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](#_Toc1065402)

[Section 1 4](#_Toc1065403)

[Objective 4](#_Toc1065404)

[Requirements 4](#_Toc1065405)

[App overall description 5](#_Toc1065406)

[User Personas 5](#_Toc1065407)

[Who would want to use Smart Commute? 5](#_Toc1065408)

[Why would they want to use it? 5](#_Toc1065409)

[What are they looking for? 5](#_Toc1065410)

[How does Smart Commute provide the utilities that solve their problems? 5](#_Toc1065411)

[App overall structure 5](#_Toc1065412)

[Description of framework 5](#_Toc1065413)

[Section 2 6](#_Toc1065414)

[Design 6](#_Toc1065415)

[System architecture 6](#_Toc1065416)

[Entity relationship diagram 6](#_Toc1065417)

[Web interface, database 6](#_Toc1065418)

[Section 3 6](#_Toc1065419)

[Implementation 6](#_Toc1065420)

[Section 4 6](#_Toc1065421)

[Used tools 6](#_Toc1065422)

[Front end tools 6](#_Toc1065423)

[Back end tools 6](#_Toc1065424)

[Conclusion and future work 6](#_Toc1065425)

# Section 1

Nowadays, work routines in cities are getting progressively more hectic, as population and development increase 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.

Several 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, routing and even multiple waypoint route coordination. These apps such as Bing Maps, Google Maps, City Mapper, Moovit or Apple Maps are quite complete, but they 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, and while mobile apps might seem the most reliable tool for the people on the go, relying on a browser tab in Safari or Google Chrome in a smartphone wouldn’t be a drawback as long as there is high compatibility for mobile web browsing.

It is 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. This assures usage in the office, 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.

## 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

**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.

## App description

## User Personas

### 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.

This is not a mobile based application, nor it is intended to be (at least not yet). 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.

### 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. 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.

## Overall structure

## Description of framework

# Section 2

## Design

## System architecture

### Use case diagrams

### 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.

#### 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 (which contains settings, map and calendar), Events, Map and Firebase DB, explained in the following paragraphs.

**GUI user class**

It’s the main class and it manages the user information to access the application. It allows the user to login, sign up, and its role is to receive any user credentials and upon authentication it allows access and loading of user information.

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 the user to toggle views for day, week or month and it allows to create single day or multi day events, as well as clicking existing events.

Secondly, the map GUI initializes a map from the map’s API, loads events and renders them as markers on the map, loads the generated routes and renders them as polylines, lets the user pan or zoom the map and click on the routes or maps to show their details. This class includes the operations of loading the events and routes from the app engine (on back-end class diagram) and show event and route data upon user click to rendered elements.

Lastly, the settings GUI lets the user through input field 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 to the Database as a JSON file.

**Event class**

Whenever a new event is to be created or an existing one is being clicked, a modal appears which corresponds to the class “Event”, and the user can interact with it by adding information which are the event’s attributes. The class event allows to add, delete, update event. Additionally, it sends an event update request to the user’s database information and renders the new/existing event or de-renders a deleted event from the calendar interface.

**Map class**

Map class is present for the purpose of linking the event information, reorganizing the event’s data fields to generate a node object (with the attributes from the Nodes class and through the operation ), communicate with the maps API to obtain latitude and longitude information from the event location address string through the operation calculate and retrieve waypoints and estimated travel times between events through the operation generate Routes, which ultimately returns a route object of the class Routes, containing all the necessary information to render the current day’s routes onto the map interface.

**Firebase DB class**

Returns successful authentication value (true) to the application for any GUI user request, contains the user data per user, which is loaded upon successful login, and in general manages all user’s information, which will be explained more in detail in the back-end class diagram.

**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), and that represents all the data to be saved into the Database (it’s a generalization which is explained more in detail in the Back-End class diagram).

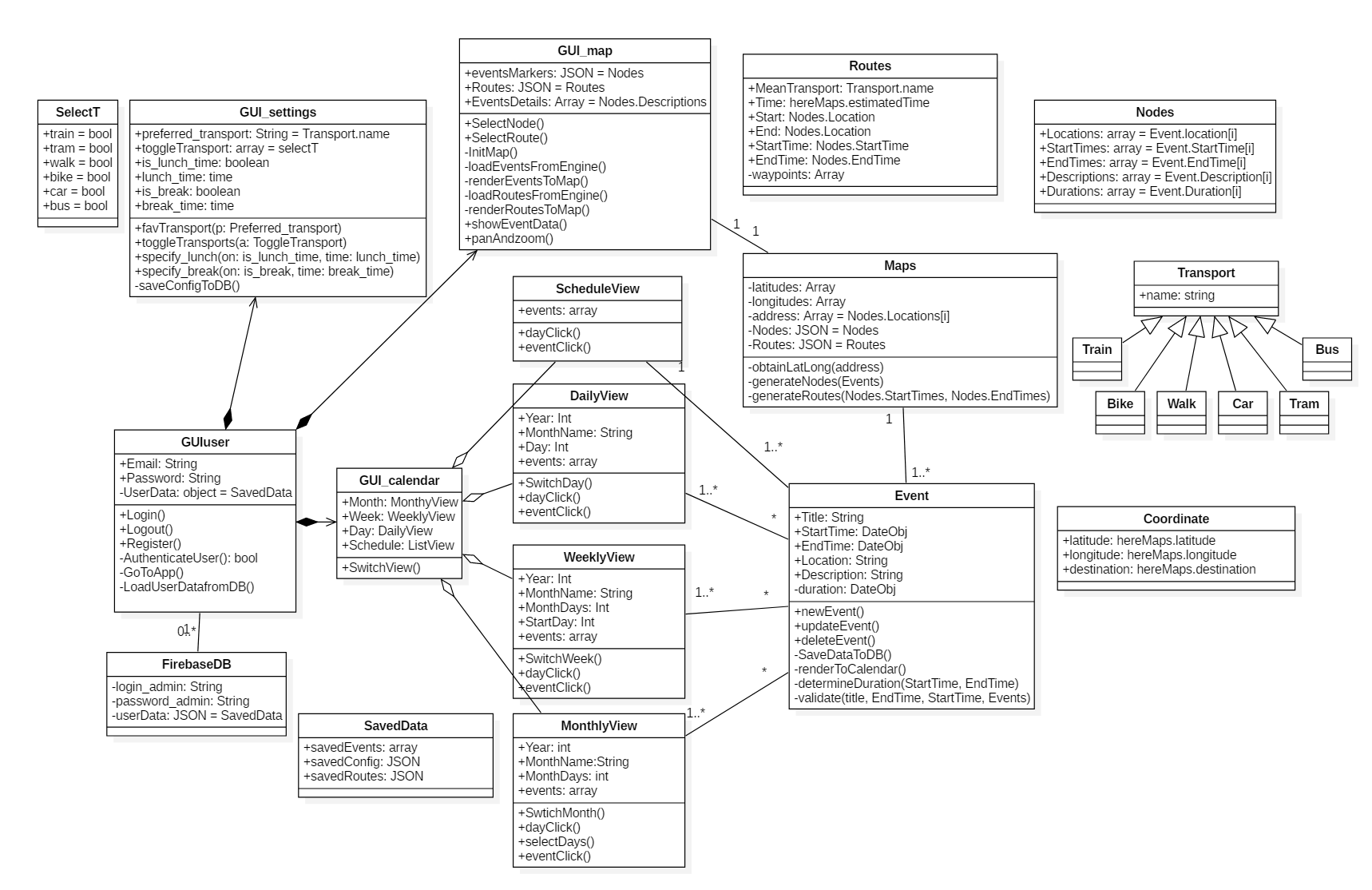


Fig. XXX. Front-End Class Diagram

#### 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 3 main classes: Firebase Database (which is a more detailed representation of the previous class mentioned above), App Engine which abstracts all the main app functionalities that require interaction with the database, and Settings which is essentially the same class as Settings GUI in the previous scheme but details the Database operations which were previously summarized into one, and compresses all the detailed attributes into a user settings JSON value.

**Firebase Database Class**

It has 3 attributes. UID corresponds to the user ID autogenerated by upon registration, is an array with all the events’ information contained in the Database (each event is a JSON file that contains a title, and ID, location, start and end times, and a description) and is a JSON value containing the user settings which have been detailed previously.

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.

Firebase Database operations are which returns a true value for any successful user authentication and enables the user data, the that retrieves a one time “snapshot” of the event data from the Database so it can be loaded and rendered into the app afterwards, that does a similar task but instead with the configuration data and that has the task of verifying that the user is logged in anytime any information is requested.

**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, the user events (retrieved from Database) and the user Routes (generated by several operations of the same class). Its operations are shared with the classes Map and Events from the front-end class diagram, specified here for clarity.

First, is the operation that validates the events by preventing time overlapping between them, or by preventing incorrect inputs such as empty title or incorrect times. Secondly, is an operation that takes and as parameters and by connecting with the maps API it generates a route object which is appended to . Sequentially, by using the maps API traffic the route time is calculated for all the considered transports and added to the route objects.

Once there are elements in the operation (compare travel time with time gap) compares the route times for each waypoint and compares it with the time remaining between a current event and its following event, and returns true or false whenever any event is being created or updated. Additionally, if true, it allows to render events and routes to the calendar interface and the map interface. These three operations are performed anytime a new event is created (once there’s more than 1 event) and anytime the user logs into a new session and are not stored into the database since the usability of this information depends on the current time and location of the user. Instead, they are recalculated with the new data and rendered onto the interfaces.

The goal of these operations is to make sure no events overlap; no routes are invalid and to ensure a continuous and coherent commuting for every day. The operations pushes validated event data to the database and as well as render the events and routes information onto the user interfaces.

**Settings Class**

The settings class contains as attribute the object, which has been previously shown as a JSON file which is updated anytime the user modifies the settings (shown in the settings GUI class in the previous diagram). Anytime settings are updated, the operation is activated.

Similarly, anytime the user logs in, the operation loads the object into the app to be used by the different modules. Lastly, to show the user the current settings via the interface, the operation loads the user settings onto the interface so they are shown anytime the user enters settings.

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.

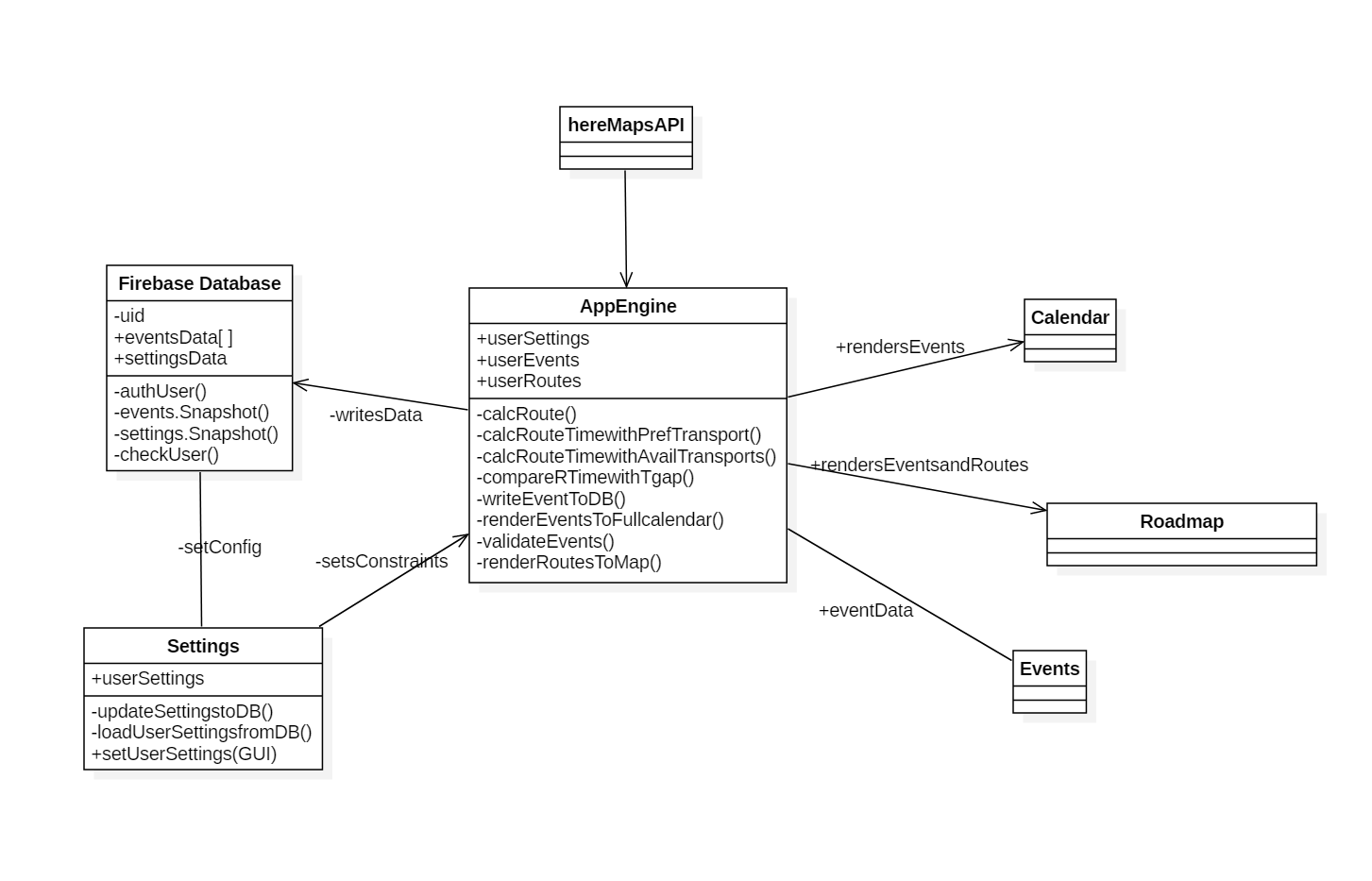


Fig. XXX. Back-End Class diagram

### Sequence diagrams

### Web interface, database

# Section 3

## Implementation

# Section 4

## Used tools

### Front end tools

### Back end tools

# Conclusion and future work