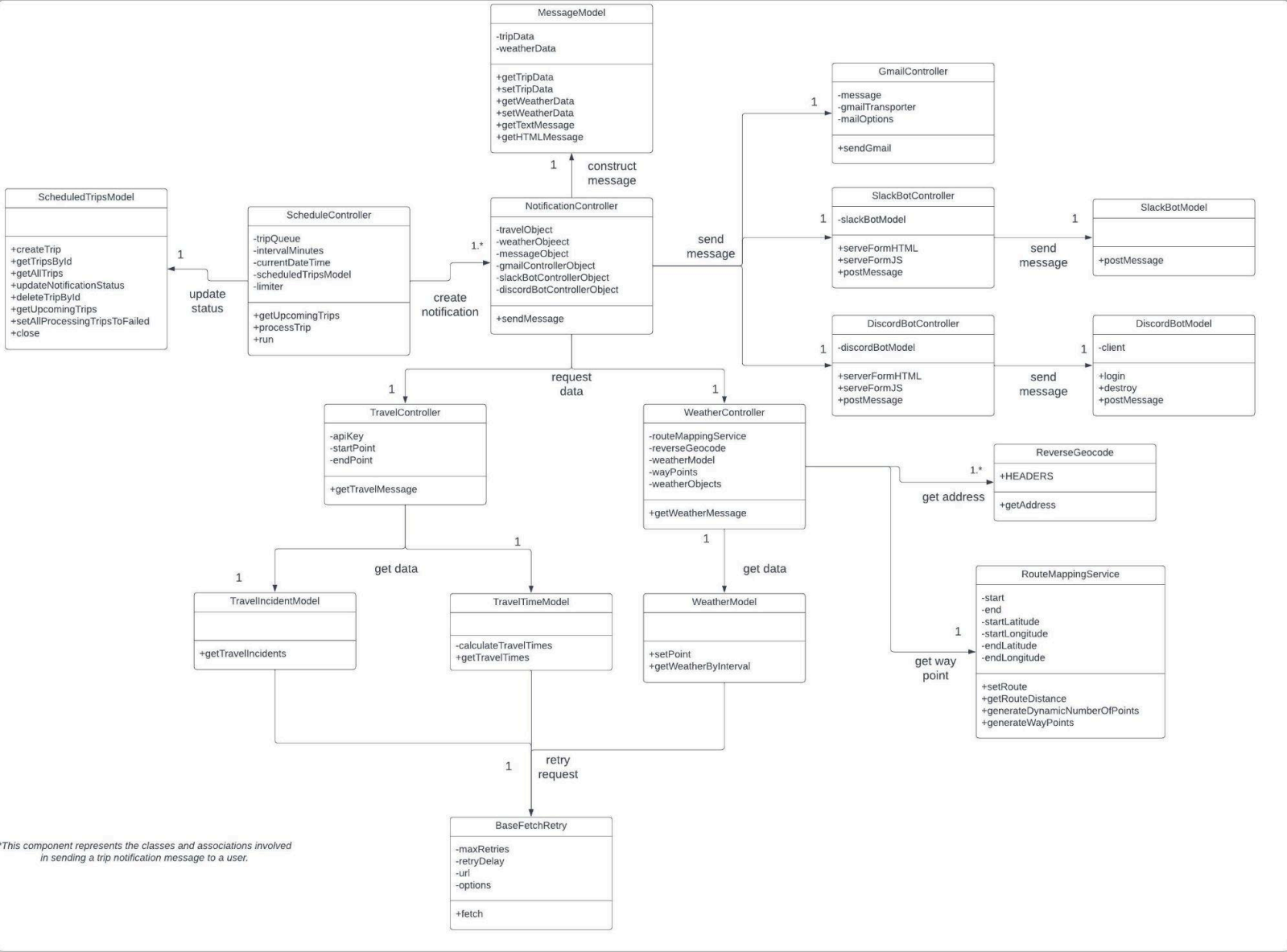
### Partial Class Diagram: Send Notification



\*This component represents the classes and associations involved in sending a trip notification message to a user.

# Final Partial Class Diagram: Send Notification

## Partial Class Diagram: Send Notification





## 9b. Data Types and Operation Signatures (Updated)

**\*\*Update Note:** the InputValidation class was removed due to this functionality ultimately being implemented within the front-end code. Removed attributes/operations have a gray highlight applied, while added ones have a green highlight applied.

**\*Class: InputValidation** - entire class removed in favor of front-end validation

Attributes:

None

Operations:

checkAddress(address: string) : boolean - simple regex pattern check

\*Address lookup through TomTom replaced this validation

checkEmail(email: string) : boolean - simple regex pattern check

checkPhone(phone: string) : boolean - simple regex pattern check

checkSlackId(id: string) : boolean - simple regex pattern check

checkDiscordId(id: string) : boolean - simple regex pattern check

Meaning:

This class checks user input for validity before accepting input.

<b>InputValidation</b>
+checkAddress(address: string) : boolean +checkEmail(email: string) : boolean +checkPhone(phone: string) : boolean +checkSlackId(id: string): boolean +checkDiscordId(id: string) : boolean

**Class: HomeController**

Attributes:

staticPath : Object - finds path to static files folder using a middleware function

scheduledTripsObject: DatabaseConnection - allows database access

formSubmissionObject: FormData - an object with user form entries

inputValidationObject : InputValidation - an instance of inputValidation

\*Input validation was moved to the front end

Operations:

serveIndex(req: Object, res: Object) : void - serves the index.html page

serveStatic(req: Object, res: Object, next: Function) : void - serves the static files

formSubmission(formSubmissionObject : FormData) : void - accepts user form input

Meaning:

This class serves the user interface and passes user input to the database.

<b>HomeController</b>
+staticPath : Object -scheduledTripsObject: DatabaseConnection -formSubmissionObject: FormData -inputValidationObject : InputValidation
+serveIndex(req: Object, res: Object) : void +serveStatic(req: Object, res: Object, next: Function) : void +formSubmission(formSubmissionObject: FormData) : void

### **Class: AddressModel**

Attributes:

apiKey: string - API key for using TomTom

Operations:

lookup(address: string) : Object - sends a request to TomTom for addresses that closely match the user input

Meaning:

This class provides address autocomplete functionality.

AddressModel
-apiKey : string
+lookup(address: string) : Object

### Class: ScheduledTripsModel

Attributes:

None

Operations:

createTrip(trip\_details: Object): Object- adds a new trip to the database

getTripById(id: string): Object - retrieves a trip's details using its trip\_id

getAllTrips(): Object- retrieves all scheduled trips from the database

updateNotificationStatus(id: string, status: string): Object- updates the notification status of an existing trip using its trip\_id

deleteTripById(id: string): Object- deletes a trip from the database using its trip\_id

getUpcomingTrips(inputDate: Date, o): Object - retrieves all trips that are scheduled for a future departure date

setAllProcessingTripsToFailed(): Object - set the notification status for all trips that were in the processing state, to a failed state

close(): void - closes the database connections

Meaning:

This class represents the data layer between the database and the rest of the system.

ScheduledTripsModel
+createTrip(trip_details: Object): Object
+getTripById(id: string): Object
+getAllTrips(): Object
+getTripsById(id: string): Object
+updateNotificationStatus(id: string, status: string): Object
+deleteTripById(id: string): Object
+getUpcomingTrips(inputDate: Date, offsetMinutes: string): Object
+setAllProcessingTripsToFailed(): Object

```
+close(): void
```

## Class: ScheduleController

### Attributes:

currentTime: Timestamp - activation time used to search for upcoming notifications

intervalMinutes: the interval used to find upcoming trips

scheduledTripsObject: ScheduledTripsModel - model to access database

notificationObject: NotificationController - controls processing of each notification

tripQueue : List<Object> - holds upcoming trip details

limiter: Object - defines the bottleneck to use for the asynchronous run of the scheduler

### Operations:

run(produceStatusReport: bool) : Promise - runs the process to dispatch and handle tasks

getCurrentDateTime(): Date - returns date representing starting time of the scheduler

closeDatabaseConnection(): void - allows caller to close database connection

getUpcomingTrips(close: bool): List<Object>: loads upcoming trips into trip queue

updateStatus(tripId: string, status: string): void - updates notification status for given id

processTrip(trip: string): Object - processes an individual trip to send notifications

logStatusReport(statusReport: List): string - logs status report to the console

### Meaning:

This class controls the upcoming scheduled notifications and manages the queue.

ScheduleController
-currentTime: Date
-intervalMinutes: string
-scheduledTripsObject: ScheduledTripsModel
-tripQueue : List<Object>
-limiter: Object

```
+run(produceStatusReport: bool) : Promise
+getCurrentDateTime(): Date
+closeDatabaseConnection(): void
-getUpcomingTrips(close: bool): List<Object>
-updateStatus(tripId: string, status: string): void
-processTrip(trip: string): Object
-dispatchTripProcessingTasks(): Promise
-processResults(results: List): List
-logStatusReport(statusReport: List): string
```

### **Class: NotificationController**

Attributes:

travelObject : TravelController - controls travel API interaction  
weatherObject: WeatherController - controls weather API interaction  
messageObject : MessageModel - provides formatted messages  
gmailControllerObject : GmailController - for email/SMS notification sending  
slackBotControllerObject : SlackBotController - for Slack notification sending  
discordBotControllerObject : DiscordBotController - for Discord notification sending  
tripData : object - holds trip data from database for a given trip

Operations:

instantiateTravelObject(tripData : object) : void - creates new TravelController object  
instantiateWeatherObject(tripData : object) : void - creates new WeatherController object  
createMessageObject(travelData : String, weatherData : String) : void - creates new  
MessageModel Object  
createGmailObject(emailAddress : String, messageObject : MessageModel) : void -  
creates new GmailController object  
createDiscordObject() : void - creates new DiscordBotController object  
createSlackObject() : void - creates new SlackBotController object  
getMessageObject() : MessageModel - returns MessageModel object  
getTripData() : object - returns tripData  
fetchTravelData() : TravelController - returns TravelController  
fetchWeatherData(): WeatherController - returns WeatherController  
sendGmail() : GmailController - returns GmailController

postSlackMessage(userId : String, message: String) : object - sends Slack Message  
 postDiscordMessage(userId : String, message: String) : object - sends Discord Message  
 sendMessage() : object - starts the process of controlling other objects

Meaning:

This class controls the process of creating/sending each individual scheduled notification.

<b>NotificationController</b>
-travelObject : TravelController -weatherObject: WeatherController -messageObject : MessageModel -gmailControllerObject : GmailController -slackBotControllerObject : SlackBotController -discordBotControllerObject : DiscordBotController -tripData : object
-instantiateTravelObject(tripData : object) : void -instantiate WeatherObject(tripData : object) : void +createMessageObject(travelData : String, weatherData : String) : void +createGmailObject(emailAddress : String, messageObject : MessageModel) : void +createDiscordObject() : void +createSlackObject() : void +getMessageObject() : MessageModel +getTripData() : object +fetchTravelData() : TravelController +fetchWeatherData(): WeatherController +sendGmail() : GmailController +postSlackMessage(userId : String, message: String) : object +postDiscordMessage(userId : String, message: String) : object +sendMessage() : integer

### **Class: TravelController**

Attributes:

apiKey : string - stores the API key

start: string - represents the start point of the route

end: string - represents the end point of the route

travelTimeModel: Object - object representing travel time for the route

travelIncidentModel: Object - object representing travel incidents for the route

reverseGeocode: Object - object that has functionality of retrieving address information

Operations:

getPluralityOfMinutes(minutes: int): string - helper method that will conditionally pluralize string

isSuccessMessage(message: Object): bool - helper method to check if object represents a successful message, used with data retrieved from models

hasTravelIncidents(message: Object): bool - helper method to check if object has any travel incident data

generateMessageTemplateTravelTime(messageType: string): string - creates a pretty printed representation of the travel time in either HTML or text

generateIncidentHTMLTemplate(travelIncidents: List): string - creates a pretty printed representation of the travel incidents in HTML format

generateIncidentTextTemplate(travelIncidents: List): string - creates a pretty printed representation of the travel incidents in text format

generateMessageTemplateTravelConditions(messageType: string): string - utilizes the HTML and text helper functions to conditionally create a pretty printed representation of the travel incidents in either text HTML format

generateMessageTemplateCombined(messageType: string): string - creates a pretty printed representation of the trip that includes both travel times and incidents along the route

getTravelMessage(): object - returns an object with travel data in html and text formats.

Meaning:

This class generates the text for the travel time and incident information that will be used in the user notification.

<b>TravelController</b>
-apiKey : string -start: string -end: string +travelTimeModel: Object +travelIncidentModel: Object +reverseGeocode: Object

```
-getPluralityOfMinutes(minutes: int): string
-isSuccessMessage(message: Object): bool
-hasTravelIncidents(message: Object): bool
-generateMessageTemplateTravelTime(messageType: string): string
-generateIncidentHTMLTemplate(travelIncidents: List): string
-generateIncidentTextTemplate(travelIncidents: List): string
-generateMessageTemplateTravelConditions(messageType: string): string
-generateMessageTemplateCombined(messageType: string): string
+getTravelMessage(): Object
```

### **Class: WeatherController**

Attributes:

apiKey : string - stores the API key

startPoint: string - represents the start point of the route

endPoint: string - represents the end point of the route

routeMappingService: Object - holds functionality for route mapping

reverseGeocode: Object - holds functionality for converting a geocode to an address using a lookup

weatherModel: Objects - holds functionality for retrieving weather data for a specific point

wayPoints: List - stores a list of waypoints for the route

weatherObjects: List - stores a list of weatherObjects for a route, data includes current weather, weather 1 hour and 3 hours out and an address translation

Operations:

setWayPoints(): void - uses the routeMapping service to set the waypoints for the route

delay(milliseconds: int): Promise - creates a way to cause a delay, utilized to meet requirements for API rate limiting



`generateWeatherObjects(): void` - sets list of `weatherObjects` for the route, `weatherObjects` includes current weather, forecasted weather and address information for each point along the route

`generateTemplateText(): string` - creates a pretty printed representation of the weather data in text form

`generateTemplateHTML(): string` - creates a pretty printed representation of the weather data in HTML form

`getWeatherMessage(): string` - returns a templated string with values corresponding to a trip instance

Meaning:

This class generates the text for the weather information that will be used in the user notification.

<b>WeatherController</b>
<code>-apiKey : string</code> <code>-startPoint: string</code> <code>-endPoint: string</code> <code>-routeMappingService: Object</code> <code>-reverseGeocode: Object</code> <code>-weatherModel: Object</code> <code>-wayPoints: List</code> <code>-weatherObjects: List</code>
<code>-setWayPoints(): void</code> <code>-delay(milliseconds: int): Promise</code> <code>-generateWeatherObjects(): void</code> <code>-generateTemplateText(): string</code> <code>-generateTemplateHTML(): string</code> <code>-formatWeather(weather: Object): string</code> <code>-getWeatherSymbols(weatherCode: string): string</code> <code>+getWeatherMessage(): string</code>

## Class: WeatherModel

### Attributes:

VERSION: string - stores the version number for the API

HEADERS: Object - stores the request's header information

timelines: List - stores timeline data for the point, this includes data for current and forecasted weather

apiKey: string - stores API key for the Tomorrow IO API

weatherCode: Object - stores lookup data to translate weather codes to text descriptions, these codes are defined by Tomorrow IO

### Operations:

getWeatherCurrent(coordinates: string): Object - returns the current weather for a given coordinate (lat/long)

getWeatherForecast(coordinates: string): Object - returns the forecasted weather for a given coordinate

celsiusToFahrenheit(celsius: int): float - converts celsius to fahrenheit

messageOperationSuccess(data: Object, message: string): Object - used to send a standardized success message when the call to the API is successful

messageOperationFailure(error: Object) - used to send a standardized failure message when the call to the API fails

validateInterval(type: string, number: int): int - enforces interval constraints

setPoint(point: string): void - allows caller to set a new point, different from the point the instance was instantiated with

getTimelines(): void - makes a call to the API to get weather information across current and forecasted timelines, data is stored as an attribute

getWeatherByInterval(type: string, number: int): Object: returns a message containing the weather data for the specified point, caller is able to pass a type (minutely or hourly) and number (that represents that quantity of the type)

### Meaning:

This class provides forecasted and current weather data for a trip.

<b>WeatherModel</b>
+VERSION: string +HEADERS: Object -timelines: List -apiKey: string -weatherCode: Object
+getWeatherCurrent(coordinates: string): Object +getWeatherForecast(coordinates: string): Object -celsiusToFahrenheit(celsius: int): float -messageOperationSuccess(data: Object, message: string): Object -messageOperationFailure(error: Object) -validateInterval(type: string, number: int): int +setPoint(point: string): void -getTimelines(): void +getWeatherByInterval(type: string, number: int): Object

### **Class: TravelTimeModel**

Attributes:

TRAVEL\_MODE: string - stores the travel mode parameter for the API

COMPUTE\_TRAVEL\_TIME\_FOR: string - stores the parameter for what the travel time is computer for, this is an API parameter

VERSION\_NUMBER: int - stores the version number for the API endpoint

ROUTE\_REPRESENTATION: string - stores the route representation parameter for the API

HEADERS: Object - stores the HTTP header information

Operations:

messageOperationSuccess(data: Object, message: string): Object - used to send a standardized success message when the call to the API is successful

messageOperationFailure(error: Object) - used to send a standardized failure message when the call to the API fails

calculateTravelTimes(): Object - calculates the travel time for route

getTravelTimes(): Object - provides travel time details for the route of the particular instance

Meaning:

This class provides travel time data for a trip.

<b>TravelTimeModel</b>
+TRAVEL_MODE: string +COMPUTE_TRAVEL_TIME_FOR: string +VERSION_NUMBER: int +ROUTE_REPRESENTATION: string +HEADERS: Object
-messageOperationSuccess(data: Object, message: string): Object -messageOperationFailure(error: Object) -calculateTravelTimes(): Object +getTravelTimes(): Object

### **Class: TravelIncidentModel**

Attributes:

VERSION\_NUMBER: int - stores the version number for the API endpoint

HEADERS: Object - stores the HTTP header information

ICON\_CATEGORY\_MAP: Object - stores the mapping from incident category code to a text description

MAGNITUDE\_MAP: Object - stores mapping from incident magnitude code to a text description

SEVERITY\_SORT\_ORDER: Object - stores sort order for incidents based on magnitude categories

Operations:

messageOperationSuccess(data: Object, message: string): Object - used to send a standardized success message when the call to the API is successful

messageOperationFailure(error: Object) - used to send a standardized failure message when the call to the API fails

sortIncidentsBySeverityDescending(incidents: List): List - sorts the given incidents list by severity, utilized later to return top 5 incidents

getTravelIncidentDetails(): Object - provides travel incident details for the route of the particular instance

Meaning:

This class provides travel incident data for a trip.

TravelIncidentModel
+VERSION_NUMBER: int +HEADERS: Object +ICON_CATEGORY_MAP: Object +MAGNITUDE_MAP: Object +SEVERITY_SORT_ORDER: Object
-messageOperationSuccess(data: Object, message: string): Object -messageOperationFailure(error: Object) -sortIncidentsBySeverityDescending(incidents: List): List +getTravelIncidents(): Object

### Class: BaseFetchRetry

Attributes:

None

Operations:

isNetworkOrServerError(statusCode: string): bool - tests whether a status code is related to a network or server error

delay(milliseconds: int): Promise - configures a delay between retries

fetchWithRetry(): Object - attempts to execute a request with retry logic and returns the response if successful

Meaning:

This class serves as a template for all other classes that will make requests to REST APIs as they all share a need to retry requests when certain status codes are returned.

<b>BaseFetchRetry</b>
<pre> +isNetworkOrServerError(statusCode: string): bool +delay(milliseconds: int): Promise +fetchWithRetry(): Object </pre>

### **Class: MessageModel**

Attributes:

tripData : string - the traffic information for the trip from the travelController  
weatherData : string - the weather information for the trip from the weatherController

Operations:

getTripData() : string - a getter that returns the string stored in the private variable

tripData

setTripData(tripData : string) - a setter that replaces the data stored in the private variable

tripData with the string, tripData, passed to it

getWeatherData() : string - a getter that returns the string stored in the private variable

weatherData

setWeatherData(weatherData : string) - a setter that replaces the data stored in the private

variable weatherData with the string, weatherData, passed to it

getTextMessage() : string - a message created from the relevant data from tripController

and weatherController

getHTMLMessage() : string - a message created from the relevant data from

tripController and weatherController

Meaning:

This class represents a single notification message constructed from the relevant data from tripController and weatherController.

<b>MessageModel</b>
-tripData : string -weatherData : string
+getTripData() : string +setTripData(tripData : string) +getWeatherData() : string +setWeatherData(weatherData : string) +getTextMessage() : string +getHTMLMessage() : string

### **Class: GmailController**

#### Attributes:

recipientList : List<string> - the list of email/SMS recipients

transporter : Object<Transporter> - a nodemailer email transporter connection object

mailOptions : Object<string> - a string object representing a valid email message built using a messageModel

#### Operations:

sendGmail() - Sends an email using a GMail email account

#### Meaning:

This class represents a single email created from a messageModel object sent to the listed recipients using a GMail email account.

<b>GmailController</b>
-recipientList : List<string> -transporter : Object<Transporter> -mailOptions : Object<string>
+sendGmail()

### **Class: SlackBotController**

#### Attributes:

slackBotModel : SlackBotModel - allows easier use of SlackBotModel operations

Operations:

serveFormHTML(req: Object, res: Object): Object - serves Slack form html

serveFormJS(req: Object, res: Object): Object - serves Slack form JS

postMessage(userId: string, formattedMessage: string) : Object - sends Slack message

Meaning:

This class controls the sending of messages to the Slack channel.

SlackBotController
-slackBotModel : SlackBotModel
+serveFormHTML(req: Object, res: Object): Object +serveFormJS(req: Object, res: Object): Object +postMessage(userId: string, formattedMessage: string) : Object

### Class: DiscordBotController

Attributes:

userId : string - the Discord userId used to send a Discord message

formattedMessage : string - message to send to Discord

\*These are passed as parameters to DiscordBotController instead

discordBotModel : DiscordBotModel - allows easier use of DiscordBotModel operations

Operations:

serveFormHTML(req: Object, res: Object): Object - serves Discord form html

serveFormJS(req: Object, res: Object): Object - serves Discord form JS

\*\*Allowed for early testing of Discord messaging functionality

postMessage(userId: string, formattedMessage: string) : Object - sends Discord message

Meaning:

This class controls the sending of messages to the Discord channel.



<b>DiscordBotController</b>
-userID : string -formattedMessage : string -discordBotModel : DiscordBotModel
+serveFormHTML(req: Object, res: Object): Object +serveFormJS(req: Object, res: Object): Object +postMessage(userId: string, formattedMessage: string) : Object

**Class: SlackBotModel** - \*\*Added to operate according to MVC design pattern

Attributes:

None

Operations:

postMessage(userId: string, formattedMessage: string) : Object - sets appropriate headers and properly formats requests to Slack API.

Meaning:

This class represents the model to send messages to the Slack channel

<b>SlackBotModel</b>
+postMessage(userId: string, formattedMessage: string) : Object

**Class: DiscordBotModel** - \*\*Added to operate according to MVC design pattern

Attributes:

client: Client - the client used to interact with the Discord API

Operations:

login(): Object - used to login to client

destroy(): Object - used to disconnect from and destroy client

postMessage(userId: string, formattedMessage: string) : Object - sets appropriate headers and properly formats requests to Discord API.

Meaning:

This class represents the model to send messages to the Discord channel

<b>DiscordBotModel</b>
-client: Client
+login(): Object +destroy(): Object +postMessage(userId: string, formattedMessage: string) : Object

### **Class: ReverseGeoCode**

#### Attributes:

HEADERS: Object - stores the HTTP header information

#### Operations:

setPoint(point: string, radius: int): void - allows the caller to set a point for the reverse geocode request

messageOperationSuccess(data: Object, message: string): Object - used to send a standardized success message when the call to the API is successful

messageOperationFailure(error: Object) - used to send a standardized failure message when the call to the API fails

getAddress(point: string, radius: int): Object - converts the given point (in lat/long form) to an address. The closest address (by radius) is found.

#### Meaning:

This class represents the functionality necessary to convert a geocode to a street address

<b>ReverseGeoCode</b>
+HEADERS: Object
-setPoint(point: string, radius: int): void -messageOperationSuccess(data: Object, message: string): Object

```
-messageOperationFailure(error: Object)
+getAddress(point: string, radius: int): Object
```

## **Class: RouteMappingService**

### Attributes:

start: string - route starting point  
end: string - route ending point  
startLatitude: string - latitude of starting point  
startLongitude: string - longitude of starting point  
endLatitude: string - latitude of ending point  
endLongitude: string - longitude of ending point

### Operations:

setRoute(start: string, end: string): void - initializes setting the latitude and longitude based on the start and end points  
setLatitudeAndLongitude(): void - sets the latitude and longitude from the start and end attributes stored in the class  
getRouteDistance(): void - calculates the distance between the start and end point in meters  
generateDynamicNumberOfPoints(routeDistanceInMeters: int, interval: int): int - calculates number of wayPoints that will be generated  
generateWayPoints(interval: int): List - generates a list of points along the route including start, way and ending points

### Meaning:

This class represents the functionality necessary to generate way points for a route

## RouteMappingService

-start: string  
-end: string  
-startLatitude: string  
-startLongitude: string  
-endLatitude: string  
-endLongitude: string

+setRoute(start: string, end: string): void  
-setLatitudeAndLongitude(): void  
+getRouteDistance(): int  
+generateDynamicNumberOfPoints(routeDistanceInMeters: int, interval: int): int  
+generateWayPoints(interval: int): List

### 9c. Traceability Matrix (Updated)

	Domain Concepts												
	inputValidation	homeController	scheduledTripsModel	externalScheduler	scheduleController	notificationController	travelController	weatherController	baseFetchRetry	messageModel	gmailController	slackBotController	discordBotController
Software Classes													
*InputValidation	x												
HomeController		x											
AddressModel		x											
ScheduledTripsModel			x										
*ExternalScheduler				x									
ScheduleController					x								
NotificationController						x							
TravelController							x						
TravelTimeModel							x						
TravelIncidentModel							x						
WeatherController								x					
WeatherModel								x					
BaseFetchRetry									x				
MessageModel										x			
GmailController											x		
SlackBotController												x	
SlackBotModel												x	
DiscordBotController													x
DiscordBotModel													x
ReverseGeoCode			x										
RouteMappingService						x							

\* The corresponding domain concepts were ultimately implemented as non-class components.

Our team decided to implement the Model-View-Controller (MVC) architectural style. This helps to implement the separation of concerns. We also tried to limit the responsibilities of individual components and conform to the design principles of expert doer, high cohesion, and low coupling. Due to this, many of our domain concepts were broken into multiple software classes. An example of this is when it made sense to have a separate controller and one or more models to better assign responsibilities. Below we provide a list of the domain concepts (in bold) followed by the classes (some similarly named) that were derived from each concept:

#### **inputValidation:**

- \*InputValidation: checks user input for validity before accepting input.  
\*This class was removed in favor of doing validation on the front end.

#### **homeController:**

- HomeController: serves the user interface and passes user input to the database.
- AddressModel: provides address autocomplete functionality.

#### **scheduledTripsModel:**

- ScheduledTripsModel: represents the data layer between the database and the rest of the system.
- \*\*ReverseGeoCode: converts stored geocodes to addresses.  
\*\* Class added after Report 2

#### **externalScheduler:**

- \*ExternalScheduler: reaches out to the scheduleController at specified intervals.  
\*This class was removed, as it was implemented as a non-class component.

#### **scheduleController:**

- ScheduleController: controls the upcoming scheduled notifications and manages the queue.

### **notificationController:**

- NotificationController: controls the process of creating/sending each individual scheduled notification.
- **\*\*RouteMappingService**: allows creation of waypoint locations for messages.  
\*\* Class added after Report 2

### **travelController:**

- TravelController: generates the text for the travel time and incident information that will be used in the user notification.
- TravelTimeModel: provides travel time data for a trip.
- TravelIncidentModel: provides travel incident data for a trip.

### **weatherController:**

- WeatherController: generates the text for the weather information that will be used in the user notification.
- WeatherModel: provides forecasted and current weather data for a trip.

### **baseFetchRetry:**

- BaseFetchRetry: serves as a template for all other classes that will make requests to REST APIs as they all share a need to retry requests when certain status codes are returned.

### **messageModel:**

- MessageModel: represents a single notification message constructed from the relevant data from tripController and weatherController.

**gmailController:**

- GmailController: represents a single email created from a messageModel object sent to the listed recipients using a GMail email account.

**slackbotController:**

- SlackbotController: controls the sending of messages to the Slack channel.

**discordController:**

- DiscordController: controls the sending of messages to the Discord channel.



## **9d. Design Patterns**

Early in our project, we decided to use the Model-View-Controller (MVC) design pattern. Projects often choose this particular pattern when developing web-based apps. It organizes the code-base conceptually and practically to improve maintainability and extensibility, like any of the design patterns. However, it is not the only pattern that we could have used.

After exposure to the design patterns present in the textbook, we discussed the merits of each one presented. Each of the patterns offers benefits and drawbacks if used.

The publisher-subscriber pattern could have been used to make our notification channels easily reusable. However, it did not seem to provide as much benefit to other aspects of the code-base. It seems to introduce more complexity in our code without observable gain over MVC. In fact, MVC provides a similar benefit with controllers that can be self-contained and provide similar functionality by exposing them as API endpoints to make them reusable. Because of this, we decided not to implement it.

A Command design pattern did not seem to fit our project. We need no undo/redo functionality. An argument could be made that our routes and the remote calls to 3rd party APIs implement functionality similar to that gained by using a Command pattern. However, it did not seem useful to apply it to the entire project.

We narrowed the scope of the project early to include only functionality we believed essential. The Decorator pattern promises extensibility for features we might add in the future. As with other patterns though, we found that it added complexity to our project without real benefit.

When we came to the Proxy design pattern, we noted specifically that the remote proxy pattern might prove useful. In fact, we observed that some parts of our system use this paradigm. Each controller serves as a client side proxy for the remote API calls to the 3rd party providers of data we retrieve. The ScheduledTripsModel serves a similar purpose for calls to create, insert, and retrieve data from the database.

For future work, a Protection Proxy pattern might be used to implement the multi-user functionality expected by a business. However, we already determined that the time available to us would require that we leave this for a revision and refactoring of the code-base. This seems a more appropriate time for changing the design pattern if necessary.

To conclude, we found that several patterns discussed in the textbook seemed to be components of the MVC pattern. There are portions of our code where one could argue that the patterns discussed above are used. In the end, we decided that refactoring the code to use those design patterns to the exclusion of any other provided no tangible benefit over our initial choice. MVC seems to allow those other patterns to be used where advantageous and it makes clear why it is often chosen for web-based projects.

## 9e. Object Constraint Language (OCL) Contracts

### **Class: HomeController**

context HomeController::serveIndex(req: Object, res: Object) inv:

staticPath  $\diamond$  null

context HomeController::serveIndex(req: Object, res: Object) pre:

req  $\diamond$  null and res  $\diamond$  null

context HomeController::serveIndex(req: Object, res: Object) post:

indexPathServed or errorThrown

context HomeController::serveStatic(req: Object, res: Object, next: Function) inv:

staticPath  $\diamond$  null

context HomeController::serveStatic(req: Object, res: Object, next: Function) pre:

req  $\diamond$  null and res  $\diamond$  null and next  $\diamond$  null

context HomeController::serveStatic(req: Object, res: Object, next: Function) post:

staticFilesServed or errorThrown

context HomeController::formSubmission(formSubmissionObject : FormData) pre:

formSubmissionObject  $\diamond$  null

context HomeController::formSubmission(formSubmissionObject : FormData) post:

tripSubmitted or formSubmissionError

### **Class: AddressModel**

context AddressModel::lookup(address: String) pre:

address  $\diamond$  null and address.size() > 0

context AddressModel::lookup(address: String) post:

addressesFound or errorThrown

### **Class: ScheduledTripsModel**

context ScheduledTripsModel::createTrip(trip\_details: Object) pre:

trip\_details <> null

context ScheduledTripsModel::createTrip(trip\_details: Object) post:

tripCreated or errorThrown

context ScheduledTripsModel::getAllTrips() inv:

databaseConnection <> null

context ScheduledTripsModel::getAllTrips() post:

tripsRetrieved or errorThrown

context ScheduledTripsModel::getTripById(id: int) pre:

id <> null and id > 0

context ScheduledTripsModel::getTripById(id: int) post:

tripRetrieved or errorThrown

context ScheduledTripsModel::getUpcomingTrips(input\_date: Date, offsetMinutes: int) pre:

input\_date <> null and offsetMinutes <> null

context ScheduledTripsModel::getUpcomingTrips(input\_date: Date, offsetMinutes: int) post:

upcomingTripsRetrieved or errorThrown

context ScheduledTripsModel::updateNotificationStatus(id: int, status: String) pre:

id <> null and status <> null

context ScheduledTripsModel::updateNotificationStatus(id: int, status: String) post:

notificationStatusUpdated or errorThrown

context ScheduledTripsModel::deleteTripById(id: int) pre:

id <> null and id > 0

context ScheduledTripsModel::deleteTripById(id: int) post:

tripDeleted or errorThrown

context ScheduledTripsModel::setAllProcessingTripsToFailed() post:  
processingTripsSetToFailed or errorThrown

context ScheduledTripsModel::close() post:  
connectionClosed or errorThrown

### **Class: ScheduleController**

context ScheduleController::run(produceStatusReport: Boolean) pre:  
produceStatusReport <> null

context ScheduleController::run(produceStatusReport: Boolean) post:  
schedulingCompleted or errorThrown

### **Class: NotificationController**

context NotificationController::createMessageObject(travelData: Object, weatherData: Object)  
pre:

travelData <> null and weatherData <> null

context NotificationController::createMessageObject(travelData: Object, weatherData: Object)  
post:

this.#messageObject <> null and messageObjectCreated or errorThrown

context NotificationController::createGmailObject(emailAddress: String, messageObject:  
Object) pre:

emailAddress <> null and messageObject <> null

context NotificationController::createGmailObject(emailAddress: String, messageObject:  
Object) post:

this.#gmailControllerObject <> null and gmailObjectCreated or errorThrown

context NotificationController::createSlackObject() post:

this.#slackBotControllerObject <> null

context NotificationController::createDiscordObject() post:  
this.#discordBotControllerObject <> null

context NotificationController::getMessageObject() post:  
this.#messageObject returned

context NotificationController::getTripData() post:  
this.#tripData returned

context NotificationController::fetchTravelData() post:  
typeof return === 'Promise<Object>'

context NotificationController::fetchWeatherData() post:  
typeof return === 'Promise<Object>'

context NotificationController::sendGMail() post:  
GmailMessageSent and typeof return === 'Promise'

context NotificationController::postSlackMessage(userId: String, message: String) pre:  
userId <> null and message <> null

context NotificationController::postSlackMessage(userId: String, message: String) post:  
slackMessagePosted or errorThrown

context NotificationController::postDiscordMessage(userId: String, message: String) pre:  
userId <> null and message <> null

context NotificationController::postDiscordMessage(userId: String, message: String) post:  
discordMessagePosted or errorThrown

context NotificationController::sendMessage() pre:

travelObject <> null and weatherObject <> null and messageObject <> null and  
gmailControllerObject <> null  
context NotificationController::sendMessage() post:  
messagesSent or errorThrown

### **Class: TravelController**

context TravelController::getTravelMessage() pre:  
start <> null and end <> null and apiKey <> null  
context TravelController::getTravelMessage() post:  
travelMessageGenerated or errorThrown

### **Class: WeatherController**

context WeatherController::getWeatherMessage() pre:  
start <> null and end <> null and tommorrowIOAPIKey <> null and TomTomAPIKey <>  
null  
context WeatherController::getWeatherMessage() post:  
weatherMessageCreated or errorThrown

### **Class: WeatherModel**

context WeatherModel::setPoint(point: String) pre:  
point <> null  
context WeatherModel::setPoint(point: String) post:  
pointSet or errorThrown  
  
context WeatherModel::getWeatherByInterval(type: string, number: int) pre:  
type === 'minutely' || type === 'hourly' and number >= 0  
context WeatherModel::getWeatherByInterval(type: string, number: int) post:  
weatherDataRetrieved or errorThrown

**Class: TravelTimeModel**

context TravelTimeModel::getTravelTimes() inv:

    this.fetchWithRetry() exists

context TravelTimeModel::getTravelTimes() post:

    travelTimesCalculated or errorThrown

**Class: TravelIncidentModel**

context TravelIncidentsModel::getTravelIncidents() post:

    this.fetchWithRetry() exists

context TravelIncidentsModel::getTravelIncidents() post:

    travelIncidentsRetrieved or errorThrown

**Class: BaseFetchRetry**

context BaseFetchRetry::isNetworkOrServerError(statusCode: string) pre:

    statusCode <> null

context BaseFetchRetry::isNetworkOrServerError(statusCode: string) post:

    typeof return === 'boolean'

context BaseFetchRetry::delay(milliseconds: int) pre:

    milliseconds <> null

context BaseFetchRetry::delay(milliseconds: int) post:

    typeof return === 'Promise'

context BaseFetchRetry::fetchWithRetry() post:

    fetchSuccessful or errorThrown



**Class: MessageModel**

context messageModel::setTripData(tripData: string) pre:

tripData <> null

context messageModel::setTripData(tripData: string) post:

this.#tripData <> null

context messageModel::getTripData() post:

this.#tripData returned

context messageModel::setWeatherData(weatherData: string) pre:

weatherData <> null

context messageModel::setWeatherData(weatherData: string) post:

this.#weatherData <> null

context messageModel::getWeatherData() pre:

this.#tripData <> null

context messageModel::getWeatherData() post:

this.#tripData returned

context messageModel::getTextMessage() post:

typeof return === 'string'

context messageModel::getHTMLMessage() post:

typeof return === 'string'

**Class: GmailController**

context GmailController::sendGMail() inv:

typeof this.#message === 'MessageModel'

context GmailController::sendGMail() pre:

this.#message <> null

context GmailController::sendGMail() post:  
    emailSentSuccessfully or errorThrown

**Class: SlackBotController**

context SlackBotController::serveFormHTML(req: Object, res: Object) inv:  
    typeof req === 'object'

context SlackBotController::serveFormHTML(req: Object, res: Object) pre:  
    req <> null

context SlackBotController::serveFormHTML(req: Object, res: Object) post:  
    HTMLFileSent or errorThrown

context SlackBotController::serveFormJS(req: Object, res: Object) inv:  
    typeof req === 'object'

context SlackBotController::serveFormJS(req: Object, res: Object) pre:  
    req <> null

context SlackBotController::serveFormJS(req: Object, res: Object) post:  
    JSFileSent or errorThrown

context SlackBotController::postMessage(...args) inv:  
    typeof args[0] === 'object' && args[0].body || args.length === 2

context SlackBotController::postMessage(...args) pre:  
    userId <> null and formattedMessage <> null and formattedMessage.length > 0

context SlackBotController::postMessage(...args) post:  
    messageSentSuccessfully or errorThrown

**Class: SlackBotModel**

context SlackBotModel::postMessage(userId: string, formattedMessage: string) inv:  
    typeof userID === 'string' && typeof formattedMessage === 'string'

context SlackBotModel::postMessage(userId: string, formattedMessage: string) pre:

userId <> null and formattedMessage <> null

context SlackBotModel::postMessage(userId: string, formattedMessage: string) post:  
messageSentSuccessfully or errorThrown

### **Class: DiscordBotController**

context DiscordBotController::serveFormHTML(req: Object, res: Object) inv:  
typeof req === 'object'

context DiscordBotController::serveFormHTML(req: Object, res: Object) pre:  
req <> null

context DiscordBotController::serveFormHTML(req: Object, res: Object) post:  
HTMLFileSent or errorThrown

context DiscordBotController::serveFormJS(req: Object, res: Object) inv:  
typeof req === 'object'

context DiscordBotController::serveFormJS(req: Object, res: Object) pre:  
req <> null

context DiscordBotController::serveFormJS(req: Object, res: Object) post:  
JSFileSent or errorThrown

context DiscordBotController::postMessage(...args) inv:  
typeof args[0] === 'object' && args[0].body || args.length === 2

context DiscordBotController::postMessage(...args) pre:  
userId <> null and formattedMessage <> null

context DiscordBotController::postMessage(...args) post:  
messageSentSuccessfully or errorThrown

### **Class: DiscordBotModel**

context DiscordBotModel::login() pre:  
this.#client <> null

context DiscordBotModel::login() post:  
    successfulLogin or errorThrown

context DiscordBotModel::destroy() pre:  
    this.#client <> null

context DiscordBotModel::destroy() post:  
    clientDestroyedSuccessfully or errorThrown

context DiscordBotModel::postMessage(userId: string, formattedMessage: string) pre:  
    typeof userID === 'string' && typeof formattedMessage === 'string'

context DiscordBotModel::postMessage(userId: string, formattedMessage: string) post:  
    messageSentSuccessfully or errorThrown

### **Class: ReverseGeoCode**

context ReverseGeocode::getAddress(point: string, radius: int) inv:  
    typeof point === 'string' && typeof radius === 'int'

context ReverseGeocode::getAddress(point: string, radius: int) pre:  
    point <> null

context ReverseGeocode::getAddress(point: string, radius: int) post:  
    addressRetrievedSuccessfully or errorMessageReturned

### **Class: RouteMappingService**

context RouteMappingService::setRoute(start: string, end: string) pre:  
    start <> null and end <> null

context RouteMappingService::setRoute(start: string, end: string) pre:  
    this.#start <null> and this.#end <> null

context RouteMappingService::getRouteDistance() post:  
    distanceInMeters returned

context RouteMappingService::generateDynamicNumberOfPoints(routeDistanceInMeters: int,  
interval: int) pre:

routeDistanceInMeters  $\neq$  null

context RouteMappingService::generateWayPoints(interval: int) post:

waypointsGeneratedSuccessfully

## **10. Algorithms and Data Structures**

### **10a. Algorithms**

Our program myTrafficWizard does not utilize any complex algorithms in its code.

### **10b. Data Structures**

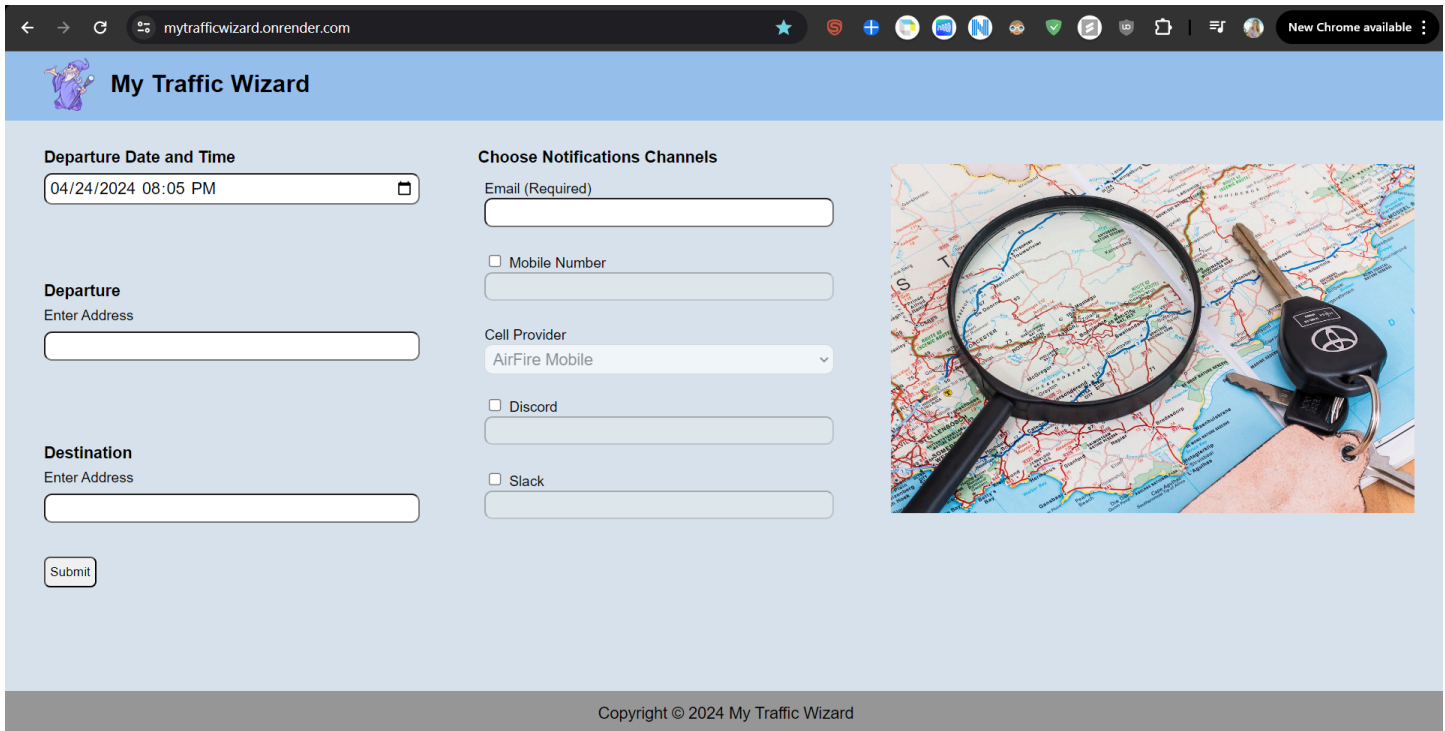
In our system architecture, we employ a queue data structure to efficiently manage outgoing notifications for weather and traffic information. Users have the flexibility to choose the time they wish to receive these updates. When the scheduler retrieves one or more notifications, they are received from the ScheduledTripsModel and ordered by departure time. The notifications are then enqueued in our system.

The queue ensures that requests are processed in the order of trip departure times, following a first-in-first-out (FIFO) approach. As the scheduled time approaches, the system dequeues the requests and sends out the corresponding weather and traffic updates to the users. By leveraging a queue-based approach, we can seamlessly handle user requests while ensuring timely delivery of information according to user preferences.

### **10c. Concurrency**

Our program myTrafficWizard does not utilize any concurrency in its code.

## 11. User Interface Design and Implementation (Updated)



The screenshot shows a web browser window with the URL `mytrafficwizard.onrender.com`. The page title is "My Traffic Wizard" and features a purple wizard icon. The interface is divided into three main sections:

- Departure Date and Time:** A text input field containing "04/24/2024 08:05 PM" with a calendar icon on the right.
- Departure:** A section with the label "Departure" and the instruction "Enter Address". It contains a single text input field.
- Destination:** A section with the label "Destination" and the instruction "Enter Address". It contains a single text input field.
- Choose Notifications Channels:** A section with the following fields:
  - "Email (Required)": A text input field.
  - "Mobile Number": A checkbox followed by a text input field.
  - "Cell Provider": A dropdown menu currently showing "AirFire Mobile".
  - "Discord": A checkbox followed by a text input field.
  - "Slack": A checkbox followed by a text input field.

A "Submit" button is located at the bottom left of the form area. To the right of the form is a decorative image of a magnifying glass over a map with a set of keys.

Copyright © 2024 My Traffic Wizard

We modified our initial user interface design after measuring the original user effort and some discussion. With that in mind, we decided to reduce the number of fields required for the user to input address information and removed an unnecessary date field. The destination date field was removed because it provided no real benefit to the user.

After implementing our design changes, we have measured the reduced effort of the interface for the user. The same sample address (123 Anywhere Hays, KS) now requires 6 fewer clicks to enter it in both the departure and destination address fields. Our changes also simplify the layout of the page for the user.


The results of our changes can be seen in the desktop layout above and the mobile layout on the next page.



## My Traffic Wizard



### Departure Date and Time

04/24/2024 08:11 PM 

### Departure

Enter Address

### Destination


Enter Address

### Choose Notifications Channels

Email (Required)

Mobile Number

Cell Provider

AirFire Mobile 

Discord

Slack



## 12. Design of Tests (Updated)

### 12a. Test Cases

#### HomeController:

<b>Test Case Identifier:</b> TC-1	
<b>Use Case Tested:</b> UC-1	
<b>Pass/Fail Criteria:</b> This test succeeds if the URL is correctly entered and the page is found and rendered correctly. The test fails if the URL is entered incorrectly or the directory/page is unavailable.	
<b>Input Data:</b> URL request	
Test Procedure	Expected Results
A valid URL with an available file to load is requested.	The page loads the correct content.
An invalid URL or URL to a non-existent file is requested.	The page fails to load correctly.

<b>Test Case Identifier:</b> TC-2	
<b>Use Case Tested:</b> UC-1	
<b>Pass/Fail Criteria:</b> This test succeeds if the user-entered data is used to create a form object. It fails when the user-entered data does not properly create a form object.	
<b>Input Data:</b> FormData object	
Test Procedure	Expected Results
A valid FormData object is received by the HomeController.	The FormData object is assigned as the formSubmissionObject in the HomeController.
An invalid FormData object is received by the HomeController.	No formSubmissionObject is assigned.

**AddressModel:**

<p><b>Test Case Identifier:</b> TC-3</p> <p><b>Use Case Tested:</b> UC-1</p> <p><b>Pass/Fail Criteria:</b> This test succeeds if the input provided causes addresses to be returned by the api. It fails when an improperly formatted api call causes an error to be returned instead of addresses.</p> <p><b>Input Data:</b> A partial address string and the api key</p>	
Test Procedure	Expected Results
An improperly crafted api call is sent to TomTom.	An error is returned.
A properly created api call is sent to TomTom.	Addresses that closely match the partial one sent in the api call are returned.

**ScheduledTripsModel:**

<b>Test Case Identifier:</b> TC-5	
<b>Use Case Tested:</b> UC-1, UC-2, UC-3, UC-4	
<b>Pass/Fail Criteria:</b> The test will pass if a trip is successfully created in the database. The test will fail if a trip fails to be created in the database.	
<b>Input:</b> Trip object containing key value pairs that correspond to field/value in the database.	
<b>Test Procedure</b>	<b>Expected Results</b>
Set valid trip details	Data is inserted into the database and success message returned
Set invalid trip details	Data is not inserted into the database and failure message returned

<b>Test Case Identifier:</b> TC-6	
<b>Use Case Tested:</b> UC-7	
<b>Pass/Fail Criteria:</b> The test will pass if the notification status on a trip is successfully updated in the database. The test will fail if the notification status on a trip fails to update in the database.	
<b>Input Data:</b> trip id	
<b>Test Procedure</b>	<b>Expected Results</b>
Set valid trip id	The notification status is updated for the requested trip and a success message is returned
Set invalid trip id	The notification status is not updated for the requested trip and a failure message is returned

**ScheduledTripsModel (cont.):**

**Test Case Identifier:** TC-7

**Use Case Tested:** UC-7

**Pass/Fail Criteria:** The test will pass if a valid result set of trips is returned for a given offset. The test will fail if an invalid number of records is returned.

**Input Data:** start timestamp and offset in minutes

<b>Test Procedure</b>	<b>Expected Results</b>
Set valid start and offset in minutes	The database returns the expected number of records for the query

**TravelController:**

<b>Test Case Identifier:</b> TC-10	
<b>Use Case Tested:</b> UC-5	
<b>Pass/Fail Criteria:</b> The test will pass if a valid travel message is requested and received. The test will fail if an invalid request is made.	
<b>Input Data:</b> startPoint, endPoint, and API key	
<b>Test Procedure</b>	<b>Expected Results</b>
Set valid startPoint, endPoint, and API key	A correctly formatted message containing travel time and incident information is returned
Set invalid startPoint and endPoint	The returned message contains content about the error
Set invalid API key	The returned message contains content about the error

**TravelTimeModel:**

<b>Test Case Identifier:</b> TC-11	
<b>Use Case Tested:</b> UC-5	
<b>Pass/Fail Criteria:</b> The test will pass if a method call to getTravelTimes returns an object with travel times. The test will fail if an invalid startPoint, endPoint, or API key is provided and getTravelTimes is called.	
<b>Input Data:</b> startPoint, endPoint, and API key	
<b>Test Procedure</b>	<b>Expected Results</b>
Instantiate object with valid startPoint, endPoint, and API key	An object is returned with travel times
Instantiate object with invalid startPoint and endPoint	An error message is returned
Instantiate object with invalid apiKey	An error message is returned

**TravelIncidentModel:**

<b>Test Case Identifier:</b> TC-12	
<b>Use Case Tested:</b> UC-5	
<b>Pass/Fail Criteria:</b> The test will pass if a method call to getTravelIncidentDetails returns an object with travel incidents. The test will fail if an invalid startPoint, endPoint, or API key is provided and getTravelIncidentDetails is called..	
<b>Input Data:</b> startPoint, endPoint, and API key	
<b>Test Procedure</b>	<b>Expected Results</b>
Instantiate object with valid startPoint, endPoint, and API key	An object is returned with travel incidents
Instantiate object with invalid startPoint and endPoint	An error message is returned
Instantiate object with invalid apiKey	An error message is returned

**WeatherController:**

<b>Test Case Identifier:</b> TC-13	
<b>Use Case Tested:</b> UC-6	
<b>Pass/Fail Criteria:</b> The test will pass if a valid weather message is requested and received. The test will fail if an invalid request is made.	
<b>Input Data:</b> startPoint, endPoint, and API key	
<b>Test Procedure</b>	<b>Expected Results</b>
Set valid startPoint, endPoint, and API key	A message containing weather data is returned
Set invalid startPoint and endPoint	An exception is thrown
Set invalid API key	An exception is thrown

**WeatherModel:**

<p><b>Test Case Identifier:</b> TC-14</p> <p><b>Use Case Tested:</b> UC-6</p> <p><b>Pass/Fail Criteria:</b> The test will pass if the weatherModel is able to pull the weather API data from tomorrow.io correctly. The test will fail if the weatherModel fails to make a call or returns incorrect data.</p> <p><b>Input Data:</b> startPoint, endPoint, and API key</p>	
Test Procedure	Expected Results
The weatherModel correctly calls the weather API	The weather information is retrieved correctly
The weatherModel fails to call the weather API	The weather information is not retrieved and a message will show stating “failed to retrieve data”

**MessageModel:**

<p><b>Test Case Identifier:</b> TC-15</p> <p><b>Use Case Tested:</b> UC-7</p> <p><b>Pass/Fail Criteria:</b> The test will pass if a correctly formatted text message and html message can be retrieved given proper input. The test will fail when incorrect input is received.</p> <p><b>Input Data:</b> Travel Data and Weather Data</p>	
Test Procedure	Expected Results
Instantiate object with incorrect travel data and incorrect weather data.	The text message and html message returned are not formatted correctly.
Instantiate object with incorrect travel data and correct weather data.	The text message and html message returned are not formatted correctly.
Instantiate object with correct travel data and incorrect weather data	The text message and html message returned are not formatted correctly.
Instantiate object with correct travel data and correct weather data	The text message and html message returned are properly formatted.

**GmailController:**

<p><b>Test Case Identifier:</b> TC-16</p> <p><b>Use Case Tested:</b> UC-7</p> <p><b>Pass/Fail Criteria:</b> The test will pass if an email is sent to the user when correct input is provided. The test will fail when a user provides incorrect information.</p> <p><b>Input Data:</b> One or more email addresses and a message to send.</p>	
<b>Test Procedure</b>	<b>Expected Results</b>
User provides an invalid email address.	The message does not arrive in their email inbox.
User provides a valid email address.	The message arrives in their email inbox.

**SlackBotController:**

<p><b>Test Case Identifier:</b> TC-17</p> <p><b>Use Case Tested:</b> UC-7</p> <p><b>Pass/Fail Criteria:</b> This test will pass if a Slack message is sent to the user when valid input is provided. The test will fail if the input is invalid.</p> <p><b>Input Data:</b> formattedMessage, userId</p>	
<b>Test Procedure</b>	<b>Expected Results</b>
A valid formattedMessage and userId are provided.	Response indicates message delivery to the user.
An invalid formattedMessage and userId are provided.	Response indicates an error occurred.



**DiscordBotController:**

<p><b>Test Case Identifier:</b> TC-18</p> <p><b>Use Case Tested:</b> UC-7</p> <p><b>Pass/Fail Criteria:</b> This test will pass if a Discord message is sent to the user when valid input is provided. The test will fail if the message fails to be sent.</p> <p><b>Input Data:</b> formattedMessage, userId</p>	
Test Procedure	Expected Results
A valid formattedMessage and userId are provided.	Response indicates message delivery to the user.
An invalid formattedMessage and userId are provided.	Response indicates an error occurred.

**BaseFetchRetry:**

<p><b>Test Case Identifier:</b> TC-19</p> <p><b>Use Case Tested:</b> UC-5, UC-6</p> <p><b>Pass/Fail Criteria:</b> This test will pass if a response code <math>\geq 500</math> or 429 occurs and the request is retried until maxRetries is reached. The test will fail if the request is not retried or a request is initiated beyond the maxRetries threshold</p> <p><b>Input Data:</b> url, options, maxRetries = 5</p>	
Test Procedure	Expected Results
Initiate request that produces a response code $\geq 500$ or equal to 429; maxRetries = 0	The request is retried until max retries is reached
Initiate request that does not produce a response code $\geq 500$ or equal to 429; maxRetries = 0	The request is not retried
Initiate request that produces a response code $\geq 500$ or equal to 429; maxRetries = maxRetries	The request is not retried

## **12b. Test Coverage**

The test cases created by our group validate the functionality of all the critical classes used in our system. They test the system's essential functionality based on our current goals, and we will be open to revising them in the future if we decide to implement stretch features. Our approach was to focus on source code coverage and creating tests to verify that each class correctly interacts with others. Ensuring that each individual unit responds as expected without any errors should make it much easier to detect and quickly locate any potential issues that need to be debugged. To make it easier to locate errors and not risk having cumulative errors that are hard to track down, we will be implementing the bottom-up integration method.

Our test cases will cover the critical functions of each class. The design of the individual tests should make it easy to confirm test pass and fail scenarios. The correct response of each individual unit is critical since other units may rely on that unit's response. Therefore, we have tried to ensure that every unit will respond appropriately to guarantee correct unit interactivity. This will help us to build a resilient, error-resistant system that provides trouble-free service to end users.

### **12c. Integration Testing**

Our integration testing will be conducted based on the bottom-up integration approach. We will start by testing the lower-level units without dependencies and work upwards to test those units that will depend upon those lower in the unit hierarchy. The highest-level units will be tested last once the units they rely on are found to successfully pass their own test cases. This approach is preferable for our system since it allows for the quick identification of any errors in lower-level units that are necessary for the functionality of higher-level units. This procedure should allow for the quick identification of any bugs, which we should be able to quickly and efficiently resolve.

For our system, this method seems to be the optimal way to avoid intensive debugging with cumulative errors at different levels. Other approaches could introduce overlapping or cumulative errors that would complicate the troubleshooting process. Ensuring the lowest-level units are working properly first make it much easier to identify the cause of an error as we move up the unit hierarchy. Bottom-up integration should allow us to avoid the pitfalls inherent in other methods when dealing with a modularly designed system like ours.

Higher-levels of the hierarchy, including the ScheduleController and NotificationController, will be tested after all lower-level unit tests have been completed. It is important to do these last since they are primarily orchestrating the actions of the lower-level units. The final round of testing will include all components from the user interface to the sending of the final notifications.

## **12d. System Testing** (Updated)

After our individual units were thoroughly tested, any modifications necessary to fully integrate the components were completed. Once passing all unit and integration tests, the focus moved to system testing. These types of tests include end-to-end testing, installation testing, deployment testing, and quality testing. Our overall goal was to ensure the reliability and smooth functioning of our application.

For installation testing, we wanted to ensure that our web interface would be available to consumers via both desktop and mobile devices. Therefore, we tested that our web page design would work for multiple devices using the Device Toolbar feature within Chrome's Inspector Console. We also confirmed the functionality on our own devices.

Deployment testing was conducted by deploying our application through Render using the feature that allows launching directly from a GitHub repo. During deployment, any errors or problems with dependencies are displayed in the log monitor. Any such issues were addressed to make sure that our web app would be fully functional. Upon successful deployment without error, Render confirms that the web app is live and accessible.

Finally, we have confirmed that our app is functional and not causing errors from end-to-end from the creation of trip submissions through the user interface, to the valid submissions entering the database, to the external scheduler that triggers the search for upcoming messages, to the retrieval of upcoming trips from the database, to the retrieval of weather and traffic data, to the formatting of messages, to the final sending of messages through all selected channels. We have tested our entire app very thoroughly to ensure that it works reliably and smoothly.

## **Project Management and Plan of Work**

### **a. Merging the Contributions from Individual Team Members**

Ensuring the final draft is uniform and consistent can be a struggle when working collaboratively with multiple team members. In order to provide an excellent final product, our team scheduled multiple recurring meetings every week to talk through the complex planning process. This enabled us to consistently be in sync on how to build our system and contribute to the written report. During our meetings, we were able to make live changes to shared documents including those for brainstorming, organizing our repo structure, and tracking any changes made as we progressed through the report sections.

We were also able to make live changes and suggested changes to the main shared report documents that we created using Google Docs. Compared to Microsoft Word, Google Docs does have fewer formatting features and less flexibility when it comes to table formatting (like having vertical text). However, our team found the ability to make live changes while working collaboratively worth the tradeoff. Although our group meetings generally involved creating complex sections together, we also did distribute sections that were modular among the team members. We would then review each other's sections at the next meeting or asynchronously via our group's Discord server to avoid any errors or misunderstanding of the intended design.

Even with the group working well to understand the concepts and logic behind our system design and report sections, there were still sometimes formatting inconsistencies. The final step was to review the document to ensure consistent capitalization, use of bold/italic styles, naming conventions, margins, table layout, etc. Our group members attempted to copy each others' table styles and other formatting conventions in different sections to make the final review easier. The final touches to enhance the appearance were then added, along with updates to the table of contents, prior to the reports being submitted.

## **b. Project Coordination and Progress Report** (Updated)

**\*\*Updated Note:** All use cases have been implemented as of our final submission. The previous status as of the second report is still shown below:

### **Report 2 Progress Report**

Currently, we are implementing the individual components that contribute to each use case. The following components have been implemented so far:

Routes: addressRoutes, discordRoutes, gmailRoutes, homeRoutes, slackRoutes

Services: BaseFetchRetry

Static: CSS file, Images

Views: index.html (our interface)

Controllers: gmailController, homeController, notificationController, slackBotController, and travelController

Models: addressModel, messageModel, scheduledTripsModel, slackBotModel, travelIncidentModel, and travelTimeModel

Additionally, we have established our database, the monolithic table, our web service (Render) and are currently versioning our work via Github. We have tested connectivity to all the APIs we will be using in the implementation and have arrived at a robust mental model of how the system will function.

We are currently meeting 3-4 times a week for around 1-2 hours. At this point, we have soft “owners” of components, meaning each person will be responsible for building some component. The focus from a project management standpoint has shifted from design to implementation. This involves much more dedicated time to bursts of programming and regrouping to ensure our solution is well integrated and tested.

### **Report 3 Updates**

In addition to the components mentioned above, we have also implemented the following:

Routes: schedulerRoutes

Services: reverseGeoCode and routeMappingService

Views: discord\_bot.html and slack\_bot.html (used for testing message sending to Discord/Slack)

Controllers: discordBotController, scheduleController, and weatherController

Models: discordBotModel and weatherModel

### c. History of Work

The project has progressed rapidly during this course. All of the primary use cases have been fully implemented. The My Traffic Wizard app has fulfilled our goals of being a unique, helpful traffic and weather application. The overall timeline of our project was very similar to our planned timeline, although there was not as much separation of implementation, testing, and deployment as we originally planned (see original project timeline below). Individual product owners often created their own tests to ensure their sections were working prior to the more uniform unit and system integration tests were created. Further, needing visible features ready to show for Demo 2 resulted in our project being divided into the components needed for the Gmail/Slack channel notifications and those needed for SMS/Discord and automated scheduling.

Phase	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Week 11	Week 12	Week 13	Week 14	Week 15
Project Planning															
Requirements															
Design															
Implementation															
Testing															
Deployment															

Original Project Timeline

Notable programming progress as of the time of the second report included the implementation of message sending via Gmail and through Slack. The infrastructure was also set up during this period with a PostgreSQL database and app hosting via Render and version control via GitHub. As the code for the messaging channels were developed, our team thoroughly tested our functions to ensure a robust, error-resistant application. Thus, our originally scheduled timeline was also a bit different than our actual timeline, as we wanted to individually deploy features within our own product ownership domains as soon as we had them ready to fully test them.

Since the second report, we have further expanded our functioning message channels to include SMS and Discord. To improve the information about the trips included in the messages, the services reverseGeoCode and routeMappingService were created. The front-end interface



discord\_bot.html was created soon after the corresponding classes were created for early testing to confirm the message sending process. The weatherController and weatherModel were also implemented after the second demo and report. This allows users to receive the current weather, as well as a brief forecast to help them plan their trip.

### **Key Accomplishments:**

- Implemented all use cases along with their associated components
- Enabled separation of concerns using the MVC design pattern
- Created successful group communication channel and meeting schedule
- Set up app's database and web app hosting via Render.com and version control via GitHub
- Integrated multiple third-party APIs into a seamless message pipeline
- Worked closely as a team to integrate our individual components
- App provides a user-friendly interface for the submission of trip alert requests
- App successfully sends messages via Discord, Slack, Gmail, and SMS
- App successfully sends trip alerts including traffic and weather data
- My Traffic Wizard provides a unique offering with channels not available on other apps

## Actual Work Timeline (Updated):

Task Name	Planned Duration	Actual Start	Actual End	Actual Duration
<b>Requirements and Documentation</b>				
Project Proposal	7	1/15/24	1/19/24	4
Report 1	28	1/29/24	2/18/24	20
Report 2	21	2/18/24	3/10/24	22
Report 3	28	3/18/24	4/28/24	42
API Research	7	1/14/24	1/21/24	8
<b>Design</b>				
System Design	5	2/12/24	3/10/24	28
Database Design	5	2/22/24	3/10/24	18
<b>Implementation</b>				
Create Render account and link to GitHub	1	1/26/24	1/26/24	1
Write code for Discord integration	7	2/11/24	4/15/24	65
Write code for email and SMS integration	5	2/11/24	4/9/24	59
Write code for weather and traffic API integration	10	2/11/24	4/18/24	68
Write code for Slack integration	7	2/11/24	3/25/24	44
Write code for notifications	7	3/4/24	4/22/24	50
Create database based on data definitions	5	2/22/24	3/9/24	17
Create user interface	5	1/31/24	2/10/24	11
Write code for web app functionality	15	3/4/24	4/12/24	40
<b>Testing</b>				
Test code for Discord integration	3	4/10/24	4/20/24	11
Test code for weather and traffic API integration	3	3/22/24	4/20/24	30
Test code for email and SMS integration	3	3/22/24	4/14/24	24
Test code for Slack integration	3	3/17/24	4/2/24	17
Test code for notifications	3	3/22/24	4/22/24	32
QA of final product	3	4/22/24	4/28/24	7
<b>Deployment</b>				
Deploy app to Render	1	2/10/24	4/28/24	79

#### **d. Breakdown of Responsibilities**

Below is the breakdown of responsibilities per component. The symbol “`” denotes a class or function and implies that it will be developed and tested by the assigned to members:

Work Item	Assigned To
`homeController	Nicole
`scheduleController	Tyler
`notificationController	Nicole
`weatherController	Tyler
`travelController	Tyler
`gmailController	Phil
`slackBotController	Nicole
`discordBotController	Jacob
`addressModel	Phil
`scheduledTripsModel	Tyler
`weatherModel	Tyler
`messageModel	Phil
`travelTimeModel	Tyler
`travelIncidentModel	Tyler
`externalScheduler	Tyler
`inputValidation	Jacob
User Interface (HTML, CSS, JS)	Nicole, Phil
`addressInputHandler	Phil
Configure routes	Nicole
Test user interface	Team

Bottom up integration testing	Team
System Testing	Team

We will coordinate and work the integration as a team. This group work will occur during our daily meetings. The unit testing will be done for each unit by the assigned members, and the integration testing will be done as a group.

### **e. Current Status**

Our team is proud to have been able to implement all of our primary use cases. Although there were a few requirements that were eliminated due to the project timeline, the majority of the requirements were fulfilled by the time of our final submission. We are happy to say that our final app provides a good value and user experience to potential customers. The app we've created fulfills a niche need, providing messaging channels that are not available in most other traffic and weather alert applications.

The My Traffic Wizard app currently allows users to receive notifications regarding their submitted trips via Discord, Slack, email, and SMS. These integrations have been well-tested and are working smoothly and consistently. Due to financial limitations, we are using a host that sleeps the app when it has not been used for 15 minutes. Therefore, there can be some delay in processing caused by this. Similarly, our external scheduler triggers are limited on the free-tier plan, reducing how often we have chosen to queue the upcoming alerts.

Other than these minor limitations, however, our app is fully functional. It provides a unique service that is beneficial to users. The niche it fills is also relatively unique compared to other apps on the market. We feel that we have worked well together to create a useful, beneficial product in such a short timeframe. Future work options to improve our app even further are discussed in the next section.

## **f. Future Work**

There were a few minor features removed from the project, as they were not feasible to implement within the given timeframe. These features would be excellent for future work. One helpful feature for users would be to display a map of their route in the web interface upon trip submission (REQ14 and REQ24). This would help them visually confirm their planned route and help to avoid the accidental selection of incorrect addresses. Another helpful feature would be to allow users to add nicknames for the routes, so that their notifications will reference a particular trip nickname (originally part of REQ1).

Another excellent focus for future work would be to enable users to set recurring reminders (REQ11) and control the timing of alerts based on their pre-departure routine (REQ10). Making the app more flexible by allowing users to decide exactly how long before the trip they want to receive a notification (or multiple notifications at different intervals prior to departure) would make the app much more flexible and better able to adapt to user needs.

Adding more options for users to modify their existing trips would also be a value-adding feature. If users could look up their upcoming trips (either through the web interface or by requesting an emailed list), they could then potentially modify the scheduling of any upcoming notifications, as desired. While the timeline of this project did not allow for the implementation of these features, they would certainly be excellent topics for future work to make our app even better.

## **References**

1. Discord, chat app. <https://discord.com/developers/docs/intro>
2. GitHub, version control and project planning. <https://docs.github.com/en>
3. Gmail, email provider. <https://developers.google.com/gmail/api/guides/sending>
4. Knock app, alerts and push notification system. <https://knock.app/>
5. Render, web app hosting provider. <https://docs.render.com/>
6. Slack, chat app. <https://api.slack.com/start>
7. Tom Tom, traffic API.  
<https://developer.tomtom.com/traffic-api/documentation/product-information/introduction>
8. Tomorrow.io, weather API. <https://www.tomorrow.io/weather-api/>