POLITECNICO DI MILANO

Scuola di Ingegneria Industriale e dell'Informazione
Master's Degree in Automation and Control Engineering

SOFTWARE ENGINEERING

PROJECT REPORT
SMART COMMUTE WEB APP

Antonio Michele Del Negro - 862516
Pável Pascacio de los Santos -852255

Prof. Matteo Rossi

March 7th, 2019

Introduction

Smart Commute is a web-based application that aims to make the user's appointments scheduling easier by considering the time it takes to go from one place to another through integrated geolocation API's. In addition, within the same application it saves the appointments' addresses and durations and shows the best way to get from one place to another, warns you if there's no time, and helps coordinate efficiently.

The application was developed through JavaScript, HTML 5 and CSS for styling, and any information gathered per user was managed and stored in the Realtime Database tool from Google Firebase. Furthermore, external API's were used to implement the Calendar and Geolocation modules such as Fullcalendar.io and Here Maps API and it's fully compatible for any modern browser such as Microsoft Edge, Google Chrome, Firefox, Opera, Apple Safari, etc. The process development of the Web App is reported in this document, which is divided as follows:

App overview, which describes the overall description of the web-app and the idea, the motivations and justifications, including the user personas, followed by the web app requirements and the general objective.

System design section is dedicated to defining the web app architecture, interactions and relations among components, using the system architecture on front and back end, described by class diagrams, flowchart diagrams, and sequence diagrams.

Then, the software implementation section is mainly devoted to describing the user interface, the structure in the database and the code structure, analysis and integration of the modules of the system. Additionally, it highlights the main libraries, frameworks and API's used for front and back-end development.

Finally, the testing section describes and reports the end-to-end testing of the different app modules as well as a more detailed unit testing of the main functions.

Index

App Overview

Nowadays, work routines are getting progressively more hectic, as population and development increase (sometimes exponentially) over time. People in several industries must quickly adapt to new routines and appointments, go from one place to the other within the city in a required amount of time, while trying not to be late. Sometimes accomplishing that requires extra time and demands the individual to spend a handful of minutes on the smartphone and calculate its day routine, sometimes even switching between apps, resulting in a quite confusing procedure which must be performed daily.

Smartphones mainly, followed by other computing devices which are also able to let the users access high level dedicated mapping software are centered on applications. Most of the functionalities of a mapping application depend on the versatility and portability of a Smartphone, hence why the attractiveness on focusing the development on mobile apps, developed on either Android or iOS. However, attempts have been made to turn smartphones more "browser-native" and web-oriented. Firefox has launched in the past its own web-powered operating system: Firefox OS, in an attempt to bring a homogeneous solution to compatibility and versatility, while promoting these software infrastructures on a "freer" environment such as the web. It launched until 2015 a series of low-end smartphones who had this system, as well as partnerships with other companies such as Panasonic, implementing Firefox OS on Smart TV's. The project ultimately failed due to a lack of vision and resources, and although Firefox's operating system was discontinued, its project was further enhanced by other Startups such as JioPhone, on the development of KaiOS; which released very low-end feature phones, mainly in India and reaching to be the second most used phone in a 1.4 billion people country.

This argument opens for an encouragement to focus on the web, as it has become more powerful over the years and the mobile web browsing has grown considerably. This allows to consider the development of a mapping web app an opportunity to provide a tool that exploits all the benefits of the open source, is compatible with virtually any smart device and aims to be as efficient (and simpler) as any other map mobile apps currently in the market.

On the other hand, many solutions have been developed until now, but in most of the cases their focus is not oriented to a highly dynamic work environment but instead on a general purpose, fully equipped application which can perform sites exploring, landmarks, provides imagery, geolocation, traffic, routing and even multiple waypoint route coordination. These apps such as Bing Maps, Google Maps, City Mapper, Moovit or Apple Maps are quite complete, very powerful and standard. They offer to be a bulky, integral software which provides many functionalities to cover up any problems most of the people could ever have. In short, they have built up very useful, complete pieces of software that adapt to most personas that use smart devices.

This has led to understatements for specific user profiles which can't be fully satisfied due to the high number of functionalities these mobile apps have. For instance, most of them don't have an integration with a user agenda, and sometimes they don't even consider it. We can seldom find dedicated schedule applications whose objective is instead an intelligent scheduler that considers events times and locations and represents them not only in a calendar interface but also in a map interface, and when they are found, they are normally restricted to smartphones or tablets but not scalable to portable computers.

The idea

Smart Commute intends to construct an even easier to use but also simpler, narrower interface, dedicated to a relatively small proportion of this large group of users that for one reason or the other must use a mapping app or software. The idea is to discard mostly all the functionalities that big mapping software currently provides, and instead taking a handful of them, and automating most of those handful of functionalities. We want to let the user focus on what its agenda is, where to go and how to manage the time, without any distractions on unnecessary features. Therefore, the use case is centered on two things: managing the appointments (the one and only source of information for us to do all the job) and viewing them on the calendar and on the map, together with the means to go from one to another. In the following sections this functionality will be better appreciated with modeling diagrams.

For a depiction of the web app overview, and in spite of differentiating it from other software, here's a scheme that shows the difference in approach towards the user needs:

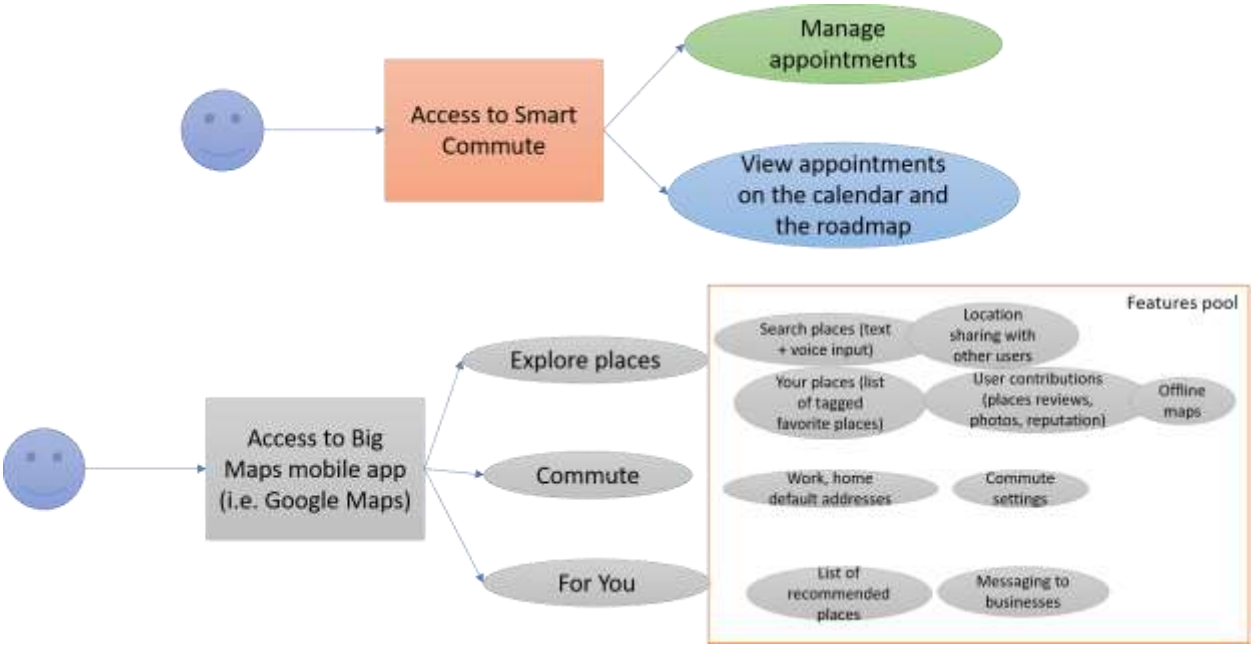

Figure 1.1: Smart Commute differentiation with other large-scale mobile apps (i.e. Google Maps)

We took the example of Google Maps for iOS and highlighted most of the main features of the mobile app. It is clearly noticeable that Google Maps (and basically any other big general-purpose mapping app) has many more functionalities and a huge feature pool, powered by carefully engineered algorithms, with their own infrastructures, API's and many years of development, just like most of its competitors. By no means Smart Commute intends to be an entire substitute for these full-scale mobile applications, but instead wants to differentiate the scope: by reducing the features and the functionalities to a very low number, and by seamlessly working in the background for the user, Smart Commute intends to craft a specific solution to a problem of convenience for some personas out there.

By defining two features: managing of appointments and view these appointments on the calendar and the "roadmap", there will be coverage of many functionalities. Particularly managing appointments, which lets the user to create, delete or update events. It triggers a deep algorithm that automatically plans all the routing for the user. As events are being created or updated by the user, Smart Commute outputs

feasibility, the route calculations, the available time left between appointments considering the commuting time, the whole graphic representation of the schedule and any time there's a change in these events, the app recalculates the routes. In the following sections the technicalities on this engine will be further detailed.

There are no features related to landmarks, exploring places, user reviews, messaging to businesses, business hours, places ratings, recommendations, offline maps or location sharing. Just a calendar interface integrated with a map with geolocation, places and routing services. In short, the features and functionalities have been narrowed down to a limited set, dedicated to a smaller proportion of the market.

Who would want to use Smart Commute?

Smart Commute is an app for the realtor, the entrepreneur, the busy law student. Basically, fits perfectly for anybody who must go from one place to the other every workday and attend meetings or errands.

Why would they want to use it?

Because Smart Commute is the dedicated solution to a dynamic work environment that constantly demands relocation several times a day. As it has been previously stated, there are dozens of solutions out there that provide the same functionality, however, it is not focused on a specific work environment and instead it normally is a general-purpose scheduler. The default calendar apps are effective in organizing events, however the lack of integration with a visual representation of such events allows for a dedicated solution.

What are they looking for?

This persona wants to readily open the browser in their office/outside, get to smart commute and check the timing for the next appointment, how much time is left to go, and which transport means to take. Also, it wants to be able to dynamically plan for the day as it goes and change, delete or create new events at any time, so that Smart Commute assures a smooth workflow.

How does Smart Commute provide the utilities that solve their problems?

It integrates a fully functional calendar with a mapping API and a routing API and uses a crafted algorithm to show in real time where and when to go according to the user's plan of the day.

Verdict

According to the arguments presented, it is plausible and possible to implement a web app, which would be compatible with any device that can access a browser, and this way the app usage is generalized to any possible computer with access to internet, maximizing compatibility and accounting for the idea of migration of mobile software to the web. Furthermore, this assures usage in any platform: visualize the agenda on the screen, or in the laptop at a café with Wi-Fi after a meeting, or with the smartphone while metro traveling. It is not intended to be a mobile application and for the current purposes of the project, Smart Commute is centered on browser only usage, compatible with all major web browsers such as Google Chrome, Firefox, Safari, Edge and Opera.

Lastly, Smart Commute provides a different approach to a problem a considerable group of users potentially have, as seen before, due to the generality of these state-of-the-art applications that provide lots of functionalities to cover up most of the demands, compromising on certain modalities that require

simplicity and practicality and that to accomplish it, this app narrows down to the basic features wanted and minimalizes (and optimizes) the entire commuting experience.

Objective

- Develop a scheduler web-app with a calendar interface that computes and accounts for travel time between appointments to make sure you're never late for an appointment.

Requirements

Use case diagram

In the figure (1.2) the general use case is depicted in order to identify and organize the main system requirements, highlighting the core actions that the user can perform within the application, and the interaction with it.

In the Use case, the actor can perform the login/logout or sign up to access or leave the main page of the system, to visualize and interact with the schedules and events indicated in the calendar (create, delete, update events) as well as modify the views of calendar (daily, weekly and monthly). Additionally, the user can review the events, its details, and routes over the Roadmap (which have been automatically generated according to the user's event times and locations). Lastly, the user can set the preferred transport means, lunch time and break times in the Settings section that serve as the user's app customization.

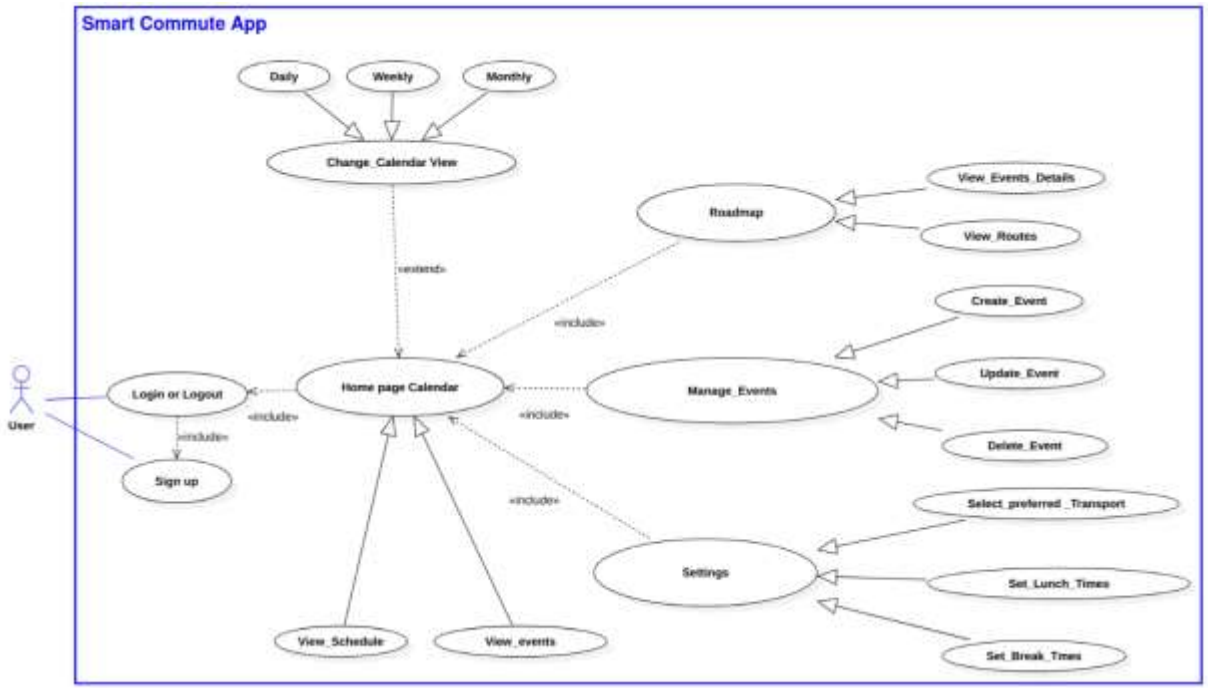

Figure 1.2: Use Case Diagram

Based on the analysis of use case diagram the requirements were classified into five groups: the user authentication requirements, calendar requirements, roadmap requirements, user settings requirements and Smart Commute requirements, which will be described in detail in the following subsections.

User Authentication Requirements

REQ. 1.1 The app will allow the user to access Smart Commute using an account

REQ. 1.1.1 The app permits to register new users using email and password

REQ. 1.1.2 The app authenticates user's credentials

REQ. 1.1.3 The app permits the user to login and logout

REQ. 1.1.4 The app helps the user to retrieve lost credentials

REQ. 1.2 The app will permit multi-user access

REQ. 1.2.1 The app provides to each user its own environment

REQ. 1.2.2 The app allows to log in different users simultaneously

REQ. 1.2.3 The app permits the users to access the platform from different browsers

REQ. 1.3 The app must communicate with the database to interact with each user's data

REQ. 1.3.1 The app must load each user's data from the Database for later use

REQ. 1.3.2 The app must be able to store each user's credentials in the Database

Calendar Requirements

REQ. 2.1 The app must provide a Calendar Interface that shows the user's events and permits the user to interact with them

REQ. 2.1.1 The app allows the user to change the calendar views to: Monthly, Weekly and Daily

REQ. 2.1.2 The calendar will allow the user to pick a day or several days from the calendar views and prompt the popup for event creation

REQ. 2.1.3 The calendar will allow the user to pick a previously created event and prompt the popup to edit event for event modification

REQ. 2.2 The calendar must provide a graphic representation of all user events

REQ. 2.2.1 The user's events must be represented on the calendar views with a tag indicating the start time and title

REQ. 2.2.2 The user's events on the calendar must show details including the full title and start and end times whenever the user hovers on the event

REQ. 2.3 The calendar must provide a form popup to manage the events

REQ. 2.3.1 The popup must allow the user to input the following data: title, start and end times, location and description

REQ. 2.3.2 The popup must allow the user to delete/create/update events and to cancel any of the operations

REQ. 2.3.3 The popup must avoid the creation of any event with an empty Title

REQ 2.4 The calendar must store and update the user's events in the Database

REQ. 2.5.1 The calendar must save/update to the Database the event data created or modified by the user

REQ. 2.5.2 The calendar must retrieve each event data from the Database and render it onto the calendar interface every time the user creates /modifies an event

REQ. 2.5.3 The calendar must retrieve each event data from the Database and render it onto the calendar interface for every user login

Roadmap Requirements

REQ. 3.1. The Roadmap will have a map and a daily calendar of the current events

REQ. 3.1.1 The map must have the following utilities: zoom in, zoom out, map layers, display routes and show route checkpoints (markers)

REQ. 3.1.2 The Roadmap must retrieve all the events from the database to calculate the routes, and render both the events as markers and the routes as polylines

REQ. 3.1.3 The daily calendar will display the user's day events, its details upon hovering, and show its details on the map marker on clicking

User settings Requirements

REQ. 4.1 The app will provide a settings menu that allows the user to modify the preferences

REQ. 4.1.1 The settings menu must permit the user to select the available and favorite transportation means (bus, car, tram, metro, bike, walk)

REQ. 4.1.2 The settings menu must preload the default settings onto the interface, or the saved user settings from the Database (if any)

REQ. 4.1.3 The settings menu must allow the user to specify and toggle on/off break and lunch times

REQ. 4.1.4 The settings menu will allow the user to save & exit back to the main page, submitting the settings data and saving it in the Database, or to go back to main and cancel the operations.

Smart Commuting Requirements

REQ. 5.1 The Smart Commute must compute and validate time of the events

REQ. 5.1.1 The Smart Commute must check that the event start time is not greater than the end time and forbids its creation/update if this requirement is not fulfilled

REQ. 5.1.2 The Smart Commute must avoid that the event's start and end conflict with the lunch or break times for weekdays

REQ. 5.1.3 The Smart Commute must check that the event time of the event to be created/updated does not overlap with the existing events' start and end times

REQ. 5.2 The Smart Commute must account for travel times for event validation

REQ. 5.2.1 The Smart Commute must calculate the route times for each event pair and user's current location, considering the user transport preferences

REQ. 5.2.2 The Smart Commute must check if the travel times are smaller than the time available between events to allow event creation/update

REQ. 5.2.3 The Smart Commute must update the travel times every time that the user modifies the location and times of the events and settings

General Web-App System architecture

The overall system architecture is illustrated in the fig. 1.3, where different users access simultaneously through a web browser to the Web-app, hosted in the application server, making a request to the server as a first step. The user's login credentials are stored in the database and after the user authentication the app server interprets the user data provided for the database and it is applied to the web app, as a last step, the application server returns the result of that process to the web browser to be displayed. The communication between client and server is generated by an exchange of messages until all the requests are met and the users can visualize the information requested.

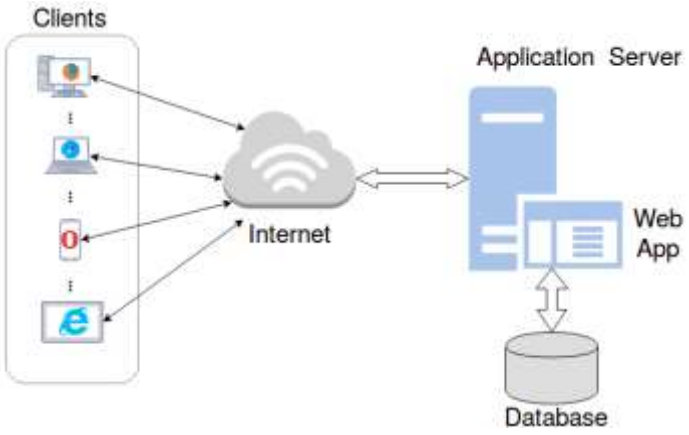

Fig. 1.3. System architecture

In the subsequent sections the Web-app design will be depicted in detail with the use of class diagrams and Sequence diagrams.

App design

In order to describe the structure, attributes, operations (or methods), and the relationships among objects of the web app, several entity relationship diagrams have been drawn based on the requirements and the general use case previously presented.

High Level Class Diagram

Firstly, a high-level class diagram is drafted in order to link the use case with an overall idea of the application interaction with the user:

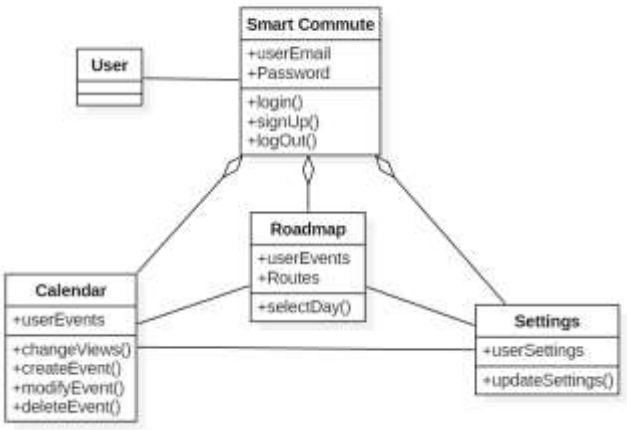

Fig 2.1. High-level class diagram

The user connects to the application by executing login or signup operations, and by providing email and password. After this step it's possible for the user to access all the software features such as the calendar, viewing/managing events, the roadmap to view the events on the map and their routes, toggling between days, and the settings to update the user settings with the desired configuration input.

Front-end Class Diagram

The diagram of figure 2.2 represents the graphical user interface as well as the main app functionality, as it is more oriented to the front-end side of the software implementation. The main classes contained in this class diagram are the following: GUI User, contained by the interfaces of Map, Calendar and Settings, the Popup interface which allows for user input, and the calendar views.

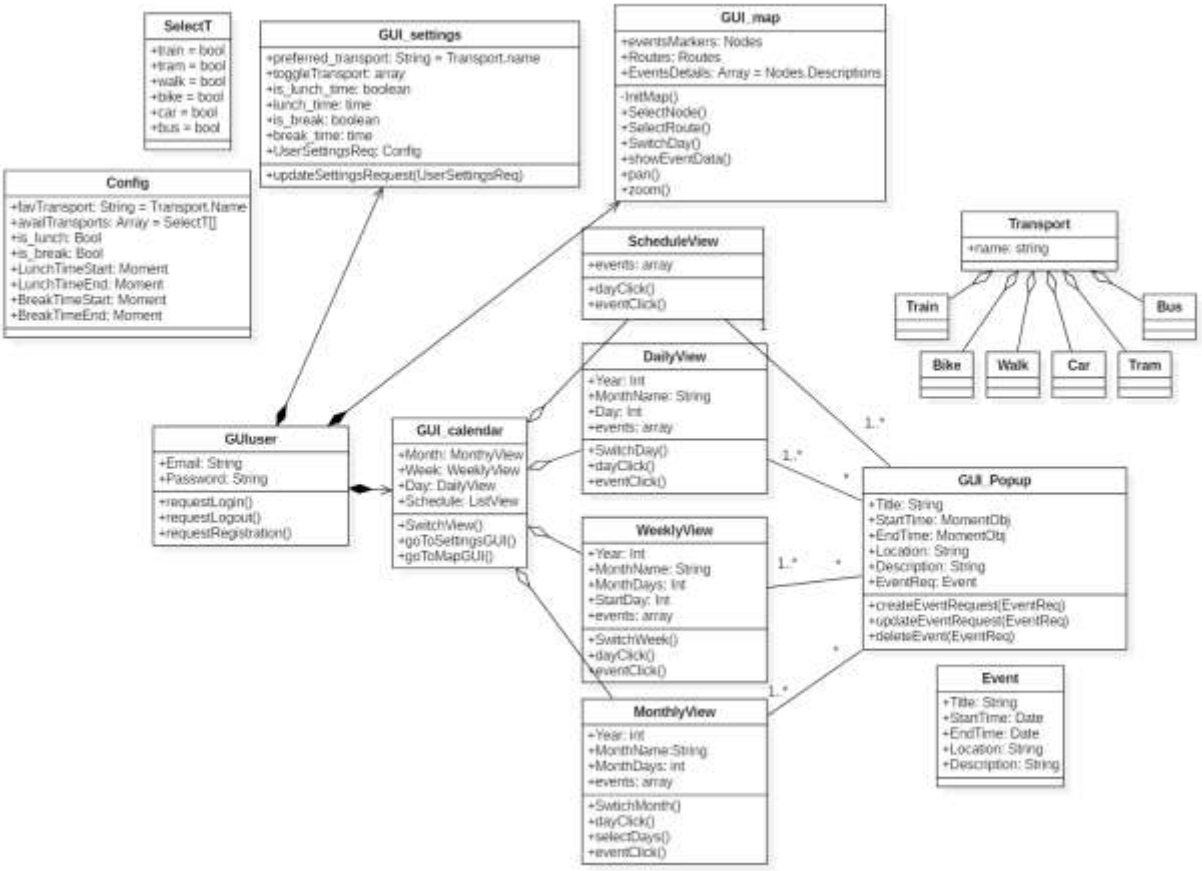

Fig. 2.2. Front-End Class Diagram

GUI user class

The GUI user class is the initial interface the user interacts with upon access to Smart Commute. The user has the possibility to request a login and allow the app to receive user credentials that will be sent for authentication, a logout (If already in the app) and registration (if no account has been created) which will send a request with the user input of Email and Password and create a new user. It's the main class and it manages the user information to access the application.

GUI user is composed by the settings, the map and the calendar interfaces. These sections define all the tasks and operations the user has access to. First, the calendar (which is composed by different view options) allows to create single day or multi day events with the function $dayClick()$, as well as clicking existing events with the operation $eventClick()$. Both operations trigger a modal (a popup, represented by the class GUI popup) where the user can input all the necessary event details it wants to create or update, or if it wants to delete certain event.

The calendar GUI also allows the user to toggle views for day, week or month. It has the subclasses of Schedule view (lists the events in order of recency), daily view (shows the current day's event, defining at what time of the day) where days can be switched, similarly with weekly view and lastly Monthly view, having the additional operation of selecting multiple days to create a multi-day event.

Secondly, the map GUI initializes a map from the map's API, which contains the rendered events shown as markers on the map, as well as the generated routes rendered as polylines. It lets the user pan or zoom the map and click on the routes or maps to show their details, as well as viewing and clicking the day's events on the today section. Additionally, it lets the user toggle between days to see the following day schedules. By default, the map GUI shows the current day's schedule.

Lastly, the settings GUI lets the user through input fields define its personal configuration, such as the favorite transport means, the available transports to use and the lunch and break times. This class additionally saves this configuration and sends a settings update request to the app engine.

GUI popup class

Triggered from the operations contained in GUI calendar of $eventClick()$ and $dayClick()$. It listens to the user input data for the event fields of title, description, start time, end time and location. Upon event creation or update, the operations $createEventRequest()$ or $updateEventRequest()$ are performed accordingly sending a request to the app engine where it will, after validating, create or update the event in question. Also, it can ask for event deletion with $deleteEvent()$. Update event and delete event options are only available for when the user clicks an existing event on the calendar interface.

Other Classes

Additionally, there's secondary classes which aid the diagram to be more readable, such as $selecT$ that represents an array of Booleans for the allowed transport means to be used for the routing calculations, $Transport$ which is a string for the transports available (to input into settings), Config that returns an object with all the user configuration data, composed by favorite transport of the class transport, available transports of the class $selecT$, lunch or break activation values (Booleans), and lunch and break start/end times (date values. Lastly, the Event class which represents the format for any event object created/to be created, specifying title, start time, end time, location and description.

Back-end Class Diagram

A second-class diagram was designed to depict all the user's information management between the Database and the application. The following class diagram has 2 main classes: Firebase (composed by the Authentication and the Database), App Engine superclass which abstracts all the main app functionalities that require interaction with the database, having the user, settings, maps and calendar engines as subclasses. There are other classes for better appreciation such as $hereMapsAPI$, $FullcalendarAPI$ (both depicting an association with the programming interfaces of these external sources), $SavedData$ composed by the saved events as an array of JSON and the saved user configuration as a JSON, Coordinate specifying the latitude and longitude retrieved from the maps API, Config which is the same class as the one in the front end diagram for better comprehension, Nodes which has as attributes all the event's information rearranged in arrays per category, Routes that specifies the routes objects generated by the Maps Engine, and finally Event, previously shown.

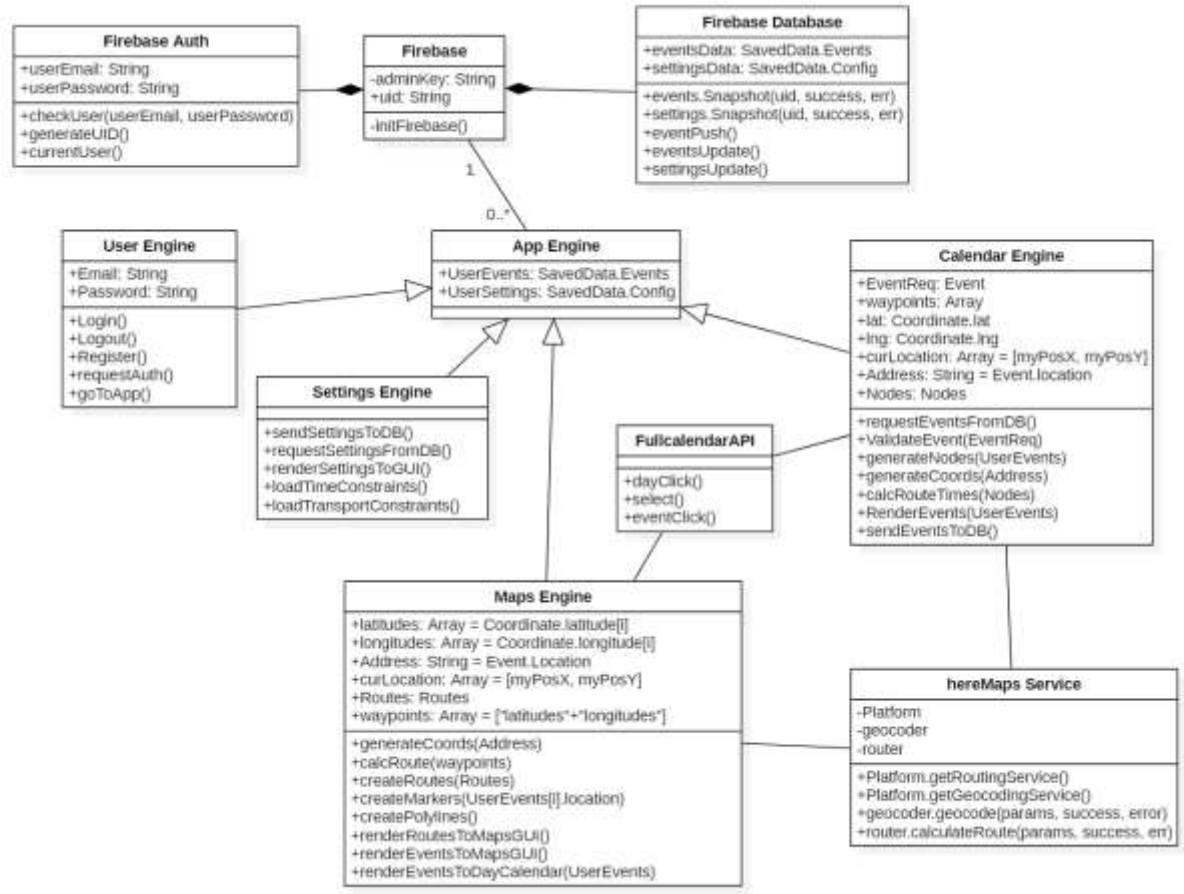

Fig. 2.3. Back-End Class diagram

Firebase Class

To link Firebase to a web application, an admin key is autogenerated and used to initialize. All the backend operations will be available once the Firebase client initiates with the operation $initFirebase()$, and they are contained either in Firebase Auth or in Firebase Database.

Firebase Auth receives the user login/register/logout requests. It generates a user ID upon registration requested by the user engine and saves the credentials in the database, checks the user credentials whenever the user engine calls the login operation with the method $checkUser()$. Firebase auth additionally verifies the user is logged in with its method $currentUser()$, necessary for event and routes validation purposes.

Firebase Database instead manages all the information related to the user. Contains events data (an array of JSON objects containing each Event's attributes) and settings data (a JSON element containing all the config attributes) per each user, and whenever there is a request to obtain such information the $snapshot()$ operations are used to retrieve them. Events or Settings snapshot is a Realtime Database method from the Firebase client, which retrieves a snapshot of any data one time only.

The database is also responsible for updating the user information, by using the operations $eventpush()$ (to add new events), $eventsUpdate()$ for modification or deletion of events, and $settingsUpdate()$ for a change in the user settings. All these operations are executed via Firebase methods, further explained in this documentation.

The following figure (2.4) depicts a standard structure for a set of events as a JSON and the structure syntax for the settings data (showing the default data) used in the application and the Database:

Figure 2.4. Standard Syntax for Event objects and Settings data.

App Engine Class

The app engine is an abstraction of all the app functionalities that require interaction with the Database. It contains the user settings and the user events (retrieved from Database) and it is a superclass with the following subclasses: User Engine, Settings Engine, Maps Engine and Calendar Engine.

User Engine Class

Receives from the user input the credentials, contained in the email and password attributes and provided by the front-end request operations for login, logout or register. The user engine class does these operations by passing the email and password via $login()$ to $checkUser()$ or register via $Register()$ to $generateUID()$ operations from firebase. In practice, these operations from the user engine have the firebase auth operations nested in the algorithm. If the user ID is needed for verification purposes, User Engine has the operation $requestAuth()$ which asks for these values from Firebase. Lastly, the operation $goToApp()$ redirects upon successful authentication to the user's app home page.

Settings Engine Class

Essentially listens for $updateSettingsRequest()$ from the GUI and sends the request with the operation $sendSettingsToDB()$ whenever the user updates its configuration. At any app initialization, settings engine requests the config file with $requestSettingsFromDB()$, allowing the Database to provide the settings data ($UserSettings$ attribute) to be used by the different engine modules. The settings data is then rendered onto the interface with $renderSettingsToGUI()$ so they are shown anytime the user enters settings. Finally, the operations $loadTimeConstraints()$ and $loadTransportConstraints()$ pass the saved settings information to the different engine modules for event or route validation purposes. It is important to mention that upon sign up there is no user settings defined. The app instead loads a static default settings object that establishes the default user configuration from Smart Commute.

Calendar Engine Class

This class has the operation $validateEvent()$ which prevents incorrect inputs such as empty title or incorrect times, followed by verifying no time overlapping between events, resorting to Firebase events snapshot (using $requestEventsFromDB()$) and comparing each of the user events' times and returning a true or false value. After this step, the calendar engine then $generateCoords()$ by requesting to the Here Maps Service the geolocation method the latitude and longitude values, which are then used (along the user location and adjacent events) to $calcRouteTimes()$. For easier code comprehension and to assure correct input syntax on the routing service, by using the $UserEvents$ values, the operation $generateNodes()$ rearranges all the information from each event by appending each category of the events into different arrays.

Additionally, these route times are then compared with the time gaps between events, checking event addition feasibility accounting the traveling time. Validation is executed once the user requests to create or edit an event from the GUI, passed with the attribute $EventReq$. Also includes the operation $RenderEvents()$, rendering the $UserEvents$ data once validated by the calendar engine and the maps engine. Lastly, whenever events are fully validated, the operation $sendEventsToDB()$ asks the database to push or update the new event data.

Maps Engine Class

Maps engine has the latitudes and longitudes of the event locations as attributes, obtained with the method $generateCoords()$ with Address as the input parameter, which corresponds to the events location string. This operation resorts to the Here Maps Service class to match the user's address input and returns a geolocation coordinate value pair (latitude, longitude), via the operation $geocoder.geocode()$.

Then each event location, as well as the current user location, are appended into a waypoints array which contains $k + 1$ elements, being k the total number of daily events. The operation $calcRoute()$ requests to Here maps routing service in the same way as in the calendar engine, to obtain a route object from a pair of coordinates. It takes element pairs sequentially from the waypoints array, creating k route objects which contain the route time, the transportation means, the start and end points, and the routing instructions.

Finally, for each event element and route element created, the maps engine creates the objects to be then rendered onto the GUI. These are: $createMarkers()$ for events and $createPolylines()$ for routes. The events are then finally rendered with $renderRoutesToMapsGUI()$ and $renderEventsToMapsGUI()$, and onto the day calendar with $renderEventstoDayCalendar()$.

Components Class Diagram

A minor class diagram was extracted with all the variable types used throughout the application. Some details might have been omitted, but as for the main syntax and structure, all the relevant information is contained:

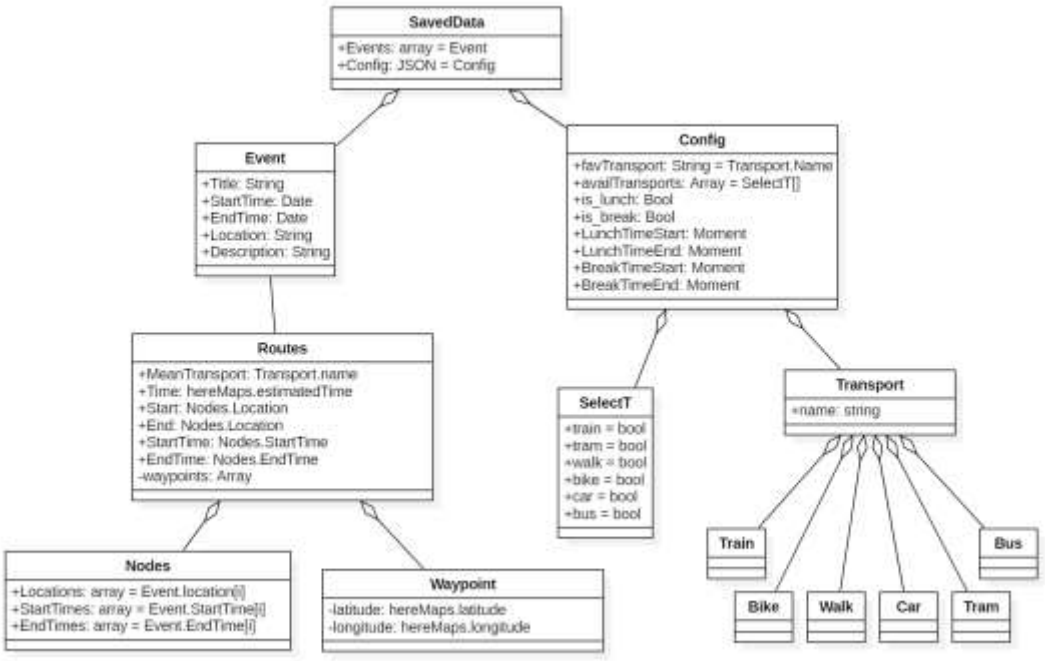

Fig 2.5. Components Class Diagram

The Event and Waypoint are used in several main classes' attributes. Nodes is the type of class for the rearrangement operation that generates nodes, $Routes$ similarly represent the structure used for the route objects created by the map engine. $Transport$ is a string value which equals any value of its sub elements. $SelectT$ is instead an array of Boolean values that indicate which transports to consider for route calculations. $Waypoint$ is a class with latitude and longitude as components, and Config wraps all the settings data. Lastly, $SavedData$ organizes the events as arrays and inherits the settings data as a JSON.

Sequence diagrams

Log in to Web-App and load user data to the environment

The diagram depicted in the figure 2.6 represents the scenario where the user logs into Smart Commute and his events and configuration data are loaded from the database into the work environment of the app after authentication.

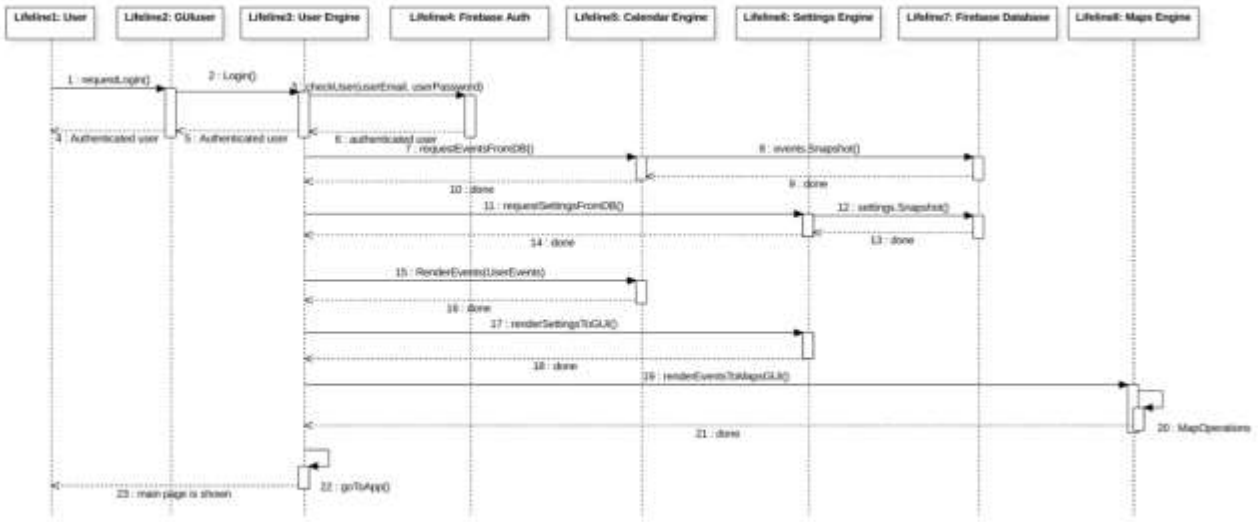

Fig. 2.6. Log into the app sequence diagram

The interaction between parts of the system to carry out the tasks is described as follows.

1. The user submits its email and password through the GUI User's input field and requests to be logged in.
2. The GUI User transmits the request to the User Engine.
3. The User Engine requests the authentication of the user email and password to the Database (Firebase Auth).
4. The Firebase Auth starts the process of user authentication and replies to the User Engine and then the User Engine to the GUI User and finally to the user.
5. The User Engine requests the events contained in the Database to the Calendar Engine.
6. The Calendar Engine asks to Firebase Database the events snapshot.
7. Firebase Database completes the request and confirms to Calendar Engine and afterwards Calendar Engine confirms to User Engine.

8. The User Engine asks to render the events into the GUI Calendar to Calendar Engine, the end of this process is confirmed to the requester.
9. The User Engine asks for render the settings into the GUI Settings to Settings Engine, the end of this process is confirmed to the requester.
10. The User Engine asks for render the events into the GUI Map to Map Engine, the end of this process is confirmed to the requester.
11. After all these processes the User Engine permits the user to go to the App and shows the main page.

Create an event

The Figure 2.7 exemplifies the scenario where the user interacts with the calendar interface to create an event, as well as its validation process carried out with respect to the user's default configuration and the geographical location where the events are carried out.

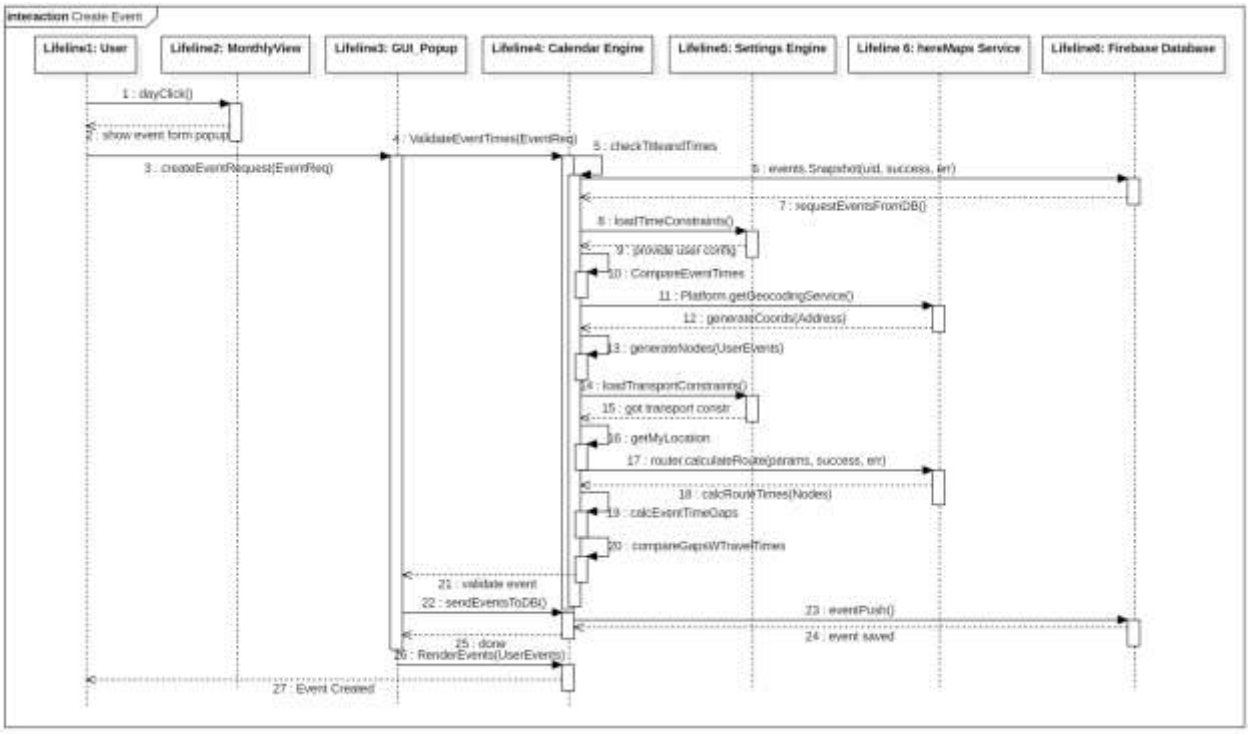

Fig. 2.7. Create an event sequence diagram

1. The User Clicking over one day or interval of days from the GUI Calendar (e.g. Monthly view) launches the event form popup where the data of the event will be introduced.
2. Then, the user introduces the event's information and requests the event creation using the GUI Popup.
3. The GUI Popup asks to the Calendar Engine to validate the event.
4. The Calendar Engine then checks that the user has introduced a valid title and a consistent event duration.
5. The calendar engine requests to the firebase DB a snapshot of all the events previously saved by the user, and to the settings engine to provide the time constraints of lunch and break times.
6. The calendar engine then compares event times to prevent overlapping with the existing ones.

7. Then asks the Here Maps Service to generate the coordinates of the address via the method of geocoding.
8. Here maps service replies the coordinates and the calendar engine generates a coordinate object with it. After this, it generates the event nodes, extracting only the start, end times and location and sorts them in time.
9. The calendar engine then loads the transport constraints by the user configuration from the settings, and afterwards retrieves the user location.
10. Then it requests the Here Maps routing service method to calculate the route times between the user location and each event location sequentially, calculates the time gaps between events and compares it to the travel times between the user and each of the events, and finally it validates the event which initially has been requested by the user.
11. The Calendar Engine replies to the GUI Popup that the event has been validated.
12. A request is made to send the event to the Database, then confirms action has been completed.
13. Event is then rendered onto the GUI, and the popup window is then closed.

Modify and Delete Event

The scenarios of creating and modifying an event can be considered as variants of the previous sequence diagram since modifying an event is considered as an overwriting of the data and leads to the same validation processes performed in the scenario create event. In particular, the scenario to delete an event contains the same path performed for the two previous cases (create and modify) except that this process does not contain the validation stage and save in the database, in substitution it contains the delete command.

Select Event on Map

The Figure 2.8 shows the scenario where the user interacts with the interface to view an event in the roadmap, its details and a list of all the current day's events in the right side of the interface where the day's events are shown.

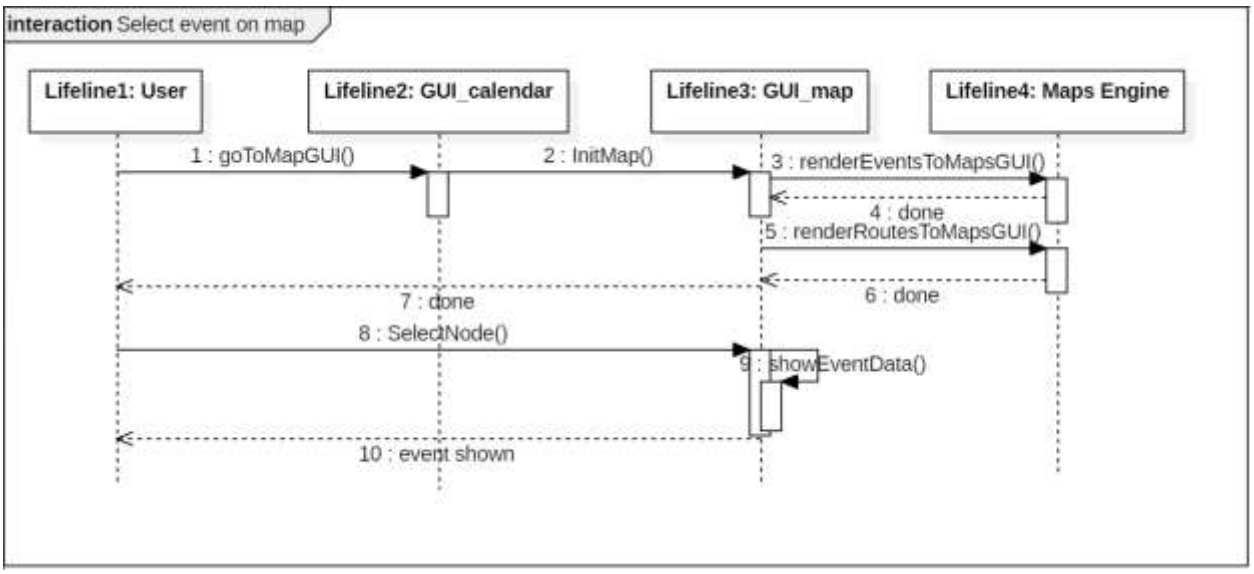

Fig. 2.8. Select current day event on Map

1. The user clicks on the option “Roadmap” located in home, triggering $goToMapGUI()$ operation from the calendar interface.
2. Then the interface requests the map interface to initialize the map via the operation $InitMap()$.
3. Once initialized, all the event items are rendered onto the day view calendar in the right side, and afterwards onto the map, with the operation $renderEventsToMapsGUI()$ from the map’s engine.
4. Similarly, the already calculated routes are rendered onto the map interface by the map’s engine.
5. Once both operations are performed, the rendered items are shown to the user and the GUI is fully loaded.
6. Lastly, the user requests an event information by clicking an event marker (a node) in the map. The GUI reacts to the click with $SelectNode()$ operation.
7. The map interface triggers a popup with the specified event details.
8. Lastly, it is shown to the user in the interface.

Flowchart Diagrams

What happens once the user requests an event creation or update?

Through the sequence diagrams we have shown each of the processes performed by the app once a user requests the creation or update of an event. Part of the app intention is to allow the user to have an effortless interaction whenever it wants to update the day schedule. The idea and what's expected from the user persona is to want to enter the application, so it updates today's roadmap, checks it and goes back to its routine.

To make this process as simple as possible, the Smart Commute algorithm intends to work around everything behind the interface and output the most convenient answer. In practice, the user really has three main tools (the only ones it needs): to create, edit or delete an event. This simplicity is intended to be accomplished by crafting an algorithm that does all the tasks behind curtains for the user, many of those are commonly asked by other mapping apps which probably aren't of outmost importance, and instead might waste time.

To further understand how this is done, the following flow chart diagram orients the reader on the algorithm behind the Calendar Engine and Maps Engine classes previously explained, so that the overall idea is understood:

Figure 2.9. Event creation/update flow chart

The event creation or update first checks within the same interface whether the user as input a title. Following this, the script calls the app engine to verify the event feasibility through a series of filters. Initially checks if the start time comes before the end time, then it verifies that the event time does not collide with any other event created previously by the user. Once no collision is assured, the algorithm checks if the event has passed to write it immediately or, if it hasn't, to consider the travel times. The event in question is then used as a pivot to calculate the route between the previous event (or the user's location, if the event is upcoming) and the following event, with the preferred transport means. These two travel durations are then compared with the time gaps previous and after the event. In practice, It would be the time between the end of the previous event (or the current time, if event is upcoming) and the start of the event to be created, and the time between its end and the start of the following event (if any). If there's time to travel to the event and from it to the following one, then ultimately, the event is created/updated.

How are the events and their routes rendered on the roadmap?

A brief flow chart diagram shows the workflow of the code which renders all the upcoming events on the roadmap interface:

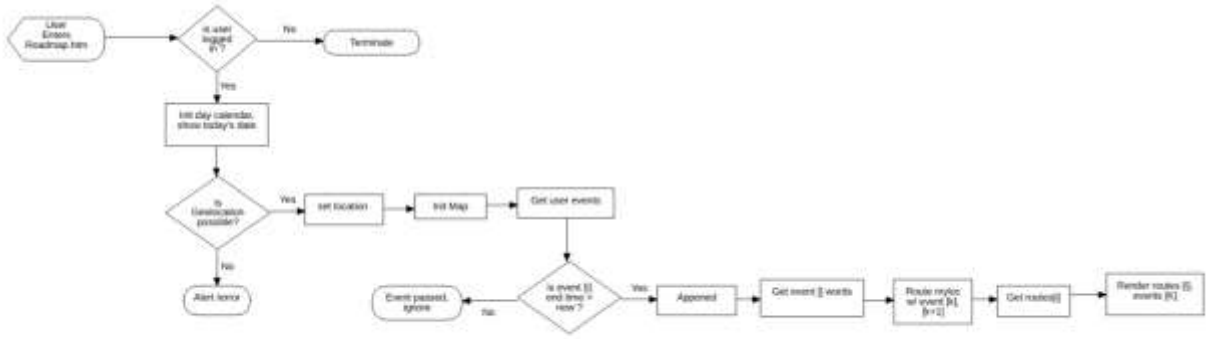

Fig 2.10. Event and route rendering on the roadmap interface

The user events have been previously validated with the event addition/update algorithm, ensuring no invalid routes are calculated (for a more advanced methodology for the roadmap script, check the section conclusion and future work). Therefore, after checking the authentication, the roadmap engine just needs to retrieve the current user location to initialize the map, get the user events, filter to only upcoming events and finally sequentially obtain the geolocation for each event plus

How are the user settings saved?

The following flowchart explains in a simple way the workflow of the code once the user enters the settings interface:

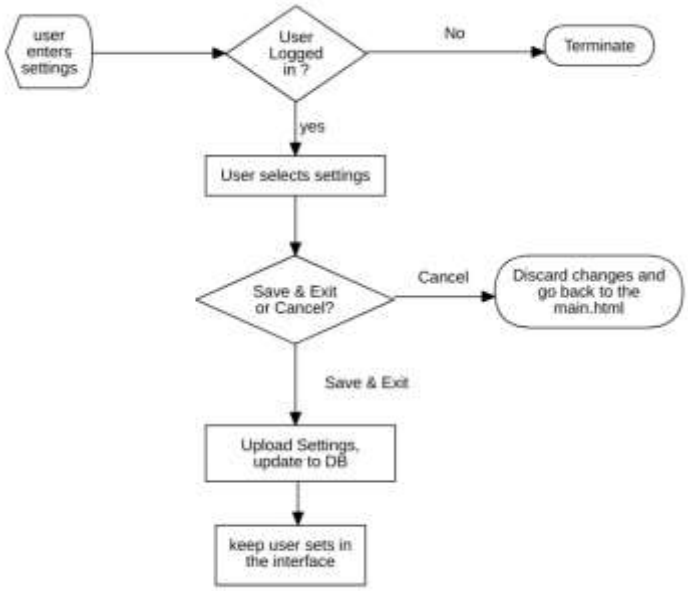

Fig 2.11. Settings interface

Once the user enters the settings interface, as usual the app verifies that the authentication is positive. After having selected all the preferences, if the user inputs the cancel button then the data is discarded. Otherwise an update in the Database is done, as well as an update in the settings on the GUI.

Implementation

Graphical User interfaces.

GUI User (authentication interface)

The authentication interface is the first interface of the web app, which has the function of authenticating the user through the request of his credentials (email and password) in case of registered users, if the user is not registered in the application it must create an account by providing his email and generating a password. The figure xxx shows the two options mentioned before, registered user and add account.

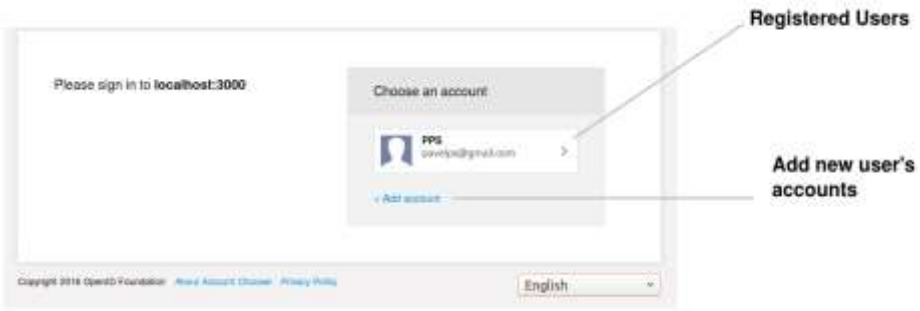

Fig. 3.1. User GUI – account selection

In the figure 3.2, the dialog form is shown to submit the email and password for registered users or to add a new account and the button Sign in, which permits to access the main page and loads the configuration and events data of the user in the web app.

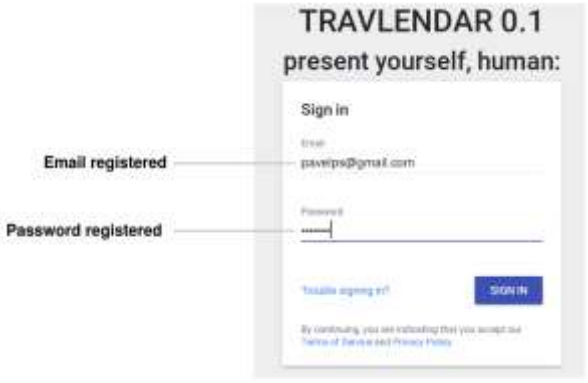

Fig. 3.2. User GUI - authentication

GUI Calendar (Main Page)

The main page interface represented in the Figure xxx is integrated by the calendar and its view option (Monthly, weekly and list or daily), the options to access to the configuration of user's preference, visualization of the routes-events in the map and the log out.

In this interface is possible to select the dates to generate the events using a popup dialog form (which will be explained in the following subsection), visualize it and present a brief information of the event

when the mouse is over the event. In addition, permits to change to the other 2 functionalities of the app (road map and settings) and log out through the upper buttons shown in the figure 3.3.

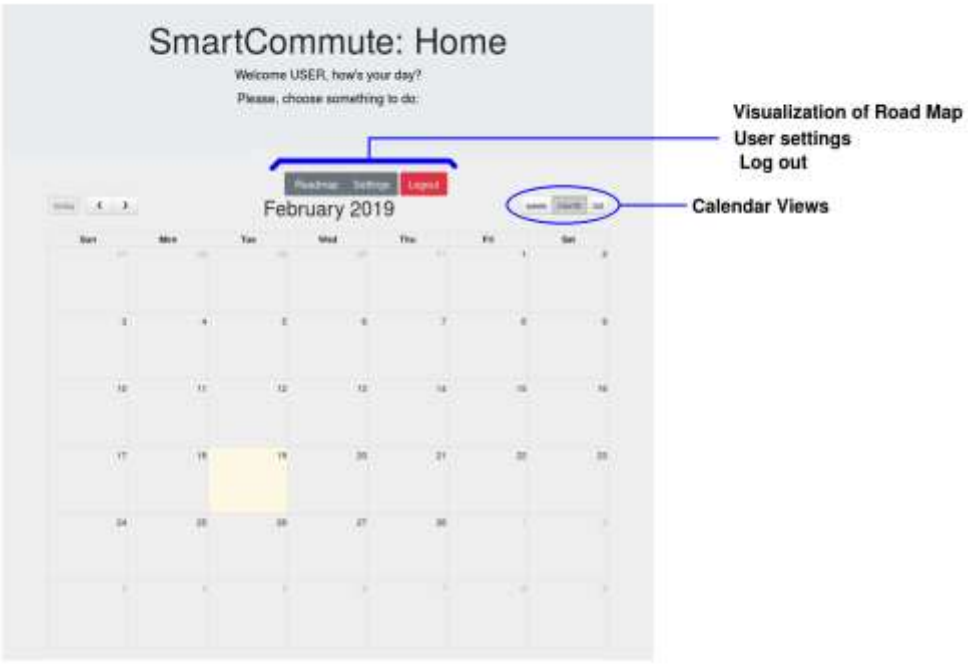

Fig 3.3. Smart Commute Calendar GUI.

GUI Popup (Event Dialog Form)

This interface is a Popup Dialog form where the user introduces the data of the event (Title, Location, start time, End Time and Description) in the fields to create, update of delete an event. The figure 3.4 represents the case of a creation of an event where it allows for the event creation request or to cancel the operation. On the other hand, once an event is created it can be updated or deleted by clicking on the event itself in the calendar GUI. A similar popup is prompted in this case, and it shows the options that allow for event modification, plus the options update, delete and to cancel the operation, as shown in the Figure 3.5.

Fig 3.4. Event creation popup

Fig 3.5. Event modification popup

GUI Settings

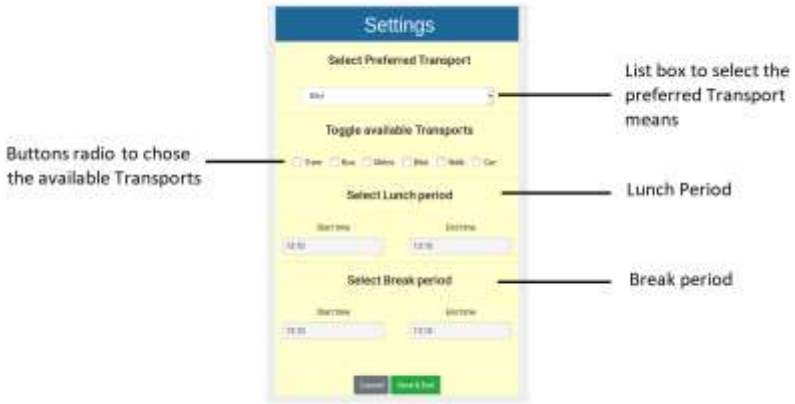

Fig 3.6. Settings GUI

The Settings interface is illustrated in Figure 3.6 and permits the user to select the preferred transportation means, the lunch time periods and the break time periods to be considered. These times are by default set on all weekdays and do not contain a specific place, just provide a window for the user to have a time out.

The data introduced in this interface will be stored in the user's data base and used to compute the events, routes and travel times.

GUI Road Map

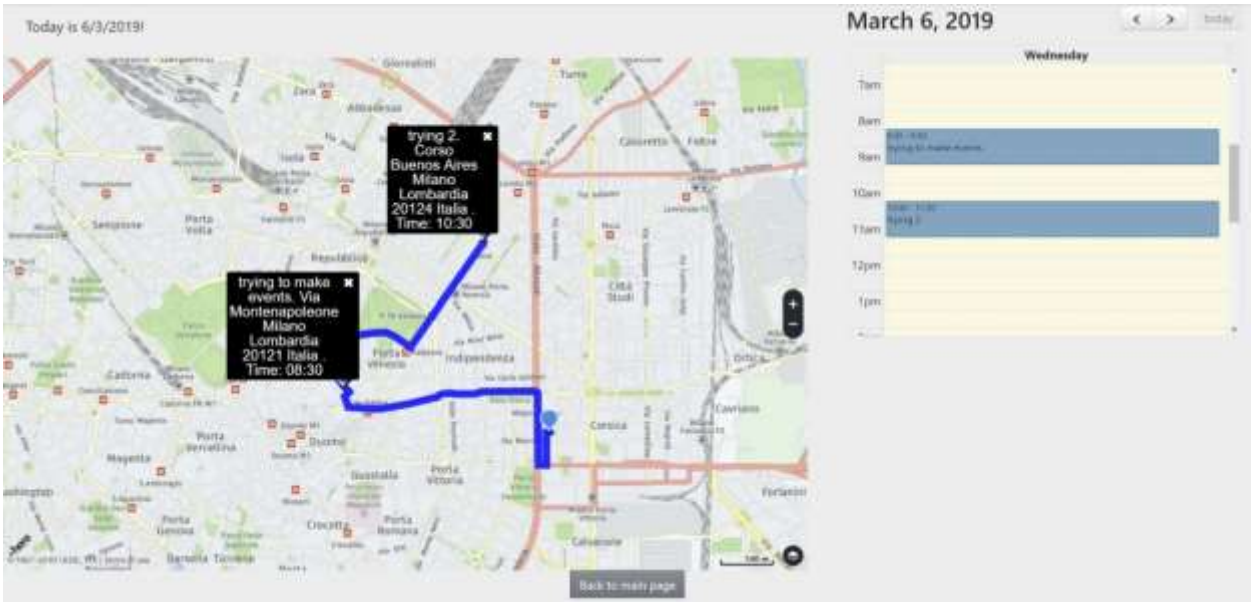

Fig 3.7. Roadmap GUI

The Road Map interface is integrated by two main components, the first one is an interactive map with basic functionalities. Once the map engine preloads the events, they are then rendered onto this map as markers, and its calculated routes as polylines.

The second component consists of a daily calendar that contains the events of the day and any time an event is clicked, it shows the details on the map from its correspondent marker as an info popup. In addition, two arrow buttons are integrated in it, which allow to switch the views of the events on the map per day.

Firebase Database Console

This graphical interface is not intended for the user but instead intended for the software administrator. In it, it's possible to manage all the created users that have an account in Smart Commute, along with their data objects, namely the event objects and the settings objects. Each user address index is defined with its user ID generated from the Firebase authentication system, directly integrated to the console and automatically communicates the service with its Database extension. In the following figure the Firebase Database GUI can be appreciated, as well as an example of a user's data structure entries. Note that additionally, each event has its own ID, generated from the Firebase methods used to push objects into the user entries and sub-entries, in this case, under the label 'events'.

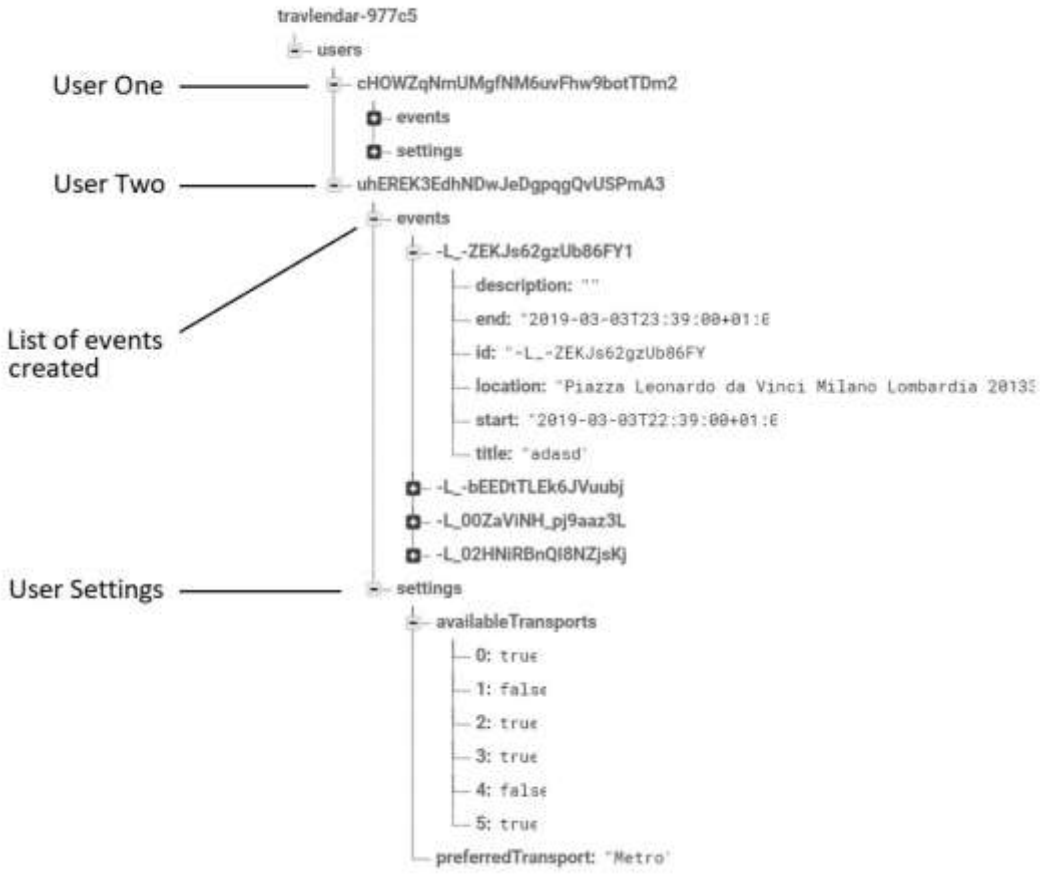

Fig 3.8. Smart Commute Database structure, admin graphical interface

Structure of the code

Smart Commute is ordered in a repository on GitHub. The filesystem differentiates the compiled code, the source code, the documentation, UML diagrams, libraries and modules from external frameworks. All scripts and codes are sufficiently commented. The following scheme shows the relevant code where the core functionalities reside:

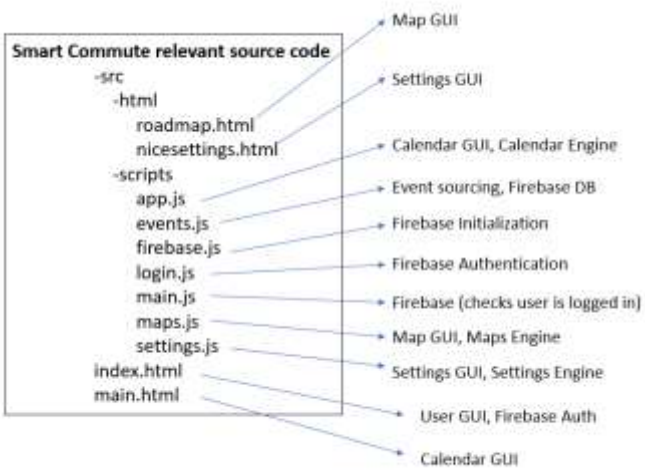

Fig. 3.9. Smart Commute relevant source code

These scripts contain the core functionalities of the code. There are other several compiled script files and old versions of other scripts being unused, which will be further eliminated once its inside content is not useful as a guide for the developers anymore. In the following section the main scripts will be described, matched with the classes in the class diagram (particularly the back-end class diagram) with its operations and attributes.

App.js

Contains the main functionalities of the calendar, as well as the algorithm outlined in the event creation flowchart. It bears all the front-end functionality of the main calendar in the homepage as well as all the algorithm linked to the event addition/modification criteria.

```
Initializes FullCalendar ({day click, and event click functions});
If user clicks a day {
        Set values ();
        If user clicks submit {
                Define the event data;
                Check if event has title;
                Validate event ();}
If user clicks an event {
        Set values ();
        If user clicks update {
                Define event data;
                Check if event has title;
                Validate event ();
        If user clicks delete {deletes event};
--some secondary functions--
Function: Validate event () {
          If event times are correct {
                Retrieve other events
                Check no time overlap between any event
                Generate nodes
                Geocode event addresses
                Calculate route times
                Check there's time to travel for previous and following event
                Create / Update event};
```

Fig 3.10. App.js pseudocode

This script contains all the attributes and operations from calendar engine within the backend class diagram. It requests the events from the database, validates event times, generates nodes, generates coordinates per event, calculates the route times and creates / updates event by writing it into the database and ultimately rendering it onto the calendar GUI. These attributes are listed within the calendar engine class. Throughout the event addition process, several variables are to be created, all contained as attributes in the same class: Event request, user events, waypoints (for route calculation), latitude and longitude which is generated from the Here Maps API geocoder, the current location, address strings (used to generate the latitude and longitude pairs for each event), the nodes (which is a rearrangement of the events' start and end times as well as the address string) and finally route objects, which, in this module, the algorithm only cares about the travel time generated to be used in the event addition criteria.

It's mentioned it takes part of the calendar GUI as well since it has variables connected to the DOM from the calendar GUI, contained mainly in the file main.html.

Maps.js

Contains the main functionalities of the roadmap and is responsible of the marker and route rendering. In the following pseudocode it's shown the script outline:

```
Initializes FullCalendar ({with only day view, allow to toggle days});
Checks user authentication successful;
Gets user's current location and define as latitude and longitude;
Initialize HERE maps platform;
Obtain user ID from firebase auth;
Retrieve all user events {
        Rearrange the events, sort them from recent to old;
        Filter out events that passed;
        Declare waypoints variable and append current location as first waypoint;
        For each event {
                Get event coordinates (latitude, longitude) and append each waypoint};
        For each waypoint {
                Get event route from checkpoint pairs};
        Initialize map UI;
        For each event and route {
                Generate markers;
                Generate polylines;
                Render event as marker, route as polyline};
```

Fig 3.11. Maps.js pseudocode

It has all the attributes and operations from the map engine class contained in the backend UML class diagram. Once calendar is initialized and the user authentication check is successful, the navigator geocoder is called, providing as callback arguments the current user's latitude and longitude. Here maps platform is initialized, and the user ID is obtained from firebase. Then it retrieves all events into an array of event objects called user events and filters them to only the upcoming ones. Sorts them in order of recency and for each event, extracts the address string, retrieves a coordinate pair from the geocoder and pushes that coordinate pair to the waypoints array. For every waypoint element and its following one (starting from the user location), the routing service provides a route object with the route information. Finally, both routes and events are taken and rendered onto the map with the Here maps front end methods.

Settings.js

Contains the main functionalities of the settings and is responsible for the settings interface in overall. In the following pseudocode the settings engine is shown in a comprehensible way:

```
Initializes get settings function;
Checks user authentication and retrieves id;
Retrieves from firebase database the settings data () {
        Loads the data on front end;}
If user clicks save & exit button {
        Listens to all user config input and sets it as variables;
        Parses the input data into the Saved Data format to write to the database;
        Writes the user settings onto the interface;
        Overwrites the user settings in the Database with new ones;}
```

Fig 3.12. Settings.js pseudocode

This script contains all the attributes and operations from the settings engine such as send settings to the database, request the settings from the database, render settings onto the interface, and loads time and transport constraints. After authentication and retrieval of the user ID, the script requests from firebase the settings data object from the user. If no settings are found, a default settings object will be saved instead. Once the settings object (saved settings or default settings) is declared, it is loaded onto the user interface once it's accessed by the user (if user enters settings). If user presses save & exit button, the settings values will be submitted and overwritten in the database for later use.

Used tools

Front end tools

Bootstrap is an open source front-end web framework developed by Twitter. Bootstrap templates were used in the design of forms and buttons of the Smart Commute Web app, as they were among the most reliable and easy to use frameworks in the market.

FullCalendar (ver. 4) is a jQuery plugin used to build the calendar front end infrastructure, allowing for ease event manipulation on the interface and intuitive methods that trigger user-defined functions. One of the most complete, lightweight calendar plugins.

Here Maps (ver.) is an API that provides map rendering and permits to stablish communication with the back-end services provided by HERE REST APIs, these last ones give tools for routing, intermodal routing and geocoding. All these tools were combined and integrated to develop the map interface and the routes on the map.

Timedropper is a jQuery UI time-picker created by Felice Gatusso. This plugin was implemented in the event creation/update popups to let the user pick the time of the events graphically. Provides a better solution for time selection than the default HTML5 time element.

Firebase Authentication UI, is a UI plugin which sets an easy to use authentication platform, providing diverse functionalities such as authentication using passwords, phone numbers, popular identity providers

like Google, Facebook and Twitter, and more. In Smart Commute it is used as the interface that allows the user to authenticate, register and log in into the app.

Back end tools

Nodejs is an open source JavaScript runtime environment that is useful for both servers and desktop and Gulp is a tool kit for automating tasks in web development, and both tools are used to build the network app and execute it, as well as running it interactively through local host.

jQuery is a JavaScript library that simplifies the programming time and is used to save time and space in the coding of our Web app. Additionally, as FullCalendar is built on jQuery, it was mandatory to have this library.

Firebase Authentication SDK provides methods to create and manage users that use their email addresses and passwords to sign in and was used in the process to manage the user log in, log out, registration and validation.

Firebase Database is the cloud hosted Database used to store and interact with the user data from the Web app.

Testing

The testing process was black box focused. An end-to-end testing of the several modules was performed, and a more detailed white box unit testing of the main components is advised for future version testing.

As for the end-to-end testing, it consisted of the overall test of the main modules: the event addition, event modification, event deletion, roadmap, and settings.

The functions tested can be classified as follows:

1. Testing of functions that arrange and set the correct format of the events and settings information to be used in the processes of storing and rendering.
2. Testing of functions that check and validate the information introduced by the user throughout event popups.
3. Testing of functions that compute the event times and validate the fact that they don't overlap with each other.
4. Testing of functions that compute the routes between places stored in the events and validate the feasibility of the event addition.
5. Testing of Functions that manage the interaction of the user with the dialog boxes.

The results of the tests are shown in the annex of this document.

Conclusions

The Smart Commute application presented, developed a solution to simplify the processes of scheduling, calculating times and routes between user's appointments, helping to efficiently coordinate scheduled activities considering the travel time between events, through a friendly interface that removes unnecessary functionalities presented in commercial general-purpose mapping software designed to cover as many user personas as possible. Additionally, due to its web-base approach it provides the advantage of being used in various devices and in any operating system that has a web browser, that is, it has great portability and compatibility.

The web-app development process was based in the waterfall model, which allowed to develop in an orderly and simple way each of the stages of software development and meet the idea, objective and requirements specified. However, due to time constraints some of the stipulated requirements had to be limited and projected for future versions.

In addition, we can conclude the great utility provided by the systematic development of the software, both for the control, testing and organization of the activities during the work, as well as for an orderly documentation of the product and the stages developed.

Future work

- Manage better the asynchronous calls: sometimes the app fails to successfully render all the events and routes due to slow requests to the here maps server. A method in which the algorithm waits for all the data to be retrieved
- Integrate settings into the app engine for break and lunch times, and for the transport preferences
- Geolocation is currently done twice: for event travel time verification and for the roadmap engine. In a future release, only one geolocation request is necessary: in the calendar engine, if all the event creation requirements are met, the script can append the coordinate key to the event object and upload it to the database.
- Let the rendering of events and routes be day exclusive, not all in one, also controlled by the previous and next buttons on the side calendar.
- Add route duration details on popups.
- Implement the app into a hosting server.
- Try enabling offline functionality powered by hoodie JS

References

[i] https://medium.com/@bfrancis/the-legacy-of-firefox-os-c58ec32d94f0
[ii] https://mobiforge.com/news-comment/mobile-audience-growth-web-not-apps-comscore

Annex: Test Reports

Project Name: Smart Commute Report Test 1: Event creation	
Module Name:	Calendar Engine / event creation – script: app.js
Module description:	The event creation is comprised of several functions from the moment the user triggers the event submission (functions called in the same order): • $newEvent(start, end)$ • $validateEvents(userid, eventData, calEvent)$ • $isNotColliding(userid, eventData, calEvent)$ • $generateNodes(allEvNodes, currentTime)$ • $geocodeNext()$ or $geocodeDef()$ • $validateWithRoutingtime()$ • $newEventAdditionProcess(userid, eventData)$
Type of Test:	End to end test
Test Software:	Google Chrome Console
Test Description:	Tests the event creation from a user request
Test Date:	09 March 2019
Test Scenario:	• The user is logged in • The user clicks a day in the month • Fills the event popup form • Clicks submit event

Test case 1/5: submit event with null title						
Pre-Conditions:	• User must be logged in • User must click a day from the calendar month in month view from Calendar GUI					
Step #	**Step Details**	**Test Data**	**Expected Result**	**Actual Result**	**Status (Pass/Fail)**	**Notes**
1	Provide an event title	<empty>	Title can be entered	Expected	Pass	
2	Provide an event location	“Piazza Leonardo Da Vinci, 32”	Location can be entered	Expected	Pass	Location should match a result from the here maps addresses database

3	Provide start time	13:00	Start time can be entered	Expected	Pass	
4	Provide end time	15:00	End time can be entered	Expected	Pass	
5	User clicks event submit		Alert: please insert a title	Alert: please insert a title	Pass	
Post-Condition		App retains the popup open waiting for correct user input				
Test case 2/5 submit event with incorrect start and end times						
Pre-Conditions:		• User must be logged in • User must click a day from the calendar month in month view from Calendar GUI				
Step #	**Step Details**	**Test Data**	**Expected Result**	**Actual Result**	**Status (Pass/Fail)**	**Notes**
1	Provide event title	"Meeting with professor"	Title can be entered	Expected	Pass	
2	Provide an event location	"Piazza Leonardo Da Vinci, 32"	Location can be entered	Expected	Pass	Location should match a result from the here maps addresses database
3	Provide start time	14:00	Start time can be entered	Expected	Pass	
4	Provide end time	12:00	End time can be entered	Expected	Pass	
5	User clicks event submit		Alert: event times are invalid. Please correct	Alert: event times are invalid. Please correct	Pass	
Post-condition		App retains the popup open waiting for correct user input				
Test case 3/5 user submits event successfully						
Pre-Conditions:		• User must be logged in • User must click a day from the calendar month in month view from Calendar GUI				

Step #	Step Details	Test Data	Expected Result	Actual Result	Status (Pass/Fail)	Notes
1	Provide event title	“Meeting with professor”	Title can be entered	Expected	Pass	
2	Provide an event location	“Piazza Leonardo Da Vinci, 32”	Location can be entered	Expected	Pass	Location should match a result from the here maps addresses database
3	Provide start time	13:00	Start time can be entered	Expected	Pass	
4	Provide end time	14:30	End time can be entered	Expected	Pass	
5	User clicks event submit		Event is submitted	Event is submitted	Pass	Event is rendered onto the calendar and saved in Firebase DB
Post-condition		Popup closes and shows event on the calendar GUI				

Test case 4/5 user submits a second event which overlaps with the first event’s duration						
Pre-Conditions:		• User must be logged in • User has successfully created an event (test case 3) • User must click a day from the calendar month in month view from Calendar GUI				
Step #	**Step Details**	**Test Data**	**Expected Result**	**Actual Result**	**Status (Pass/Fail)**	**Notes**
1	Provide event title	“Shopping in C. Buenos Aires”	Title can be entered	Expected	Pass	
2	Provide an event location	“Corso Buenos Aires, Milano”	Location can be entered	Expected	Pass	Location should match a result from the here maps addresses database
3	Provide start time	14:00	Start time can be entered	Expected	Pass	
4	Provide end time	16:00	End time can be entered	Expected	Pass	
5	User clicks event submit		Alert: one or more events overlapping.	Alert: one or more events	Pass	Event time is compared with all event times and event created

			Try changing the time.	overlapping. Try changing the time.		(test case 3) overlaps with this one.
Post-condition		App retains the popup open waiting for adequate time input				

Test case 5/5 user submits a second event whose travel time is larger than the time available						
Pre-Conditions:		• User must be logged in • User has successfully created an event (test case 3) • User must click a day from the calendar month in month view from Calendar GUI				
Step #	**Step Details**	**Test Data**	**Expected Result**	**Actual Result**	**Status (Pass/Fail)**	**Notes**
1	Provide event title	"Friends reunion in Eindhoven"	Title can be entered	Expected	Pass	
2	Provide an event location	"Eindhoven, Netherlands"	Location can be entered	Expected	Pass	Location should match a result from the here maps addresses database
3	Provide start time	15:30	Start time can be entered	Expected	Pass	
4	Provide end time	19:00	End time can be entered	Expected	Pass	
5	User clicks event submit		Alert: you're either short on time to get there or you'd be unable to get to the following appointment!	Alert: you're either short on time to get there or you'd be unable to get to the following appointment!	Pass	Event travel time is compared with time gap between the end of first event (test case 3) and its start time and results in a lack of time to travel.
Post-condition		App retains the popup open waiting for adequate input				

Project Name: Smart Commute Report Test 2: Event modification	
Module Name:	Calendar Engine / event modification – script: app.js
Module description:	The event modification is comprised of several functions from the moment the user triggers the event update (functions called in the same order): • $editEvent(calEvent)$ • $validateEvents(userid, eventData, calEvent)$ • $isNotColliding(userid, eventData, calEvent)$ • $generateNodes(allEvNodes, currentTime)$ • $geocodeNext()\ or\ geocodeDef()$ • $validateWithRoutingtime()$ • $eventModificationProcess(userid, calEvent, eventData)$
Type of Test:	End to end test
Test Software:	Google Chrome Console
Test Description:	Tests the event modification from a user request
Test Date:	09 March 2019
Test Scenario:	• The user is logged in • The user clicks an existing event previously created (during current session or previous sessions) • Fills the event popup form • Clicks update event

Test case 1/5: update event with null title						
Pre-Conditions:	• User must be logged in • User must click a day from the calendar month in month view from Calendar GUI					
Step #	**Step Details**	**Test Data**	**Expected Result**	**Actual Result**	**Status (Pass/Fail)**	**Notes**
1	Provide an event title	<empty>	Title can be entered	Expected	Pass	
2	Provide an event location	“Piazza Leonardo Da Vinci, 32”	Location can be entered	Expected	Pass	Location should match a result from the here maps addresses database

3	Provide start time	13:00	Start time can be entered	Expected	Pass	
4	Provide end time	15:00	End time can be entered	Expected	Pass	
5	User clicks update event		Alert: please insert a title	Alert: please insert a title	Pass	
Post-Condition		App retains the popup open waiting for correct user input				
Test case 2/5: update event with incorrect start and end times						
Pre-Conditions:		• User must be logged in • User must click a day from the calendar month in month view from Calendar GUI				
Step #	**Step Details**	**Test Data**	**Expected Result**	**Actual Result**	**Status (Pass/Fail)**	**Notes**
1	Provide event title	"Meeting with professor"	Title can be entered	Expected	Pass	
2	Provide an event location	"Piazza Leonardo Da Vinci, 32"	Location can be entered	Expected	Pass	Location should match a result from the here maps addresses database
3	Provide start time	14:00	Start time can be entered	Expected	Pass	
4	Provide end time	12:00	End time can be entered	Expected	Pass	
5	User clicks update event		Alert: event times are invalid. Please correct	Alert: event times are invalid. Please correct	Pass	
Post-condition		App retains the popup open waiting for correct user input				
Test case 3/5 user updates event successfully						
Pre-Conditions:		• User must be logged in • User must click a day from the calendar month in month view from Calendar GUI				

Step #	Step Details	Test Data	Expected Result	Actual Result	Status (Pass/Fail)	Notes
1	Provide event title	“Meeting in B11”	Title can be entered	Expected	Pass	
2	Provide an event location	“Piazza Leonardo Da Vinci, 32”	Location can be entered	Expected	Pass	Location should match a result from the here maps addresses database
3	Provide start time	15:00	Start time can be entered	Expected	Pass	
4	Provide end time	16:30	End time can be entered	Expected	Pass	
5	User clicks update event		Event is submitted	Event is submitted	Pass	Event is rendered onto the calendar and saved in Firebase DB
Post-condition		Popup closes and shows event on the calendar GUI				

Test case 4/5 user submits a second event which overlaps with the first event’s duration						
Pre-Conditions:		• User must be logged in • User has successfully created an event (test case 3) • User must click a day from the calendar month in month view from Calendar GUI				
Step #	**Step Details**	**Test Data**	**Expected Result**	**Actual Result**	**Status (Pass/Fail)**	**Notes**
1	Provide event title	“Shopping in C. Buenos Aires”	Title can be entered	Expected	Pass	
2	Provide an event location	“Corso Buenos Aires, Milano”	Location can be entered	Expected	Pass	Location should match a result from the here maps addresses database
3	Provide start time	16:00	Start time can be entered	Expected	Pass	
4	Provide end time	19:00	End time can be entered	Expected	Pass	
5	User clicks update event		Alert: one or more events overlapping.	Alert: one or more events	Pass	Event time is compared with all event times and event created

			Try changing the time.	overlapping. Try changing the time.		(test case 3) overlaps with this one.
Post-condition		App retains the popup open waiting for adequate time input				

Test case 5/5 user submits a second event whose travel time is larger than the time available						
Pre-Conditions:		• User must be logged in • User has successfully created an event (test case 3) • User must click a day from the calendar month in month view from Calendar GUI				
Step #	**Step Details**	**Test Data**	**Expected Result**	**Actual Result**	**Status (Pass/Fail)**	**Notes**
1	Provide event title	"Friends reunion in Eindhoven"	Title can be entered	Expected	Pass	
2	Provide an event location	"Eindhoven, Netherlands"	Location can be entered	Expected	Pass	Location should match a result from the here maps addresses database
3	Provide start time	17:00	Start time can be entered	Expected	Pass	
4	Provide end time	19:00	End time can be entered	Expected	Pass	
5	User clicks update event		Alert: you're either short on time to get there or you'd be unable to get to the following appointment!	Alert: you're either short on time to get there or you'd be unable to get to the following appointment!	Pass	Event travel time is compared with time gap between the end of first event (test case 3) and its start time and results in a lack of time to travel.
Post-condition		App retains the popup open waiting for adequate input				

Project Name: Smart Commute Report Test 3: Event deletion	
Module Name:	Calendar Engine / event deletion – script: app.js
Module description:	The event deletion is comprised of several functions from the moment the user triggers delete event (functions called in the same order): • $editEvent(calEvent)$ • $\$('delete').on('click', function());$ • $eventRenderer();$
Type of Test:	End to end test
Test Software:	Google Chrome Console
Test Description:	Tests the event modification from a user request
Test Date:	09 March 2019

Test Scenario:	• The user is logged in • The user clicks an existing event previously created (during current session or previous sessions) • Clicks delete event

Test case 1/1: delete any preexisting event						
Pre-Conditions:		• User must be logged in • User must click a day from the calendar month in month view from Calendar GUI				
Step #	**Step Details**	**Test Data**	**Expected Result**	**Actual Result**	**Status (Pass/Fail)**	**Notes**
1	Click “delete button”		Event is deleted	Event is deleted	Pass	Event disappears from the GUI and is deleted from the Database
Post-Condition		App closes the popup and goes back to the calendar GUI				

Project Name: Smart Commute Report Test 4: Roadmap GUI						
Module Name:	Roadmap Engine – script: maps.js					
Module description:	The event creation is comprised of several functions from the moment the user triggers the event submission (functions called in the same order): • $checkAuth()$ • $\$('calendarInRoadmap').fullCalendar(\{\});$ • $getLocation()$ • $mapWMyLoc(myPosX, myPosY)$ • $gotAllUpcEvtCoords(map, mapEvents, defaultLayers, geocoder, mycoords, userEvents);$ • $gotAllUpcEvtRoutes(myRouter, mycoords, userEvents);$ • $renderingEach(routes, j, map, ui, userEvents);$					
Type of Test:	End to end test					
Test Software:	Google Chrome Console					
Test Description:	Tests the event and route rendering on the roadmap upon user access					
Test Date:	09 March 2019					
Test Scenario:	• The user is logged in • The user clicks the roadmap button • The events and routes are then sequentially rendered onto the map and the right-side daily calendar					
Test case 1/1: User enters the roadmap						
Pre-Conditions:	• User must be logged in • User must click the button roadmap from the main page (calendar GUI)					
Step #	**Step Details**	**Test Data**	**Expected Result**	**Actual Result**	**Status (Pass/Fail)**	**Notes**
1	Click roadmap in main GUI	<empty>	Switch to Roadmap GUI	Expected	Pass	
Post-Condition	Roadmap GUI is loaded with all events and routes rendered respectively.					

Project Name: Smart Commute Report Test 5: User changes settings	
Module Name:	Settings – script: settings.js
Module description:	The event creation is comprised of several functions from the moment the user triggers the event submission (functions called in the same order): • $getSettings()$ • $checkUser()$ • $settingsData(data);$ • $loadDatasOnFrontEnd();$ • $\$('saveNexit').on('click', function());$ • $loadDatasOnFrontEnd();$
Type of Test:	End to end test
Test Software:	Google Chrome Console
Test Description:	Tests the change of settings by the user
Test Date:	09 March 2019
Test Scenario:	• The user is logged in • The user clicks the settings button • The user modifies the settings at will • The user clicks “save & exit”

Test case 1/1 User modifies its settings

Pre-Conditions:	• User must be logged in • User must click the button settings from the main page (calendar GUI)

Step #	Step Details	Test Data	Expected Result	Actual Result	Status (Pass/Fail)	Notes
1	Click settings on main GUI	<empty>	Switch to Settings GUI	Switch to Settings GUI	Pass	
2	User selects preferred transport	“metro”	Dropdown can be accessed	Expected	Pass	

3	User checks available transports	[true, true, true, true, true, true] (selects all transports as available)	Round checks can be used	Expected	Pass	Each element of the array in test data indicates whether the transport is checked or not, for the 6 types of transport. Available entries: true, false.
4	User checks lunch time	13:00 – 14:00	Lunch time can be entered	Expected	Pass	
5	User checks break time	17:15 – 17:30	Break time can be entered	Expected	Pass	
6	User clicks “save & exit”		App redirects back to calendar GUI	App redirects back to calendar GUI	Pass	Lunch and break times are not yet implemented and haven’t been saved in the Database.
Post-Condition		Settings preloaded respectively onto the GUI and saved in the Database				