Cover, title

# 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 is divided into the following sections:

Section 1 describes the web-app requirements and objectives, user personas (motivations and justifications), an overall structure of the app and a general description of the framework under which it was developed

Section 2 presents the app design, the system architecture on front and back end, integrated by entity relationships diagrams such as use case diagrams, flowchart diagrams, sequence diagrams and class diagrams.

Section 3 shows the software implementation, testing and software execution.

Section 4 addresses the description of the main steps followed along the web app development, highlighting the main libraries, API’s, tools and modules implemented.

Index

[Introduction 2](#_Toc2197689)

[App Overview 4](#_Toc2197690)

[The idea 5](#_Toc2197691)

[Who would want to use Smart Commute? 6](#_Toc2197692)

[Why would they want to use it? 6](#_Toc2197693)

[What are they looking for? 6](#_Toc2197694)

[How does Smart Commute provide the utilities that solve their problems? 6](#_Toc2197695)

[Objective 7](#_Toc2197696)

[Requirements 7](#_Toc2197697)

[Use case diagram 7](#_Toc2197698)

[System architecture 8](#_Toc2197699)

[Section 2: App design 10](#_Toc2197700)

[Entity relationship diagrams 10](#_Toc2197701)

[High Level Class Diagram 10](#_Toc2197702)

[Front-end Class Diagram 10](#_Toc2197703)

[Back-end Class Diagram 12](#_Toc2197704)

[Sequence diagrams 16](#_Toc2197705)

[Log in to Web-App and load user data to the environment 16](#_Toc2197706)

[Create an event 17](#_Toc2197707)

[Modify and Delete Event 18](#_Toc2197708)

[Select Event on Map 19](#_Toc2197709)

[Web interface, database 20](#_Toc2197710)

[Section 3 20](#_Toc2197711)

[Implementation 20](#_Toc2197712)

[Used tools 22](#_Toc2197713)

[Testing 22](#_Toc2197714)

[Conclusion and future work 22](#_Toc2197715)

# App Overview

Why meaningful? Who is the target? What is the app trying to solve? How is it different from the competitors? Must fix syntax on citations.

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. (<https://medium.com/@bfrancis/the-legacy-of-firefox-os-c58ec32d94f0> )

This argument opens for an encouragement to focus on the web, as it has become more powerful over the years (<https://mobiforge.com/news-comment/mobile-audience-growth-web-not-apps-comscore> ) 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 (figure xxx) that shows the difference in approach towards the user needs:

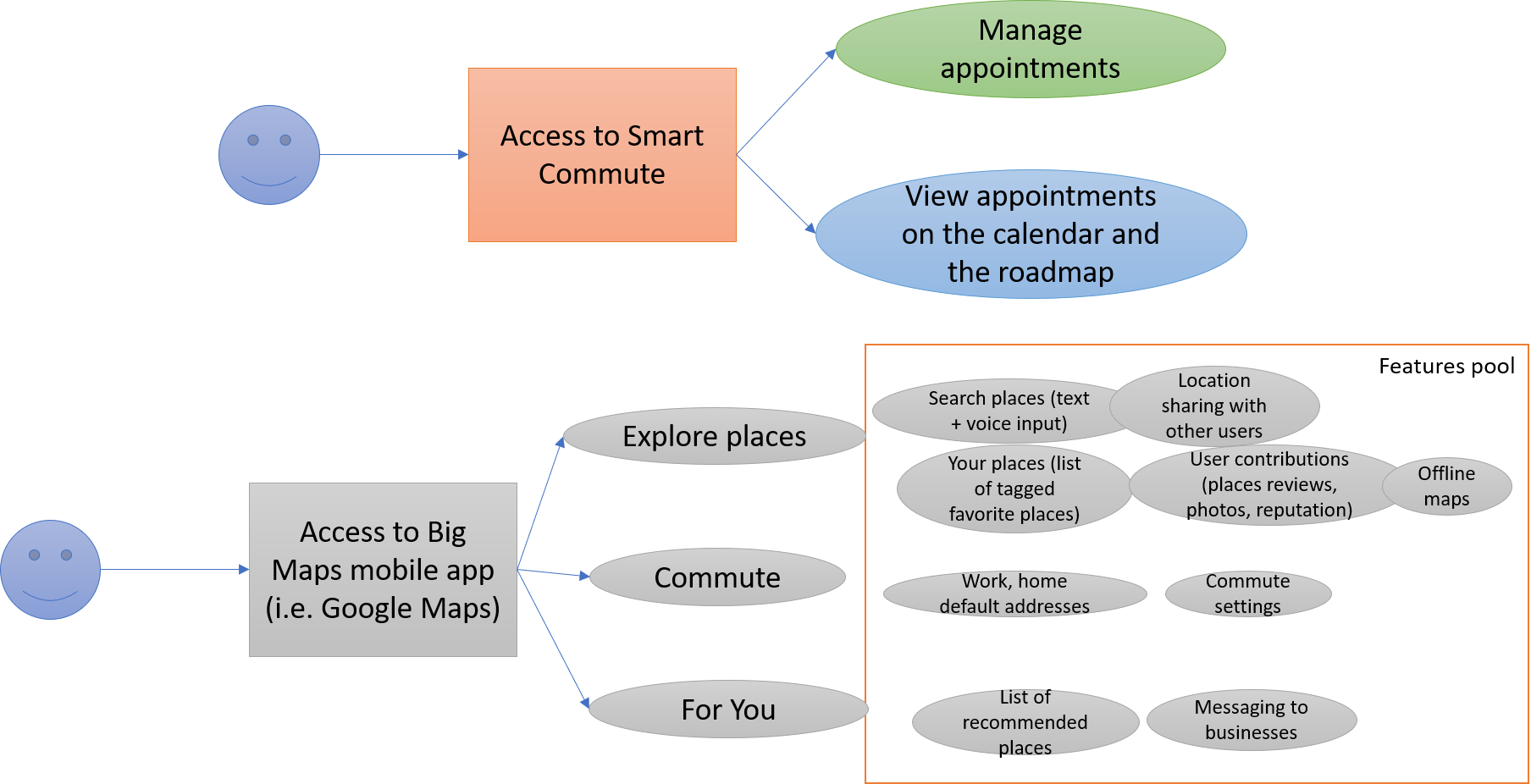


Figure XXX. Smart Commute differentiation with other large-scale mobile apps (i.e. Google Maps)

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 following figure XXX, 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.

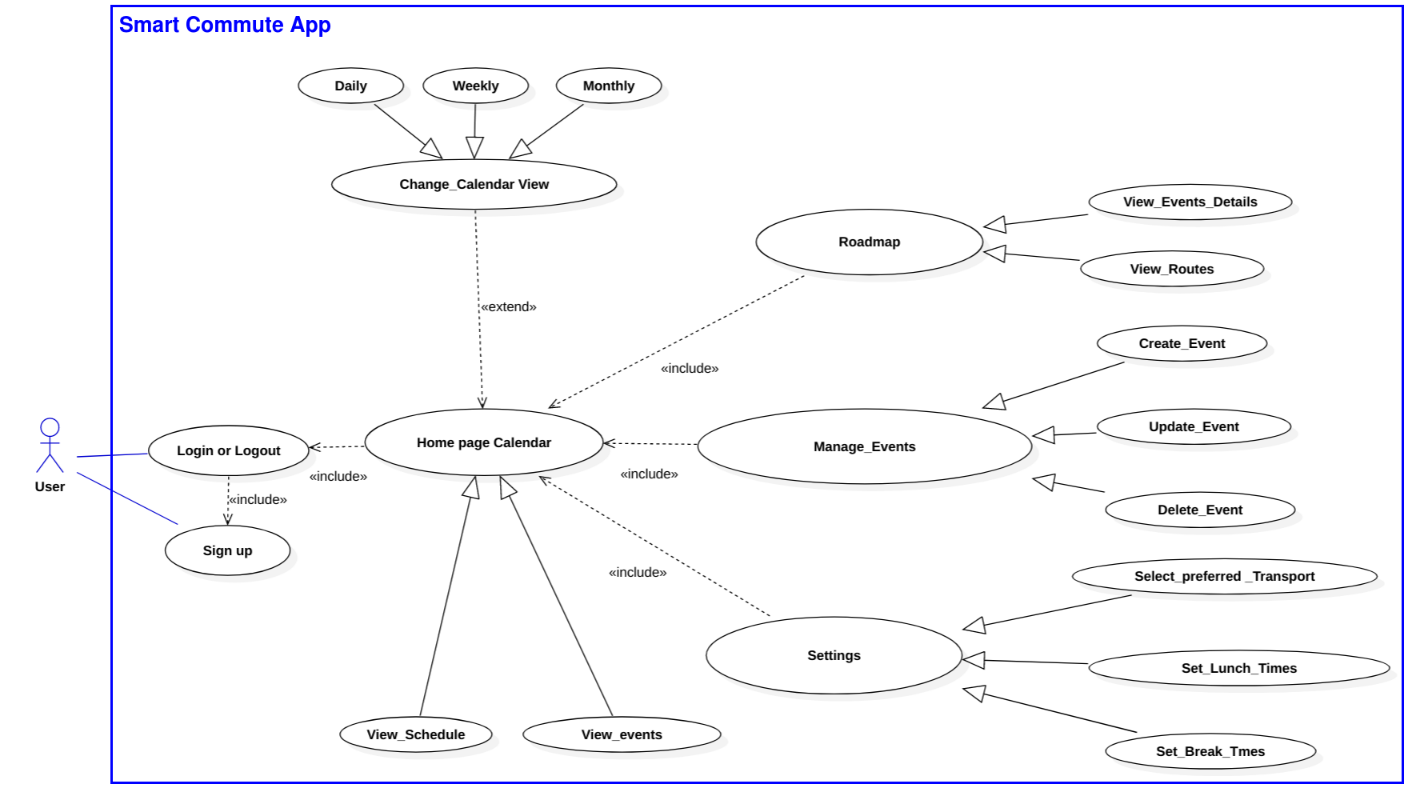


Figure XXX. Use Case Diagram

**Authentication System**

Allow access per user to its own private environment through login authentication, and new users to register into the system via creation of credentials with email and password. The application’s environment must be multi-user, which means that many users can access their own app’s saved data at the same time and each user data is stored separately in the database. Furthermore, it must allow the user to recover the password.

**Calendar interface**

The interface must contain a calendar that allows the user to manage its activities through the creation of events, and including the view options per month, week, and day. For each event, the interface allows the user to specify the following fields: title, start and end times, location and description. Additionally, the app automatically validates the information proportioned by the user to prevent event overlapping. The user must also be able to edit existent events, create multiday events, drag and drop events within each view, and delete events.

The calendar interface is automatically updated every time the user enters the app, and preloads and renders existing events and routines, if any, and any time an event is created updated or deleted, the application will update the user’s data and push it to the database. This way, the user’s data will be saved for any changes made.

**Smart Commuting**

The app must generate the most efficient travel between user’s appointments, by taking the information provided by the user (event times and addresses), considering the available means of transport to compute the travel time between appointments and the time gap available between events, compare both durations and ultimately avoid for time overlapping by warning to the user time inconsistency upon creation or time update of events. The smart commuting algorithm must be performed per day, and the commuting time will be constrained to the lunch and break times set by the user, and whether weekend days are considered for the user’s routine.

Smart commuting must be dynamically updated anytime the user modifies or creates events, to prevent the addition of inconsistencies. Through a map interface, the user’s event must be represented as well as the routes between them for a graphical representation of the events for a selected day.

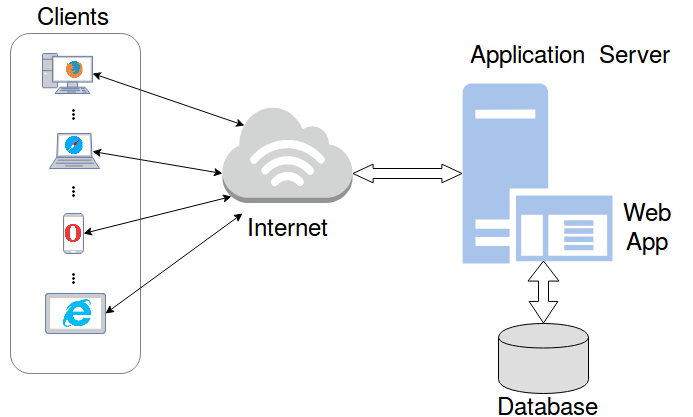
**User customizable**

The web app must allow the user to specify the preferred travel means, such as car, bike, walk, tram, metro or bus, to be weekday sensitive, to configure the workdays of the week, predefined break times and save the user preferences to the app as a configuration to be applied to future sessions.

let the user configure its workdays, predefined breaks and alert times.

## System architecture

The overall system architecture is illustrated in the fig. xxx, 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.



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

# Section 2: App design

## Entity relationship diagrams

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

### High Level Class Diagram

### Front-end Class Diagram

Firstly, the diagram of figure XXX 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.

**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 as well as clicking existing events with the operation 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 and . 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 or 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 . 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 that represents an array of Booleans for the allowed transport means to be used for the routing calculations, 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 , 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.

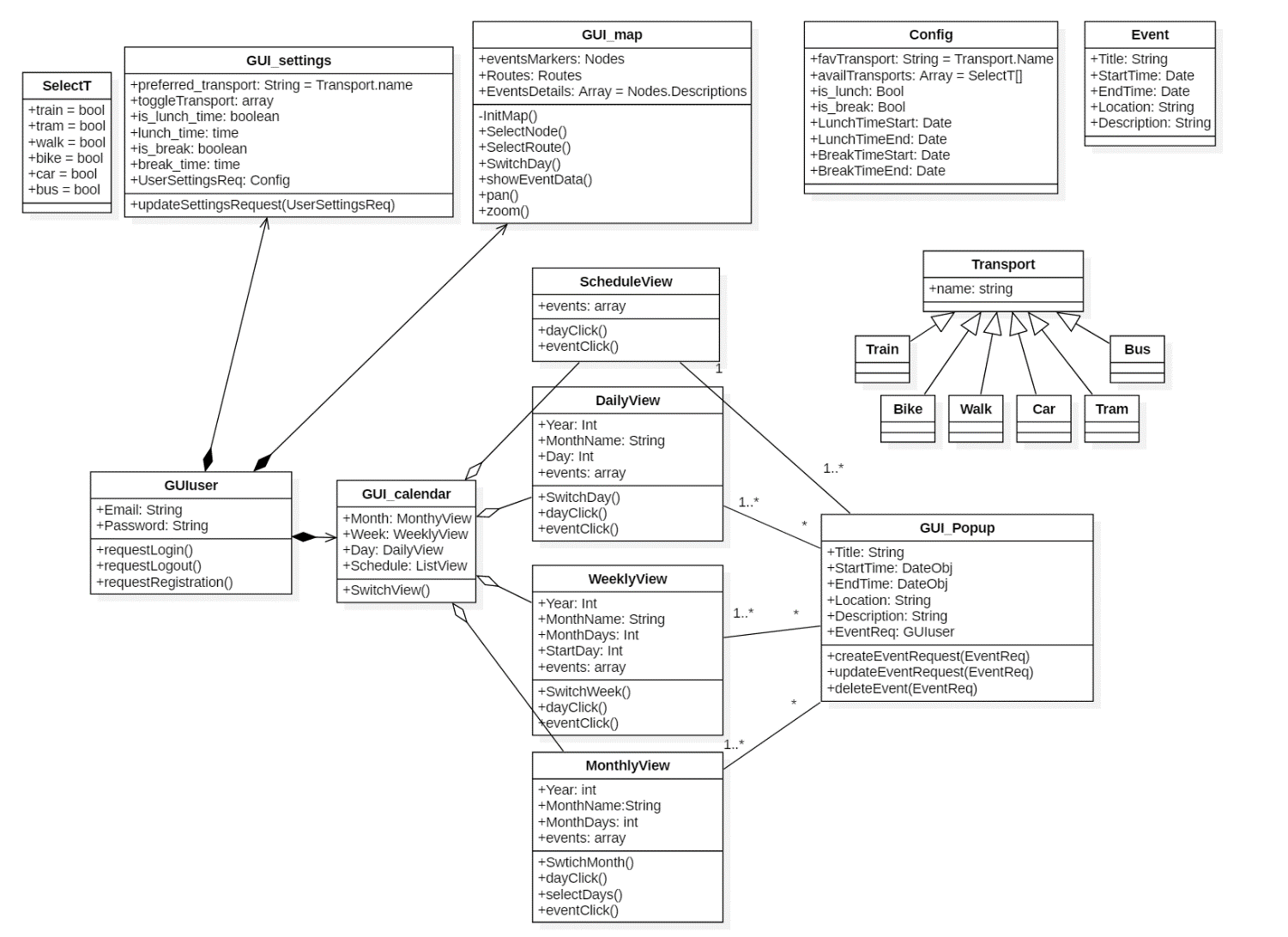


Fig. XXX. Front-End Class Diagram

### 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 , (both depicting an association with the programming interfaces of these external sources), 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.

**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 , 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 Firebase auth additionally verifies the user is logged in with its method , 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 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 (to add new events), for modification or deletion of events, and for a change in the user settings. All these operations are executed via Firebase methods, further explained in this documentation.

The following figure (XXX) 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 XXX. 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 to or register via to 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 which asks for these values from Firebase. Lastly, the operation redirects upon successful authentication to the user’s app home page.

**Settings Engine Class**

Essentially listens for from the GUI and sends the request with the operation whenever the user updates its configuration. At any app initialization, settings engine requests the config file with , allowing the Database to provide the settings data ( attribute) to be used by the different engine modules. The settings data is then rendered onto the interface with so they are shown anytime the user enters settings. Finally, the operations and 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 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 )and comparing each of the user events’ times and returning a true or false value. Validation is executed once the user requests to create or edit an event from the GUI, passed with the attribute . Also includes the operation , rendering the data once validated by the calendar engine and the maps engine. Lastly, whenever events are fully validated, the operation 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 with Address as the input parameter, which corresponds to the events location string. This operation resorts to the maps API to match the user’s address input and returns a geolocation coordinate value pair (latitude, longitude).

By using the values, the operation rearranges all the information from each event by appending each category of the events into different arrays, this makes the software easier to read and easier for the inserting of parameters on the operations and , which request to the maps API the transport means information and routing information to generate the objects (per day, equal to the number of events in that day).

These operations are succeeded by , executed after the transport constraints are received from the settings interface. This operation consists of verifying that the duration of the route is less than the duration between the events that define that route. If it is the case, the new event poses no restrictions to the roadmap so it can be added, therefore the operation is executed, and the routes and events are rendered to the map’s user interface with the operations and respectively.

**Other Classes**

Other classes were added for scheme clarification. The Event and Config classes were copied from the front-end diagram as they are used in certain main classes attributes. Additionally, and were added elements to depict the use of these external sources by the engine. 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. Lastly, organizes the events as arrays and inherits the settings data as a JSON.

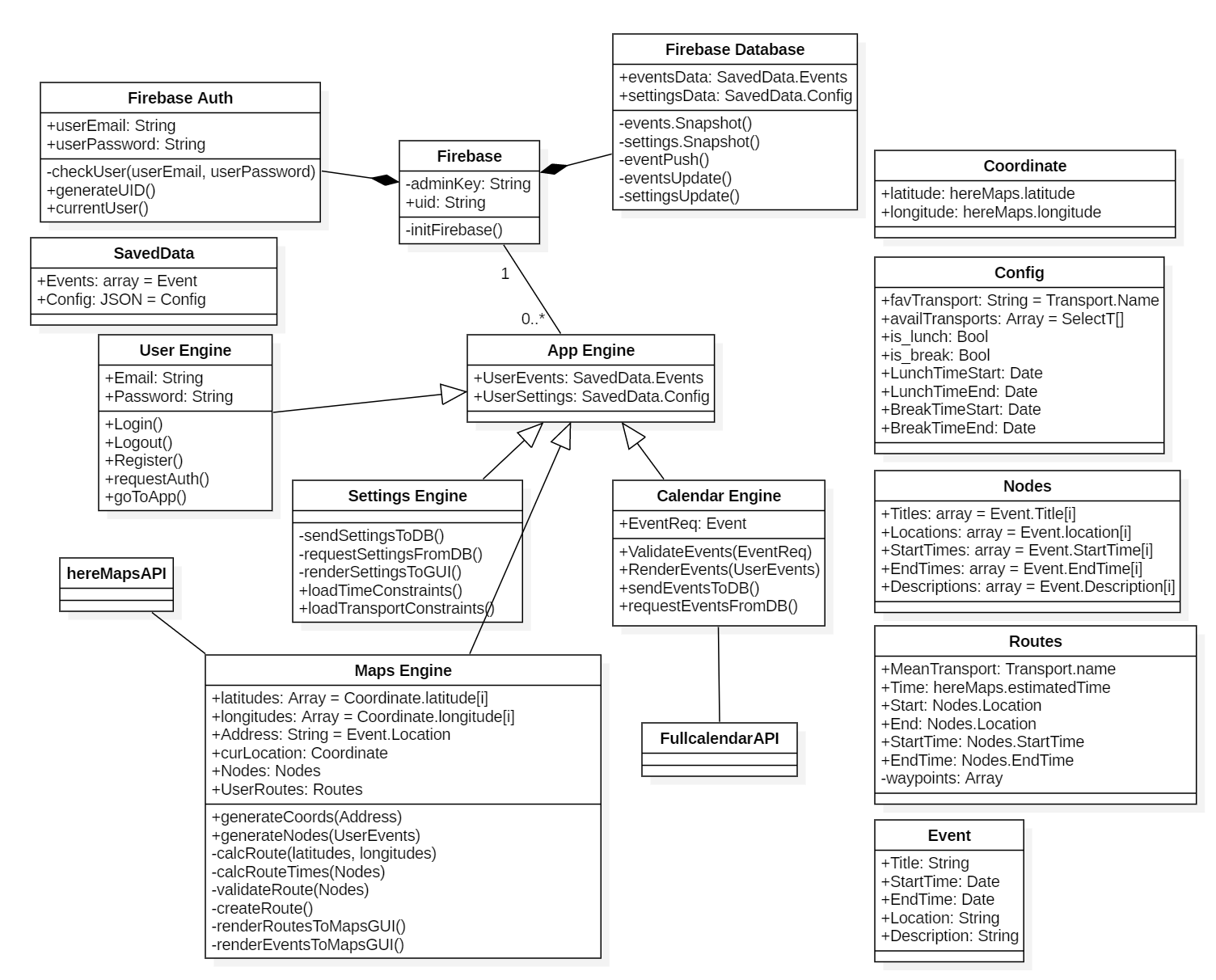


Fig. XXX. Back-End Class diagram

## Sequence diagrams

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

The diagram depicted in the figure xxx 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. 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.

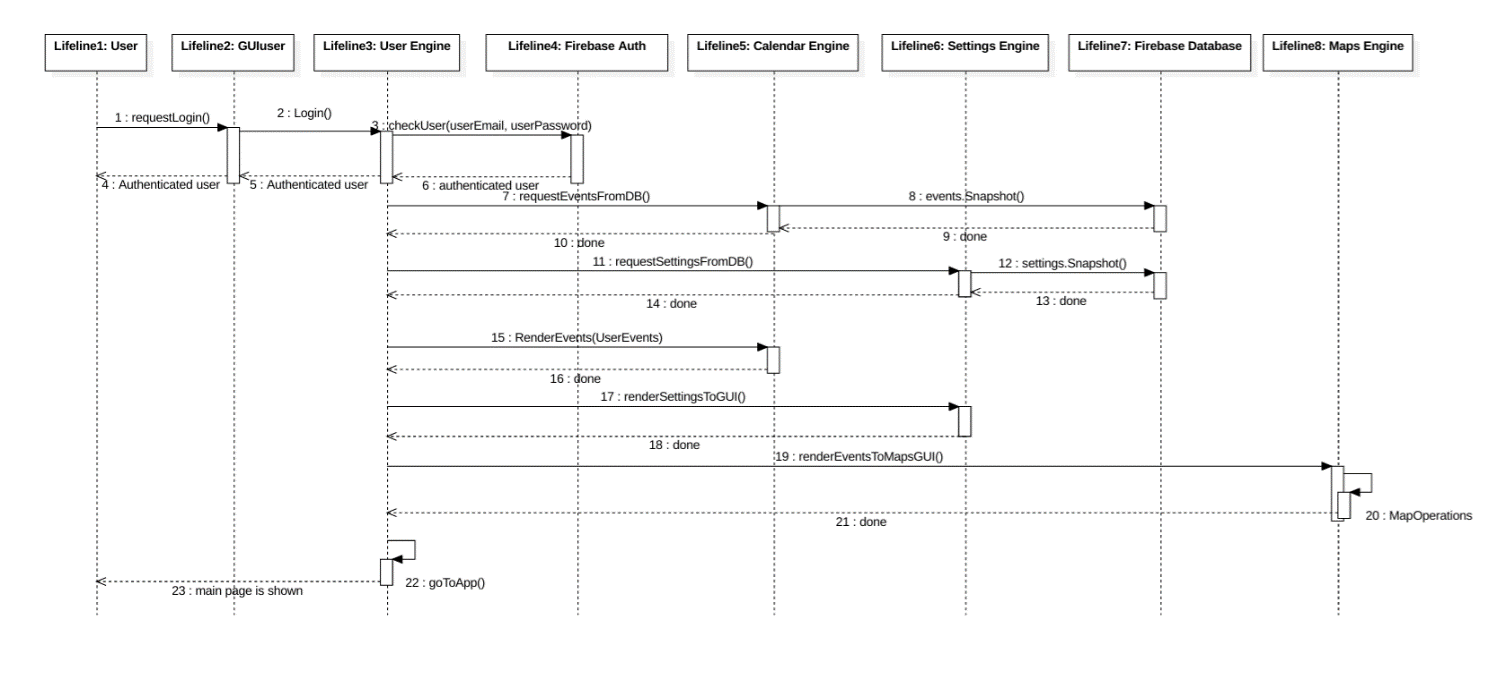


Fig. XXX. Log into the app sequence diagram

### Create an event

The Figure XXX 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.

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 requests the data of time constraints (e.g. lunch and break times) to the Settings Engine, and this provides to the requester the data of user configuration.
5. The Calendar Engine then asks to Map Engine to generate the coordinates of the address.
6. Map Engine then performs the following tasks: generates the nodes of the user’s events, requests the transport constraints specified by the user to Settings Engine, and with the time and transport constrains obtained it calculates the geographic coordinates of events and times of routes, then creates and validates the routes by making sure these routes are feasible, and at the end of these tasks, replies to Calendar engine that the event is validated.
7. The Calendar Engine replies to the GUI Popup that requested to validate the event that the event has been validated.
8. After the confirmation of the validation the GUI Popup requests to the Calendar Engine to send the event to the Database.
9. The Calendar Engine requests to push the event to the database to be stored and then it confirms to the requester that the action was completed successfully.
10. Subsequently, the Calendar Engine replies to the GUI Popup that requested to save the event, that the action has been completed.
11. The GUI Popup asks to the Calendar Engine to render the event.
12. The Calendar Engine renders the event onto the Calendar interface and finally the event is created.

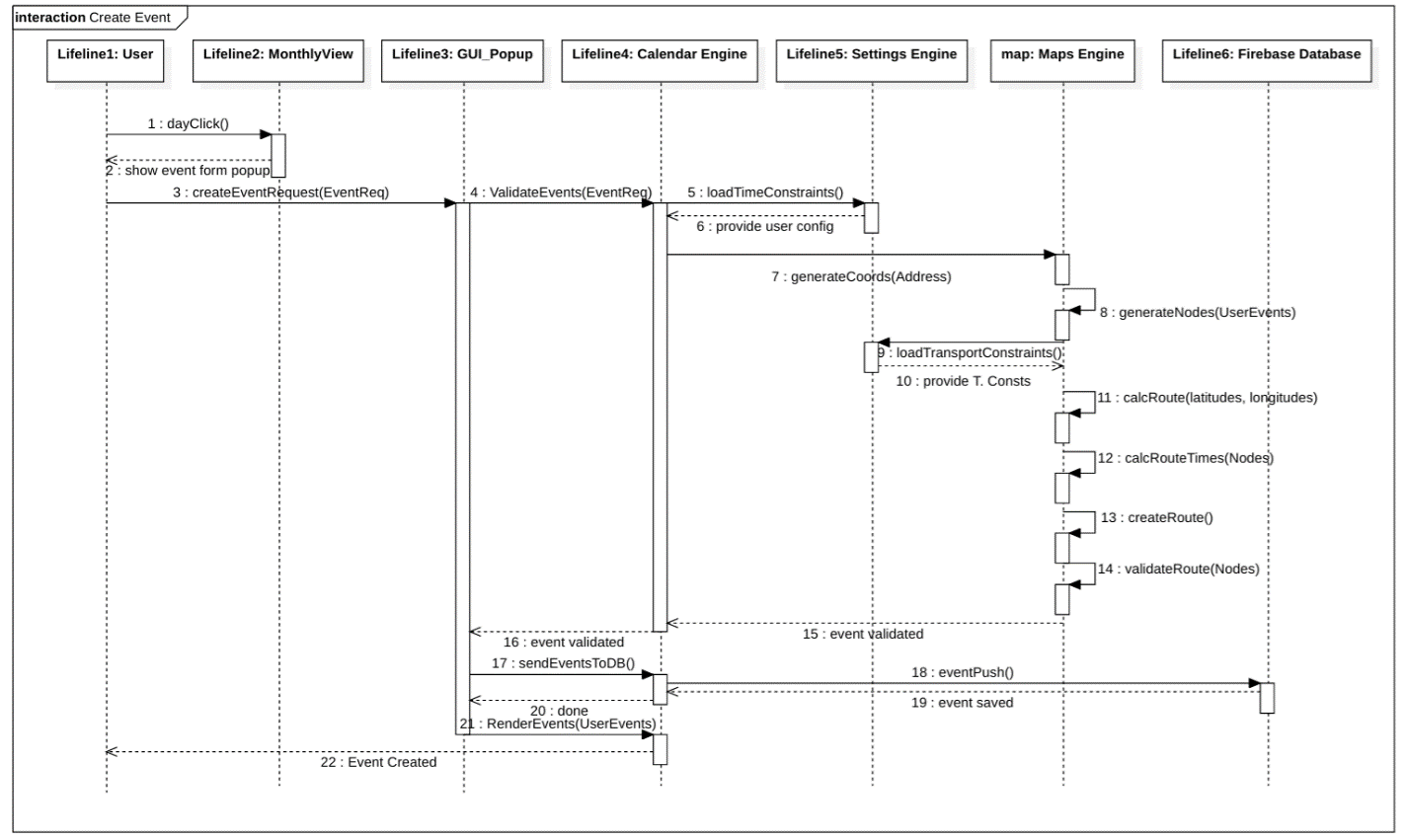


Fig. XXX. Create an event sequence diagram

### Modify and Delete Event

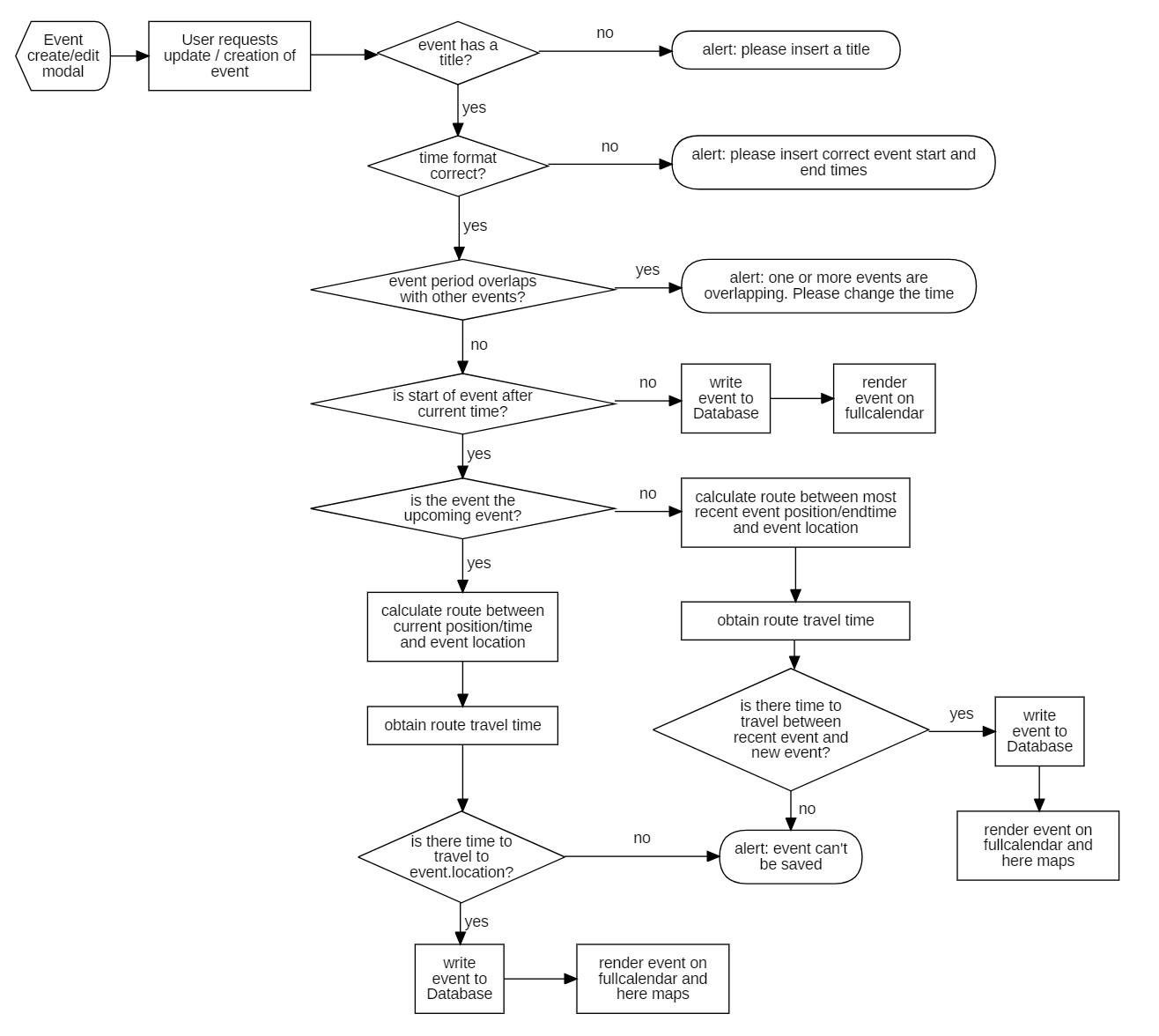
The scenarios of creating and modifying an event can be considered as variants of the previous sequence diagram since modifying an event is considered as an overwriting of the data and leads to the same validation processes performed in the scenario create event. In particular, the scenario to delete 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.

#### 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 workaround everything behind the interface and output the most convenience 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:



### Select Event on Map

The Figure XXX 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.

1. The user clicks on the option “Roadmap” located in home, triggering operation from the calendar interface.
2. Then the interface requests the map interface to initialize the map via the operation
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 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 operation.
7. The map interface triggers a popup with the specified event details.
8. Lastly, it is shown to the user in the interface.

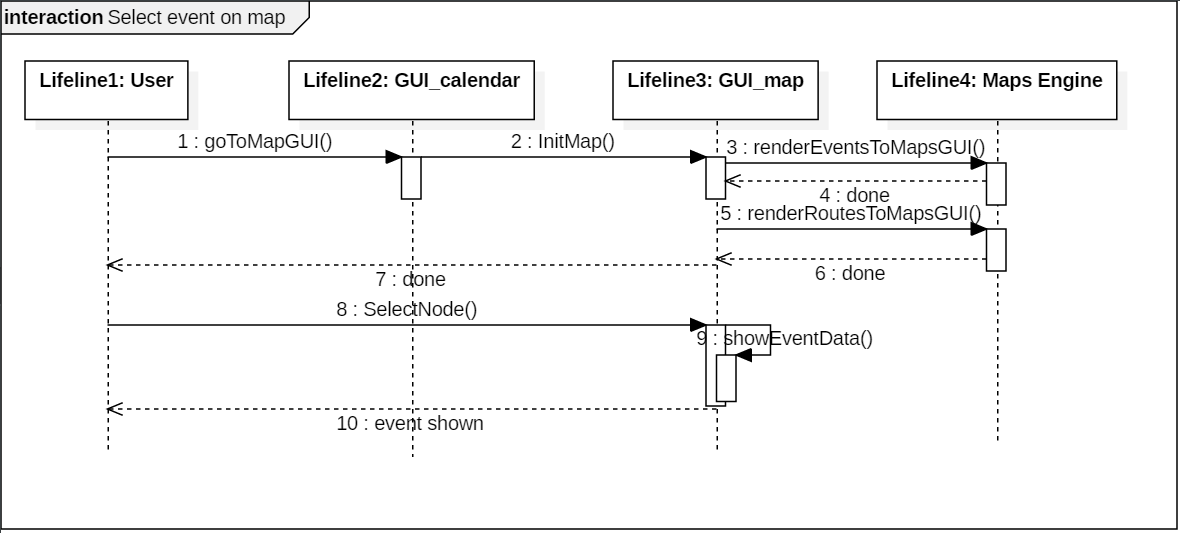


Fig. XXX. Select current day event on Map

## Web interface, database

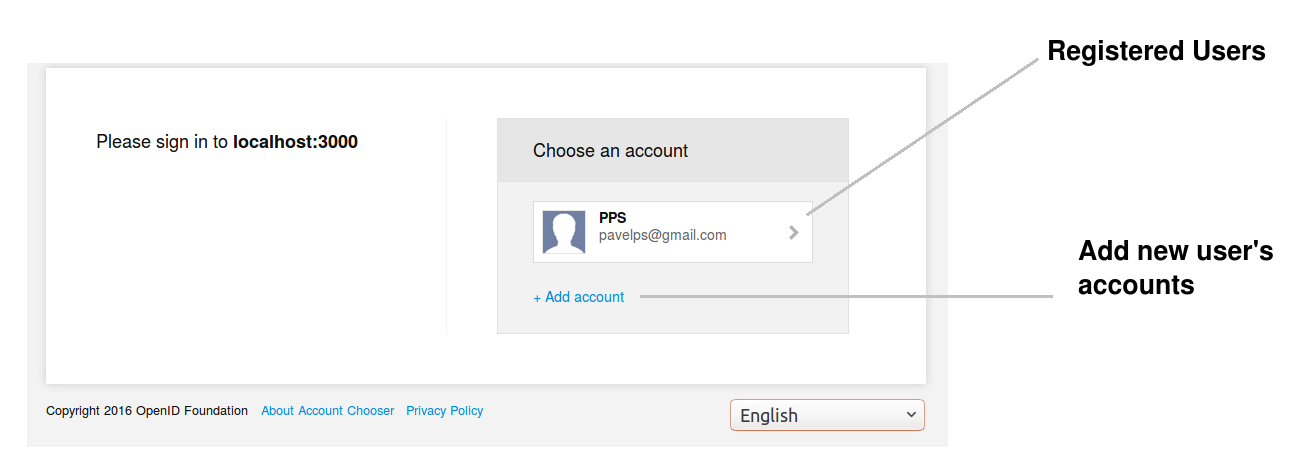
# Section 3

## Implementation

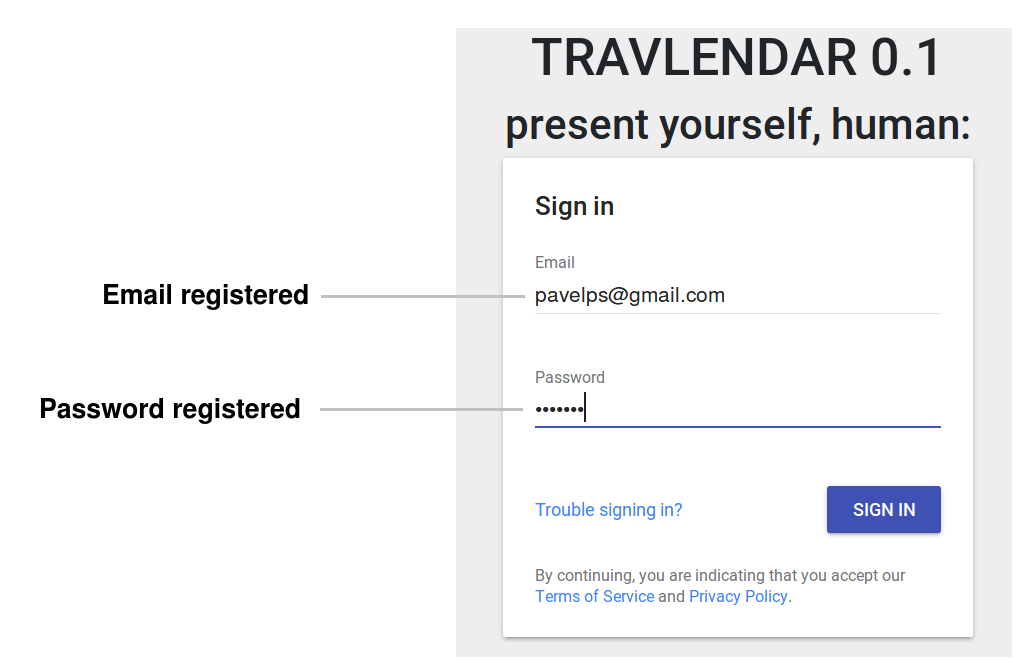
**Graphical User interfaces.**

**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 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.

****

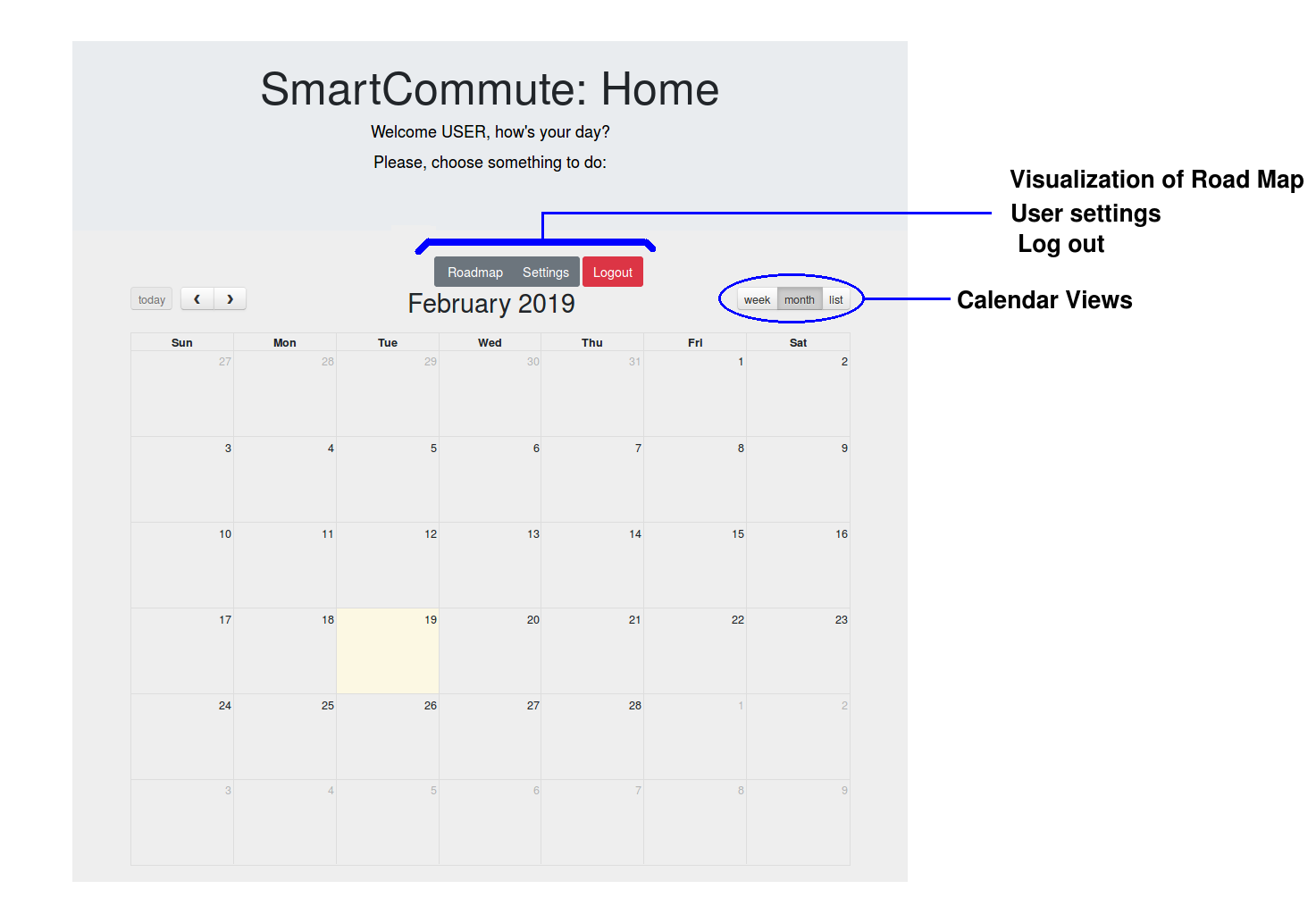
In the figure xxx, is illustrated the dialog form to submit the email and password for registered user or to add a new account and the button Sign in, which permit to access to the main page and load the configuration and events data of the user in the web app.

****

**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, permit 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 xxx.



**Event register**

This interface is a Popup Dialog form where the user

### Used tools

#### Front end tools

Bootstrap

FullCalendar

Here Maps

#### Back end tools

Firebase Auth

Firebase Database

Here Maps

Gulp

## Testing

# Conclusion and future work