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](#_Toc1434554)

[Section 1 4](#_Toc1434555)

[Objective 4](#_Toc1434556)

[Requirements 4](#_Toc1434557)

[App description 5](#_Toc1434558)

[User Personas 5](#_Toc1434559)

[Who would want to use Smart Commute? 5](#_Toc1434560)

[Why would they want to use it? 5](#_Toc1434561)

[What are they looking for? 6](#_Toc1434562)

[How does Smart Commute provide the utilities that solve their problems? 6](#_Toc1434563)

[Overall structure 6](#_Toc1434564)

[Description of framework 6](#_Toc1434565)

[Section 2 6](#_Toc1434566)

[Design 6](#_Toc1434567)

[System architecture 6](#_Toc1434568)

[Use case diagrams 6](#_Toc1434569)

[Entity relationship diagrams 6](#_Toc1434570)

[Sequence diagrams 11](#_Toc1434571)

[Web interface, database 11](#_Toc1434572)

[Section 3 11](#_Toc1434573)

[Implementation 11](#_Toc1434574)

[Section 4 11](#_Toc1434575)

[Used tools 11](#_Toc1434576)

[Front end tools 11](#_Toc1434577)

[Back end tools 11](#_Toc1434578)

[Conclusion and future work 11](#_Toc1434579)

# 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, 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.

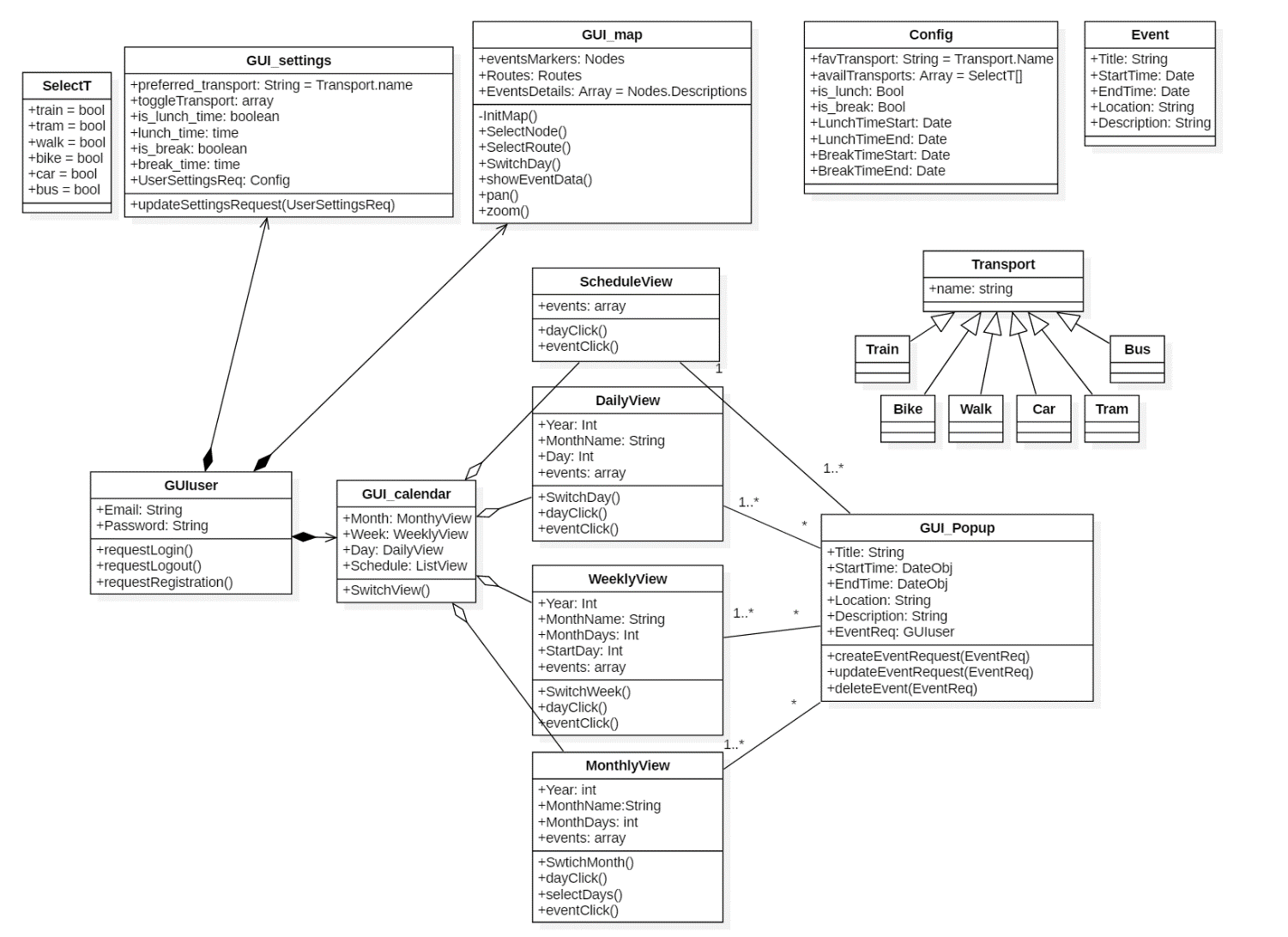


Fig. XXX. Front-End Class Diagram

#### 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.

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.

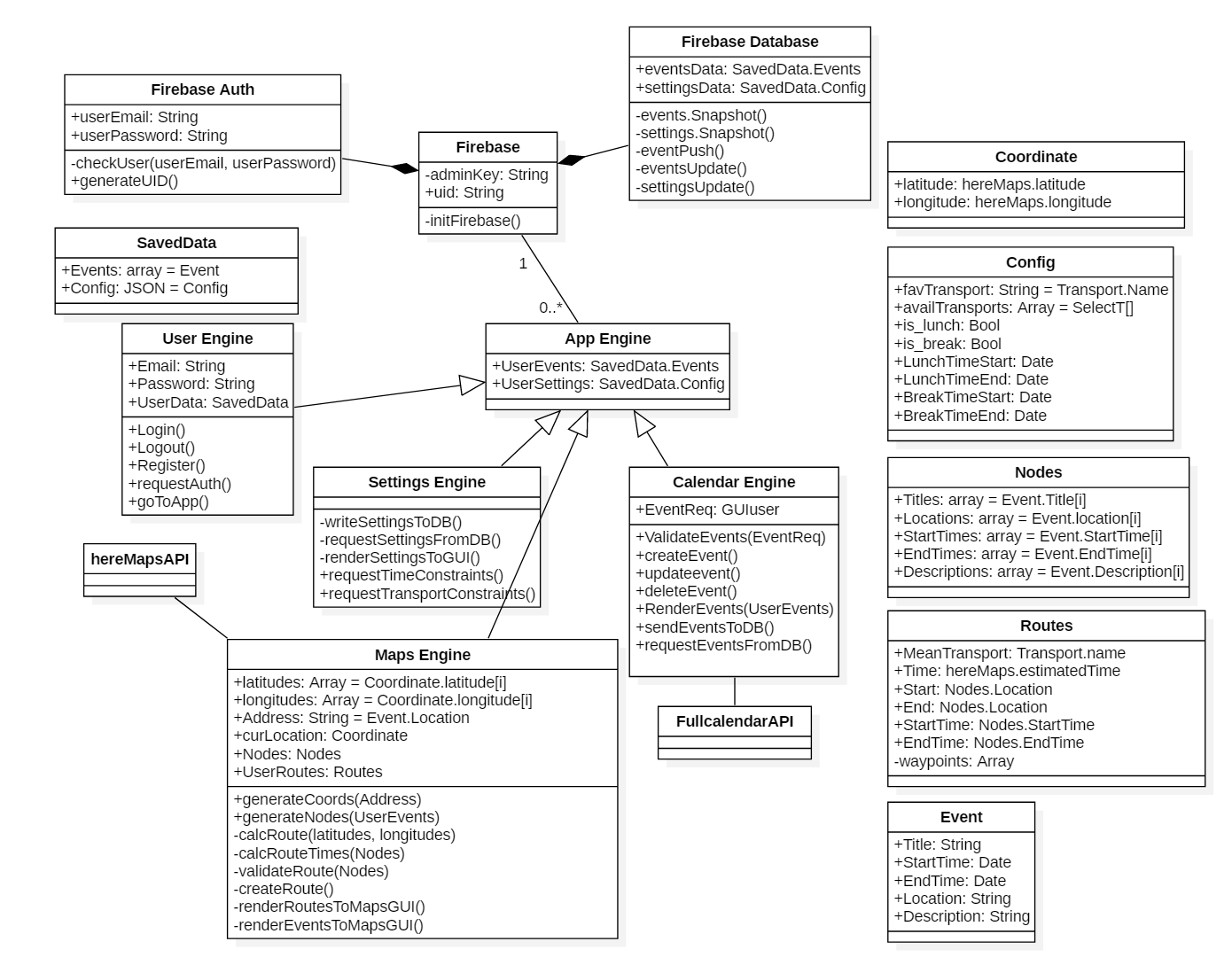


Fig. XXX. Back-End Class diagram

### Sequence diagrams

### Web interface, database

# Section 3

## Implementation

# Section 4

## Used tools

### Front end tools

### Back end tools

# Conclusion and future work