Software Engineering 2

Travlendar+

Implementation and **T**esting **D**ocument

Menchetti Guglielmo
Norcini Lorenzo
Scarlatti Tommaso

January 07, 2018

Contents

1 Introduction

1.1 Purpose and Scope

This Document represents the Implementation and Testing Document for the *Travlendar+* project.
The purpose of this document is to provide a comprehensive overview of the implementation and testing activity for the development of the application.
Travlendar+ application aims at providing a calendar based system that allows users to schedule meetings and appointments, giving information concerning travel means and times to reach a specific event also providing options to book rides .

1.2 Definitions, Acronyms, Abbreviations

1.2.1 Definitions

- **Framework**: is an abstraction in which software providing generic functionality can be selectively changed by additional user-written code.
- **Packet manager**: a software tool designed to optimize the download and storage of binary files, artifacts and packages used and produced in the software development process.

1.2.2 Acronyms

DBMS	Data Base Management System
HTTP	Hyper Text Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
API	Application Program Interface
REST	REpresentational State Transfer
ORM	Object-Relational Mapping
MVC	Model View Controller
SDK	Standard Development Kit
JSON	JavaScript Object Notation
CRUD	Create, Read, Update and Delete
LLVM	Low Level Virtual Machine
CSRF	Cross-Site Request Forgery

1.3 Reference Documents

- Design document

- RASD document
- Project assignment

1.4 Overview

The rest of the document is organized in this way:

- **Implemented requirements:** explains which functional requirements outlined in the RASD are accomplished, and how they are performed.
- **Design choices:** provides reasons about the implementation decisions taken in order to develop the application.
- **Source code structure:** explains and motivates how the source code is structured both in the front end and in the back end.
- **Testing:** provides the main testing cases applied to the the application
- **REST API:** describes the API implemented for the application.

2 Implemented requirements

In this section we describe the implemented functionalities with reference to the requirements outlined in the RASD document.
The requirement with their reference number green are the ones available in the current implementation, the others are in red.

2.1 Registration

- [R.1] A Guest must be able to register. During the registration the System will ask to provide credentials.
- [R.2] The System must check if the Guest credentials are valid.
- [R.3] The System must store all User credentials.

Database

The database stores the User information and credentials inside the table denominated 'users'.
Passwords are stored as salted hashes for security purposes.

Back-end

The Create User API [1] handles the registration process validating the provided data and handling communication with the database. A confirmation e-mail is sent to verify the validity of the mail address.

Front-end

The homepage of the client application for a Guest contains a registration section showing a form to fill with registration information. The client application validates the information, performs the request to the server and shows the response.

2.2 Login

- [R.4] The System must allow the User to log in using his/her personal credentials.

Database

The database stores the authentication tokens and client tokens in tables created by the Passport library for this purpose.

Back-end

The Login API [2] is managed by the Passport library and allows a registered User to obtain an OAuth token to authenticate future requests.

Front-end

The homepage of the application contains (if the User is not logged in) a login form. The client application validates the information, performs the request to the server and shows the response.

2.3 User management

- **[R.5]** The System must allow the User to change personal credentials.
- **[R.20]** The System allows the User to define specific constraint for each travel means.
- **[R.6]** The System must send a confirmation email if username, email or password is changed.

Database

The database stores the User information and credentials inside the table denominated 'users'.

Back-end

The Update User API [4] and the Update Preferences API [5] validate the request and allow to edit the User preferences and change the password.
In the current implementation no confirmation is required in order to perform these changes.

Front-end

The client application provides two sections (accessible by the side bar menu) named 'Settings' and 'Preferences' that allow respectively to change User information and set Travel preferences. In both cases the client application

validates the information, performs the request to the server and shows the response.

2.4 Event creation

- **[R.7]** The User must be allowed to create events, specifying all the mandatory fields.
- **[R.9]** The System allows the User to provide optional event information: membership category and a brief description.
- **[R.12]** The System must prompt the User to specify the location from which a certain event will be reached.
- **[R.16]** The System must allow the User to select a specific event.
- **[R.22]** The System must allow the User to select the Eco Mode in order to minimize carbon footprint.
- **[R.24]** The System must allow the User to create repetitive events.
- **[R.23]** The System must allow the User to create flexible events
- **[R.27]** The System must allow the User to select a travel mean for a repetitive event only for a specific day after its creation.

Database

The database stores the Event and Travel information in the tables the following tables: 'events', 'travels', 'flexibleEvent', 'repetitiveEvent'.

Back-end

The Create Event API [9] allows the creation of repetitive, flexible or standard Events with or without defining an associated Travel. It also provide validation of the input parameters. In the current implementation the Travel option for a repetitive Event is chosen once and applied globally for all repetitions.

Front-end

In the section page relative to a specific day, the application has a '+' button that allows to create a new Event.
This button open the Event creation page where all the details of the Event can be specified. Again the client application validates the information, performs the request to the server and shows the response.

2.5 Travel

- **[R.13]** The System must compute travel time between appointments.

Database

The database stores the Event and Travel information in the tables the following tables: 'events', 'travels', 'flexibleEvent', 'repetitiveEvent'.

Back-end

The Create Event API [12] returns a list of the Travel options available to reach an Event from a specified location.

Front-end

Inside the Event creation section the User may activate the Travel section. After specifying the start location, the User will be prompted to choose one of the available options.

2.6 Event visualization

- **[R.10]** The System must allow the User to view all the events for a specific day.
- **[R.16]** The System must allow the User to select a specific event.

Database

The database stores the Event and Travel information in the tables the following tables: 'events', 'travels', 'flexibleEvent', 'repetitiveEvent'.

Back-end

The Get Event API [10] and Get Schedule API[7] return respectively information about a single Event and information about all the Events within a specified time frame. It also provides validation of the input parameters.

Front-end

In the 'calendar' section(accessible by the side bar menu), days with at least one Event are marked with a dot. Selecting a specific day will show the list of that day's Events.

2.7 Event deletion

- [R.8][R.8] The User must be allowed to edit or delete a specific event in his/her schedule.

Database

The database stores the Event and Travel information in the tables the following tables: 'events', 'travels', 'flexibleEvent', 'repetitiveEvent'.

Back-end

The Delete Event API [11] allows to delete an Event. In the current implementation the editing of an event is not supported.

Front-end

The delete button is accessible swiping left on an Event. Tapping on the event will open the editing page, though at the moment the User is not allowed to apply the changes.

2.8 Valid Event Generation

- [R.13] The System must compute travel time between appointments.
- [R.14] The System must guarantee a feasible schedule, that is, the User is able to move from an appointment to another in time.

Back-end

The Create Event API [12] returns the available Travel options obtained from various external services. To each option is associated a set of adjustments to the schedule in order to make is feasible, such adjustments are computed by solving the CSP problem associated with the schedule.
In computing these adjustments we take into account travel time and the eventual flexibility of Events.

2.9 Feasibility check

2.10 Valid Event Generation

- **[R.11]** The System must check if the event created or edited by the User is feasible.
- **[R.15]** The System allows the User to add or edit an event only if it's not in conflict with other events already existent.
- **[R.25]** The System must check the feasibility of flexible events.
- **[R.26]** The System must check the repetitive events. The System must warn the User in case of conflicts for some of the day specified: the User will be allowed to add the event only in the day with no conflicts or to discard the event creation.

Back-end

During the Event creation process several consistency checks are preformed. The creation fails in case there is a conflict with the events of the schedule, or in case of other logical errors (e.g. an Event that ends before it starts).
In the current implementation in case of conflict the whole repetitive Event is discarded.

Front-end

The application will show an alert in case a feasibility check fails.

2.11 Notification

- **[R.21]** The System must allow the User to be notified a specific time before any event.

- [R.30] The System must allow to activate notification and setting its time.
- [R.31] The System must activate a ring at the time selected by the User.

Front-end

In the current implementation the notification function is disabled due to an issue with the client side platform.
The use the Push Notification functionalities available in the iOS development kit requires a certified Apple developer account.

2.12 Booking

- [R.28] The System must allow the User to select a specific travel means.
- [R.29] The System must provide an interface for third party services allowing the User to authenticate with the service.

Database

The Database stores information concerning the booking in the 'bookings' table. At each entry of this table is associate one of the travels of the User: the 'bookingId' of the 'events' table correspond to an 'id' field of the 'bookings' table.

Back-end

After the Event creation process, in which the User created an Event with a specific travel mean, the User is allowed to request information concerning available booking options. In the current implementation, the booking service is only provided for the taxi service, through the use of Uber external service.

Front-end

The book button is accessible swiping left on an Event. The User will be prompted to choose one of the options available. The outcome of the booking will be shown in an alert.

3 Design choices

In this section we present the reasons behind our implementation decisions for the design solutions shown in the Design Document.

3.1 Data layer

Due to our choice of Laravel as Backed Framework (3.2.2) the available options as DBMS were the following:

- MySQL
- PostgreSQL
- SQLite
- SQL Server

We chose to use postgreSQL since its the only open-source of these solutions, apart from SQLite (which is not a client-server database engine and therefore does not suit our purposes).
Also postgreSQL is known for its reliability and ability to operate at scale.

3.2 Business Logic Layer

3.2.1 External Services

The external services are used in the application to retrieve information concerning available travel options and related travel times. This is done through the use of the Google Maps Directions, Mapbox and Uber APIs.
The application also allows the user to book one of the available taxi option, provided by Uber service.
Furthermore, an external Mail service is used to send an email to the new registered User, in order to confirm the subscription to the application.

Google API

This API allows the developer to search for directions for several modes of transportation, including public transportation, driving and walking.
The main use of this service is to retrieve available travel options between locations and related travel times.
The calls to the API are made with the use of an access token through an HTTP GET request built with cURL.

The request parameters includes the type of transportation requested, origin and destination coordinates, expressed as floating point latitude and longitude. The response is a JSON object which contains informations such as travel times and waypoints for directions.

Mapbox API

This API, similarly to the previous one, allows the user to search for directions.
Their use in the application is due to the fact that Google API doesn't allows the developer to retrieve information concerning cycling mode of transportation.
Like before, the calls to the API are made with the use of an access token through an HTTP GET request built with cURL. The parameters of the requests are the origin and destination coordinates and the mean of transport, in this case we asked only for the cycling information. The response is a JSON object which contains informations such as travel times and waypoints for directions.

Uber API

Uber is one of the widely used taxi service in the world. This API allows the User to book one of the available services provided.
This API are used in order to retrieve information concerning travel time about Uber services and then allowing the User to book one of the available options.
The call to the API are used for two different scopes:

- **Retrieve travel options**
  For this operation, the calls to the API are made with the use of a server access token through an HTTP GET request built with cURL in which the parameters of the requests are the origin and destination coordinates.
  The response is a JSON object which contains travel options information.

- **Book a travel**
  In this case, the User has to login with his personal credential to the Uber platform. Once is logged in, the calls for the request of a booking are made with the personal access token of the User through an HTTP POST request built with cURL. The parameters of the request are the

access token and the informations concerning the travel.
The response is a JSON object which contains the booking information.

Mail Service

The Laravel framework used to build the application, provides many methods to send mails. One of the most common, and the one implemented in the application, is the one in which Google Mail is used as mail server.

3.2.2 Framework

For the Business Logic Layer we chose to implement our solution using the **Laravel Framework**.
Laravel is a free, open-source PHP web framework based on MVC, with an active and growing community behind it.
In this section we present the reasons behind this choice.

Language

Laravel is a PHP based framework. PHP is a server scripting language and arguably one the most popular languages for back-end development.
Furthermore from the last version Laravel supports PHP 7 which offers remarkable improvements both in terms of performance and security.

Routing

Laravel provides **routes management** out of the box. It allows easy binding of the routes with the methods responsible for handling incoming requests.
It also allows to create routes for the managing of CRUD resources binding each HTTP method to the specified endpoint and to the corresponding methods, as shown in the table below.

Request Type	Path	Controller Method	Route Name	Purpose
GET	/resource	index	resource.index	Show a list of the items in resource
POST	/resource	store	resource.store	Store a new item in resource
GET	/resource/create	create	resource.create	Show a form to create new item in resource
GET	/resource/{id}	show($id)	resource.show	Show an item of resource
GET	/resource/{id}/edit	edit($id)	resource.edit	Show a form to edit an item of resource
PUT or PATCH	/resource/{id}	update($id)	resource.update	Edit an item of resource
DELETE	/resource/{id}	destroy($id)	resource.destroy	Delete an item of resource

Figure 1: Laravel CRUD routing

Further details about the available routes and implemented APIs are in section 6.

ORM

Laravel includes among its default packages also **Eloquent ORM**.
This package provides a simple ActiveRecord implementation for working with a database.
The active record pattern is an architectural pattern that stores in-memory object data in relational databases or as described by its creator: "An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.".
In Laravel each database table has a corresponding "Model" which is used to interact with that table, allowing to easily define relationships between different Models.
Further details about the Database and Model structure are in section 4.2

Database Management and Query Builder

Laravel provides a uniform way to interface with a variety of different DBMSs, it allows to design the database schema and migrate it on different DBMS solutions.

Furthermore Laravel's database query builder provides a convenient, fluent interface to create and run database queries.
It can be used to perform most database operations in the application and works on all supported database systems.
The Laravel query builder uses PDO parameter binding to protect against SQL injection attacks.

Authentication

Laravel only offers session based authentication out of the box.
For the development of APIs there is the need of a different type of authentication since there is no lasting session, for this reason we use an external package called Passport, that easily integrates with Laravel.
Laravel Passport is an OAuth2 server and API authentication package that provides token based authentication.

Testing

Laravel is built with testing in mind. In fact, support for testing with PHPUnit is included by default.
The framework also ships with convenient helper methods that allow you to expressively test your applications. Along with the functionalities provided by the PHPUnit testing suite, Laravel allows to:

- simulate an incoming request to an endpoint and the subsequent response.
- make assertions concerning the state of the Database.
- activate and deactivate the various middlewares (e.g. the authentication check or the active account check).
- choose the acting role (e.g. a specific account or privilege level).
- wrap each test function in a database transaction allowing each test to be independent.

3.3 Presentation Layer/Client

3.3.1 iOS

For the presentation (front end) layer, we chose to develop a native iOS app for iPhones and iPads written in Swift. We took this choice mainly

for the widespread diffusion of the iOS operative system and for the several advantages of a native application, such as:

- It is easier to work with and also performs faster on the device
- Provides full support from the App Store
- It is complete safe and secure
- Native look and feel

On the other hand, this choice leads to a loss in terms of portability and higher maintenance costs, as we do not have a common code base for different platforms.

3.3.2 Swift

Swift is a general-purpose, multi-paradigm, compiled programming language developed by Apple Inc. for iOS, macOS, watchOS, tvOS, and Linux. Swift is designed to work with Apple's *Cocoa* and *Cocoa Touch* frameworks and the large body of existing Objective-C code written for Apple products. It is built with the open source LLVM compiler framework and has been included in Xcode since version 6.

Swift defines away large classes of common programming errors by adopting modern programming patterns:

- Variables are always initialized before use
- Array indexes are checked for out-of-bounds errors
- Integers are checked for overflow
- Optionals ensure that nil values are handled explicitly
- Memory is managed automatically
- Error handling allows controlled recovery from unexpected failures

3.3.3 External frameworks

We used several external frameworks to accomplish what we stated in the RASD and DD document in a more elegant and secure way. All the frameworks were installed with *CocoaPods*, a dependency manager for Swift and Objective-C Cocoa projects. It has over 41 thousand libraries and is used in over 3 million apps.

- **SWRevealViewController**: to add a slide-out side bar menu.
- **JTAppleCalendar**: to build a calendar from scratch and customize it for our needs.
- **SwiftDate**: to manage Dates and Timezones in Swift.
- **Alamofire**: to make elegant network HTTP requests.
- **CropViewController**: to perform basic manipulation on UIImage objects; specifically cropping and some basic rotations.
- **KeychainSwift**: helper functions for saving text in Keychain securely.
- **UberRides:** to integrate the Uber Rides API into our iOS app.
- **Quick/Nimble**: to help making asynchronous integration tests in a proper way.

3.3.4 Apple frameworks

In addition to all the external frameworks presented above, it is worth mentioning the adopted basic Apple frameworks:

- **Foundation Kit**: provides a base layer of functionality for the app, including data storage and persistence, text processing, date and time calculations, sorting and filtering, and networking.
- **UIKit**: provides the required infrastructure for the app. It provides the window and view architecture for implementing the interface, the event handling infrastructure for delivering Multitouch and other types of input.
- **MapKit**: to display map or satellite imagery directly from the app's interface, call out points of interest, and determine placemark information for map coordinates.

3.3.5 Model View Controller

Applications developed for iOS follow the MVC pattern. The MVC design pattern assigns objects in an application to one of three roles: model, view, or controller.

The pattern defines not only the roles objects play in the application, it defines the way objects communicate with each other. Each of the three

types of objects is separated from the others by abstract boundaries and communicates with objects of the other types across those boundaries.

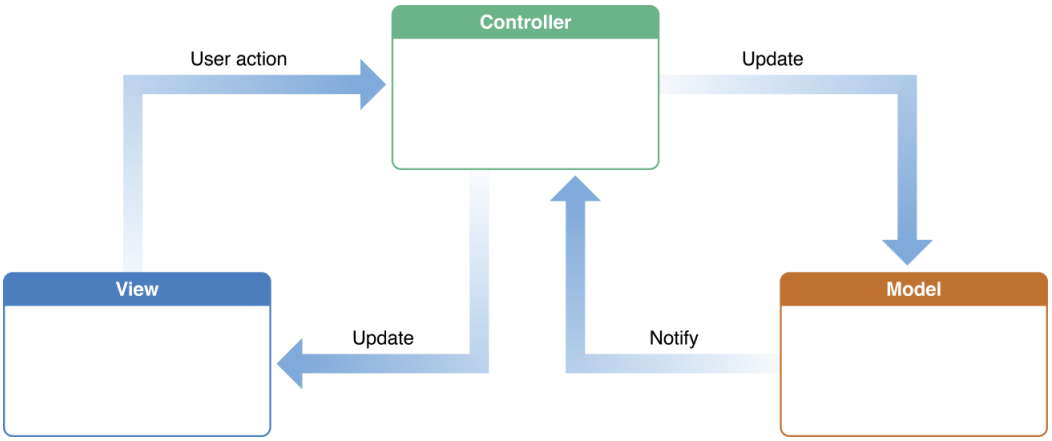

Figure 2: Model View Controller schema

- **Model:** model objects encapsulate the data specific to an application and define the logic and computation that manipulate and process that data.
- **View:** a view object is an object in an application that users can see. A view object knows how to draw itself and can respond to user actions. A major purpose of view objects is to display data from the application's model objects and to enable the editing of that data.
- **Controller:** controller object acts as an intermediary between one or more of an application's view objects and one or more of its model objects. Controller objects are thus a conduit through which view objects learn about changes in model objects and vice versa.

MVC roles played by an object can be merged. *Travlendar+* widely uses **View Controllers**, controllers that concerns their selves mostly with the view layer. They "own" the interface (the views); their primary responsibilities are to manage the interface and communicate with the model.

4 Source code structure

4.1 Laravel Project Structure

The following section describes the structure of the Laravel Project, meaning the structure of the source code for the back-end application.
Please note that this describes the structure of the code and only broadly the functionalities, for more information please refer to the rest of this document and to the comments associated with the source code, also more details about the standard structure of a Laravel project can be found in the official documentation.

The App Directory

The app directory contains the core code of the application.
The most important subdirectories in this folder are Http and Models. *Models* contain the classes comprising the application model.
Http contains what is effectively the main application logic and has again several subfolders:

- The *Controllers* directory contains the classes that handle the methods binded to the endpoints.
- The *Helpers* directory contains classes and scripts that provide functionalities used by the controllers.
- The *Interfaces* directory contains classes that provide access to external services mentioned in section 3.2.1.
- The *Middleware* directory contains the definition of HTTP filters that allow to filter the requests before or after such requests have been elaborated by the controllers.

The Bootstrap Directory

The bootstrap directory contains the *app.php* file which bootstraps the framework. This directory also houses a cache directory which contains framework generated files for performance optimization such as the route and services cache files.

The Config Directory

The *Config* directory contains all of the application's configuration files.

The Database Directory

The *Database* directory contains database migration and seeds. As we said in section 3.2.2, Laravel uses the so-called database migrations in order to increase the portability of the application.
Each of the files found in the migrations defines a table in the database (see section 4.2) with an additional file that defines the relationships between them.

The Public Directory

The *Public* directory contains the *index.php* file, which is the entry point for all requests entering the application and configures autoloading. This directory also houses assets such as images, JavaScript, and CSS.
In our case it only contains the application Logo since there we do not use other resources.

The Resources Directory

The *Resources* directory contains views as well as raw, un-compiled assets such as LESS, SASS, or JavaScript. This directory also houses language files. In the *Views* subdirectory are the web files that contain the UI for the activation activation mail and confirmation page.

The Routes Directory

The *Routes* directory contains all of the route definitions for the application. By default, several route files are included with Laravel: *web.php*, *api.php*, *console.php* and *channels.php*.
The **web.php** file contains routes that are in the web middleware group, which provides session state, CSRF protection, and cookie encryption. The only route defined here is the one for the account activation endpoint since the rest of the functionalities are implemented via RESTful API.
The **api.php** file contains routes that are in the api middleware group, which provides rate limiting. These routes are intended to be stateless, so requests entering the application through these routes are intended to be authenticated via tokens and will not have access to session state.
This file defines all the routes for the implemented APIs.
The **console.php** file is where all Closure based console commands are defined. Each Closure is bound to a command instance allowing a simple approach to interacting with each command's IO methods. Even though this file does not define HTTP routes, it defines console based entry points

(routes) into the application. Apart from Laravel default CLI commands we did not implemented any additional ones so no routes are defined here.
The **channels.php** file is where all of the supported event broadcasting channels are registered, again no additional routes are defined here.

The Storage Directory

The *Storage* directory contains compiled Blade templates, file based sessions, file caches, and other files generated by the framework. This directory is segregated into app, framework, and logs directories. The app directory may be used to store any files generated by the application. The framework directory is used to store framework generated files and caches. Finally, the logs directory contains the application's log files.
The *storage/app/public* directory may be used to store user-generated files. Other than the framework generated files this folder is not used by our application.

The Tests Directory

The *Tests* directory contains automated tests. The *Unit* subdirectory contains the Unit tests, while the *Feature* subdirectory contains the Integration tests that verify the APIs functionalities.

The Vendor Directory

The *Vendor* directory contains the Composer dependencies.
Mainly the PHPUnit testing suite and Passport authentication package.

4.2 Backend model description

As we already said there is a tight correspondence between the Database Schema and the Model given the use of the ORM pattern.
The implemented structure of the Database and therefore of the various Models is depicted below.

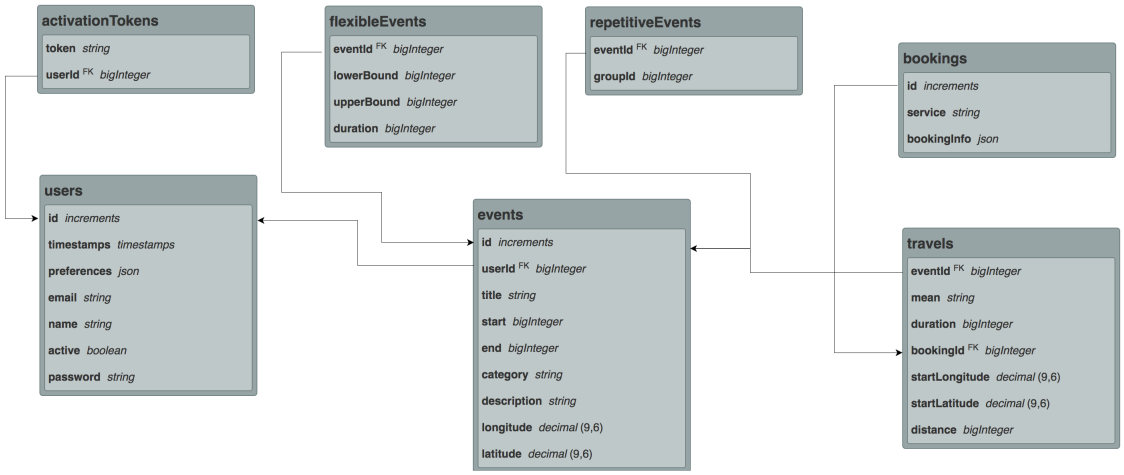

Figure 3: Model and Database Structure

4.3 Xcode project structure

The following section describes the structure of the Xcode Project, meaning the structure of the source code for the front-end iOS application.
Please note that this describes the structure of the code and only broadly the functionalities. For more details please refer to the source code comments and the Xcode and iOS documentation.

As we outlined in the previous section, view controllers are the foundation of our app's internal structure. Each view controller manages a portion of your app's user interface as well as the interactions between that interface and the model data. View controllers also facilitate transitions between different parts of your user interface.

In order to make the code as clear and maintainable as possible, we organized files in different sections, according to their functionalities.

Model

This section was intended to wrap all the relevant data structure useful to keep locally synchronized the data requested from the server. Classes have also supporting structure to help the JSON coding/decoding process. Model files are listed below:

- *User.swift*: this is the User model. It has a main struct User which en-

capsulates all the user-related information such as name, email, profile picture and the token to perform authenticated requests to the server. Furthermore, it contains structure to help the decoding of user credentials and owns the function which encapsulates the HTTP request to retrieve them.

- *Preferences.swift*: this is the Preferences model. It has a main struct *Preferences* which encapsulates all the user preferences for different travel means. It also contains support structures to help the coding/decoding process for the data sent/received to the server.

- *Day.swift*: this is the day model. It holds all the day-related data such as the date of day currently selected by the user and the events list of the day. It also contains methods and structures to make easier the JSON coding/decoding and the string formatting.

- *Event.swift*: this is the event model. It contains a main struct Event which is used for the coding/decoding of the JSON to send to or receive from the server. This structure contains all the event-related information, included the optional associated travel option. It also takes into account the current event selected in the *DayViewController* and the list of associated travels.

- *Booking.swift*: this is the booking model. It contains a main struct *BookingInfo* which is used to encapsulate all the booking-related information received from the server. It also holds a global variable *selectedBookingOption* which is used to store the currently available booking options retrieved from the server

Log View Controllers

All the view controllers related with user authentication and sign up process.

- *LogViewController.swift*: it is the initial view controller. It handles the log in process sending an authentication request to the server. In case of success moves to the *DayViewController*.

- *SignUpViewController.swift*: displays the registration form and, in case of success, segues to *StartViewController*.

- *StartViewController.swift*: shows to the user that a confirmation email has been sent to his/her account and let go back to the *LogViewController* for the authentication.

Side Menu View Controllers

Gathers all the files that generate the left side menu, including the Objective-C *SWRevealViewController* external library.

- *MenuViewController.swift*: displays the slide out left side menu and the relative options.
- *MenuTableViewCell.swift*: represents a cell of the menu.
- *SWRevealViewController.h*: header file of the library.
- *SWRevealViewController.m*: source file of the library.

Navigation View Controllers

This section contains all and only the view controllers reachable from the left-side menu.

- *DayViewController.swift*: this is the "home" of the application. It is the first view displayed after the log in, showing the current date and the associated events, organized in a scrollable view. Every time the view is loaded, it performs an HTTP GET request to retrieve the events of the day.
  On the top-right corner it has an add button that leads to the *AddEventViewController*. Furthermore, if the user taps on an event, a view with detailed information will be displayed.
- *CalendarViewController.swift*: shows an horizontally scrollable calendar, marking the current day in red and each day with at least one event with a dot. If a day cell in the calendar is tapped, the *DayViewController* will be displayed, showing the events of the selected day, if present.
- *PreferencesViewController.swift*: displays a scrollable view to set/edit travel means preferences. On the top, means can be globally activate/deactivate through toggle buttons. The save button in the right corner embeds all the infos in a JSON file and send it to the server to save the preferences.
- *SettingsViewController.swift*: within this view, users can edit their profile picture, their name and change their password or email.
- *CreditsViewController.swift*: displays the credits of the application, showing the *Politecnico di Milano* logo and the developers' name.

Event-related View Controllers

This section embeds all the view controllers connected with event management.

- *AddEventViewController.swift*: handles the creation of an event, displaying a form to fill with all the necessary information, included the location of the event (handled separately from the *LocalSearchViewController*) and an eventually travel option. When the top-right "Done" button is tapped, an HTTP POST request is sent to the server.
- *TravelViewController.swift*: handles the retrieve of the possible travel option of an event from the server and displays them in a scrollable view.

Booking View Controllers

All the view controllers involved in the booking process of third-party services.

- *BookingViewController.swift*: it currently handles the Uber booking functionalities. At the top it displays the Uber sign in button that leads to the Uber app in order to authenticate the user.

Settings View Controllers

Gathers all the view controllers reachable from the *SettingsViewController*.

- *ChangePasswordViewController.swift*: handles the change password process. Shows two text field for the old and the new password and has a "save" button to send the information to the server.

Cells Table Views

In this section are gathered all the files that represent a dynamic cell in a table view controller.

- *EventCell.swift*: is a cell of the event table view in the *DayViewController*. It shows the title, the description and when the event takes places. It also has two icons: one for the travel mean associated, one for the booking option.
- *CalendarCell.swift*: is a cell of the calendar collection view in the *CalendarViewController*. It shows the number of the corresponding day and a bullet if the day contains at least one event.

- *TravelCell.swift*: is a cell that represents a travel option in the *TravelViewController*. It displays the mean, the duration and the distance.
- *BookingCell.swift*: is a cell that represents a booking option in the *BookingViewController*. It shows the type of mean, the duration, the distance and the price.

Location View Controllers

These are all the view controllers connected with the location menagment.

- *LocalSearchViewController.swift*: displays a view of the map through the Apple MapKit and provides a search bar to seek locations.
- *LocationSearchTableViewController.swift*: overlapping table view which shows the places that match the request.

Supporting files

- *Main.storyboard*: a visual representation of the user interface of the application, showing screens of content and the connections between those screens.
- *Info.plist*: is a structured text file that contains essential configuration information for a bundled executable.
- *AppDelegate.swift*: its in charge of the application life cycle. Providing a starting point, delegating what to do when the app is put in the background, reactivated, or being terminated, and responds to events that target the app itself rather than specific view.
- *Podfile.txt*: is a specification that describes the dependencies of the targets of the Xcode project.

4.4 Frontend model description

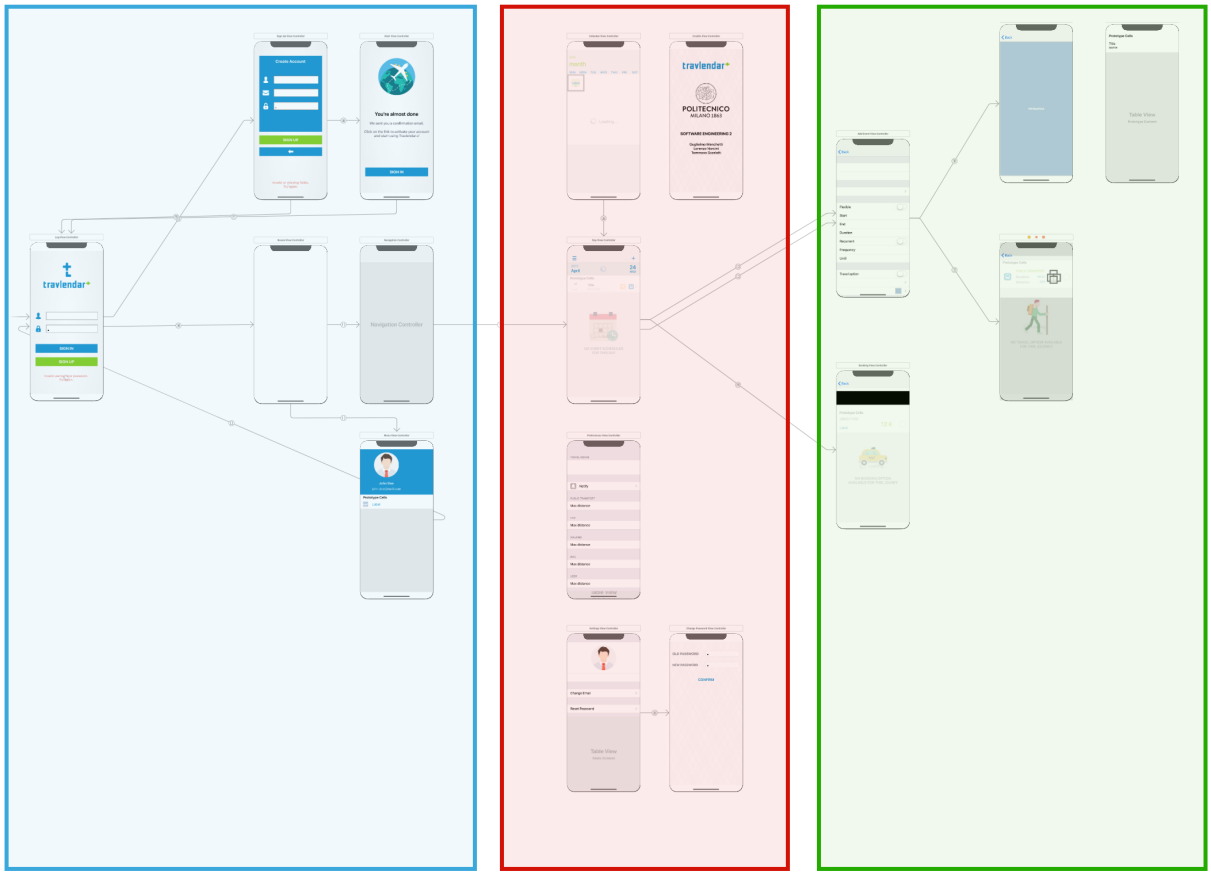

Figure 4: Storyboard structure

The above figure shows an overall view of the Main.storyboard file. Here we can clearly see the distinct sections of the application and the connections between them.

For clarity reasons, view controllers have been divided into three different logic categories.

- **Authentication, Registration and Side Menu** (blue)
  In this category we have all the view controllers related with authentication and registration process and those responsible for the appearance of the slide-out left side menu.
  They are the entry point of the application. Each new user must pass

first from the *SignUpViewController*, then to the *LogInViewController* and finally to the *MenuViewController* to navigate into the application.

- **Navigation** (red)
  This category contains all and only the view controllers reachable from the left-side menu.

- **Event-related** (green)
  All the view controllers reachable from the *DayViewController*. They are all related with the creation or the modification of an event.

Screenshots

For each category we show here a bunch of screenshots of the application.

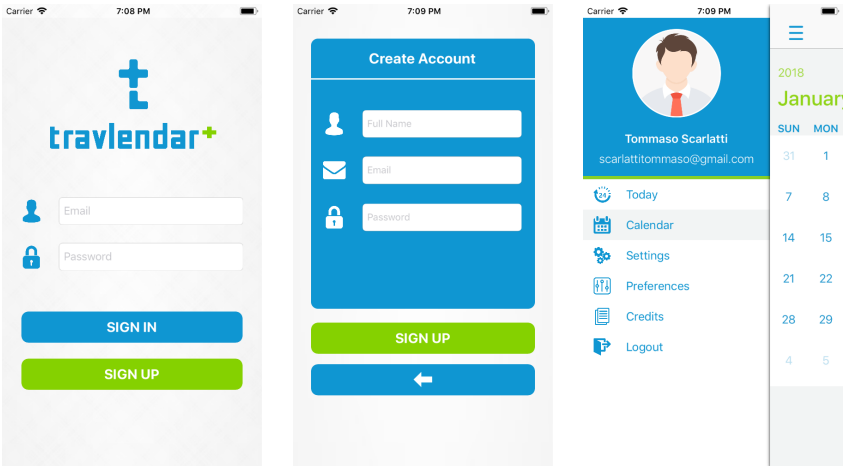

Figure 5: Authentication, Registration and Side Menu

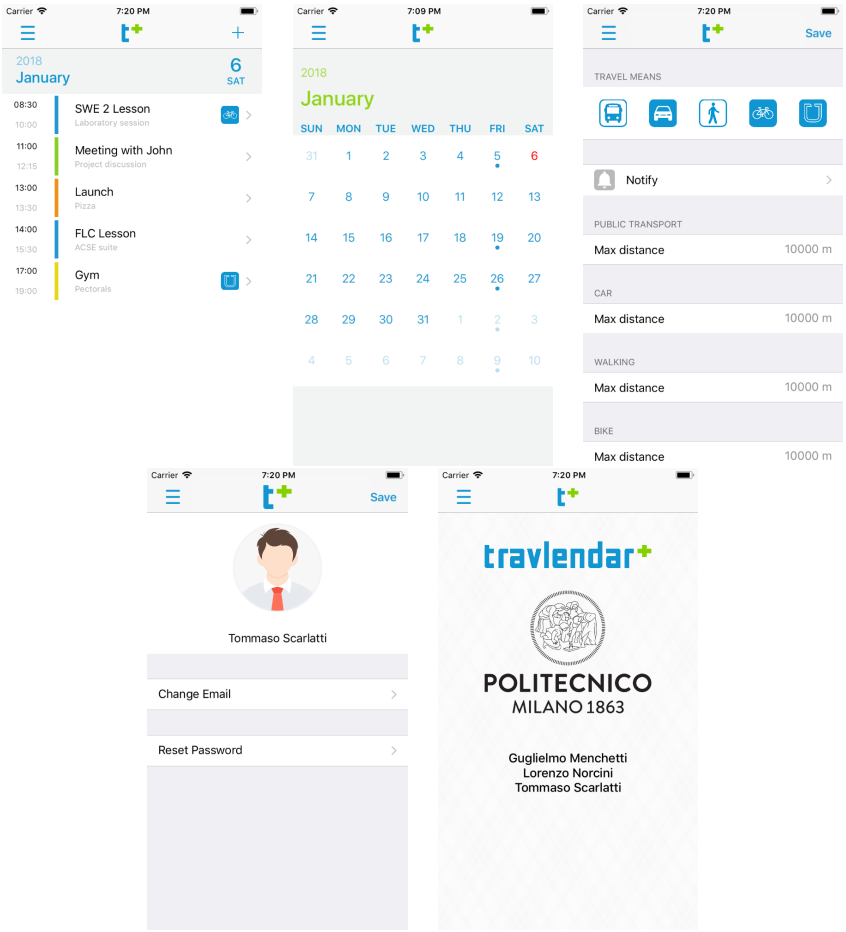

Figure 6: Navigation View Controllers

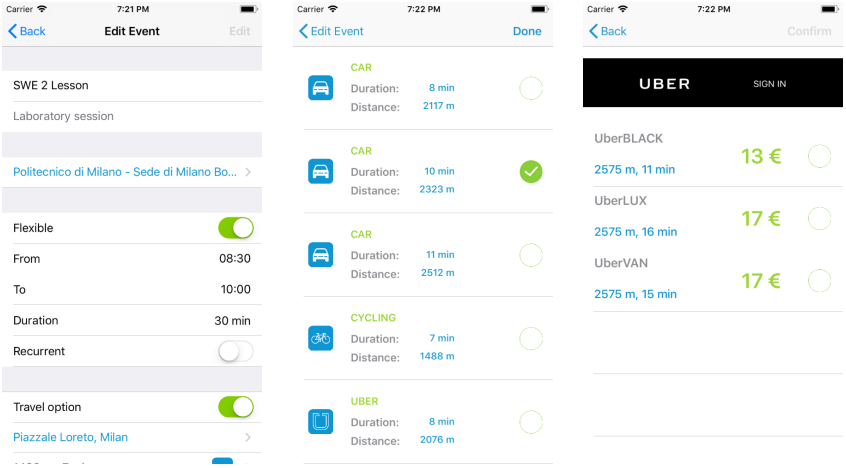

Figure 7: Event-related View Controllers

5 Testing

This section describes in the following order the performed Unit Tests, Integration/Feature Tests and System Integration Tests.

5.1 Unit Testing

Available Booking options		
state	request	assertions
Indifferent	Event and valid start travel coordinates	- Response message is "success" - Response data is not empty
Indifferent	Event with invalid start travel coordinates	- Response message is "Error in Uber services: no available travels" - Response data is empty

Create Booking		
state	request	assertions
Indifferent	Available Booking option with valid user token	- Response message is "success" - Response data is an instance of the Booking class - Response data is not equal to an empty Booking
Indifferent	Invalid Booking option with valid user token	- Response message is not "success" - Response data is equal to an empty Booking
Indifferent	Valid Booking option with invalid user token	- Response message is "unauthorized" - Response data is equal to an empty Booking

Feasibility Check		
state	input	assertions
Indifferent	Not an Event object	- TypeError
No Event in schedule causing conflict with input	Event with valid data	- feasibility result is true. - there are no errors.
Indifferent	Event with start before end	- feasibility result is false. - the error cause is "Event data not valid". - the error details are "end before start".
Indifferent	Event with flexible bounds too strict	- feasibility result is false. - the error cause is "Event data not valid". - the error details are "flexible bounds too strict".
Indifferent	Event with inconsistent flexible bounds	- feasibility result is false. - the error cause is "Event data not valid". - the error details are "flexible bounds do not respect fixed bounds".
Indifferent	Event with travel that violates preferences	- feasibility result is false. - the error cause is "Preferences not satisfied".
Events in schedule causing conflict with input	Event with valid data	- feasibility result is false. - the error cause is "Schedule conflict". - the error details are the ids of the conflicting Events.

Solve CSP		
state	input	assertions
Indifferent	Not an Array object	- TypeError
Indifferent	Array of CSP variable that have a solution	- returns valid assignments for the CSP variables
Indifferent	Array of CSP variable that have no solution	- returns an empty array

5.2 Integration/Feature Testing

In this section we present the Integration tests between the components of the back-end system.

As Integration we test the Features provided by each API.
Please note that for privileged endpoints we have omitted the cases in which the request is not authenticated.
The reason for this is that the authentication middleware is a separated component and is tested separately.
Also the expression "with valid data" refers to the validation rules associated with each request.

User Registration [API 1]		
state	request	assertions
Mail not already associated with another User	Request with valid data	- Response status code is 200 - Response message is "User creation successful" - The database has an entry with the new User data
Mail already associated with another User	Request with valid data	- Response status code is 409 - Response message is "Data not correct, possible mail duplicate" - The database has not an entry with the new User data
Indifferent	Request with short password	- Response status code is 422 - Response message is "The given data was invalid." - The error is "The password must be at least 10 characters." - The database has not an entry with the new User data
Indifferent	Request with invalid data	- Response status code is 422 - Response message is "The given data was invalid." - The database has not an entry with the new User data

Update User [API 4]		
state	request	assertions
Indifferent	Request with valid data	- Response status code is 200 - Response message is "User successfully modified" - The database entry for the User is modified
Indifferent	Request where current password provided does not match the one in storage	- Response status code is 403 - Response message is "The password provided does not match the current one" - The database entry for the User is not modified
Indifferent	Request where User id provided does not match the one associated with the account	- Response status code is 403 - Response message is "Provided User id and User token do not match" - The database entry for the User is not modified
Indifferent	Request with invalid data	- Response status code is 422 - Response message is "The given data was invalid." - The database has not an entry with the new User data

Update User Preferences[API 5]		
state	request	assertions
Indifferent	Request with valid data	- Response status code is 200 - Response message is "Preferences successfully modified" - The database entry for the User is modified
Indifferent	Request with malformed preference data	- Response status code is 400 - Response message is "Payload not convertible to object" - The database entry for the User is not modified
Indifferent	Request with invalid data	- Response status code is 422 - Response message is "The given data was invalid." - The database entry for the User is not modified

Get User Information [API 3]		
state	request	assertions
Indifferent	Request	- Response status code is 200 - Response message is "Request successful" - Returns the User information

Create Event [API 9]		
state	request	assertions
No Event in schedule causing conflict with the new Event	Request for the creation of a Standard Event with valid data	- Response status code is 200 - Response message is "Event creation successful" - The database has an entry with the new Event data
No Event in schedule causing conflict with the new Event	Request for the creation of an Event with Travel with valid data	- Response status code is 200 - Response message is "Event creation successful" - The database has an entry with the new Event data - The database has an entry with the new Travel data
Events in schedule causing conflict with the new Event	Request for the creation of an Event Standard with valid data	- Response status code is 400 - Response message is "Feasibility check failed" - The database has not an entry with the new Event data
Events in schedule causing conflict with the new Event	Request for the creation of an Event with Travel with valid data	- Response status code is 400 - Response message is "Feasibility check failed" - The database has not an entry with the new Event data - The database has not an entry with the new Travel data
No Event in schedule causing conflict with the new Event	Request for the creation of an Repetitive Event with valid data	- Response status code is 200 - Response message is "Event creation successful" - The database has an entry with the new Event data for each repetition - The database has an entry with the new Repetitive Event data for each repetition

Events in schedule causing conflict with the new Event	Request for the creation of an Repetitive Event with valid data	- Response status code is 400 - Response message is "Feasibility check failed" - The database has not an entry with the new Event data for any repetition - The database has not an entry with the new Repetitive Event data for any repetition
Indifferent	Request for the creation of an Event (of any kind) with invalid data	- Response status code is 422 - Response message is "The given data was invalid" - The database has not an entry with the new Event data
No Event in schedule causing conflict with the new Event	Request for the creation of an Flexible Event with valid data	- Response status code is 200 - Response message is "Event creation successful" - The database has an entry with the new Event data - The database has an entry with the new Flexible Event data
Events in schedule causing conflict with the new Event	Request for the creation of an Flexible Event with valid data	- Response status code is 400 - Response message is "Feasibility check failed" - The database has no an entry with the new Event data - The database has no an entry with the new Flexible Event data
Events in schedule causing conflict with the new Event	Request for the creation of an Event (of any kind but repetitive) with valid data and adjustments	- Response status code is 200 - Response message is "Event creation successful" - The database has an entry with the new Event data - The database has an entry with eventual additional information

Get Event Information [API 10]		
state	request	assertions
Indifferent	Request with a valid id	- Response status code is 200 - Response message is "Request successful" - Returns Event information
Indifferent	Request with an invalid id	- Response status code is 403 - Response message is "The requested resource does not belong to the current User or does not exists"

Delete Event [API 11]		
state	request	assertions
Indifferent	Request with a valid id	- Response status code is 200 - Response message is "Event deleted"
Indifferent	Request with an invalid id	- Response status code is 403 - Response message is "The requested resource does not belong to the current User or does not exists"

Generate Valid Event [API 12]		
state	request	assertions
Schedule is adjustable	Request with valid data	- Response status code is 200 - Response message is "Request successful" - Events created with provided informations are feasible
Schedule is not adjustable	Request with valid data	- Response status code is 200 - Response message is "Request successful" - The information provided does not allow to create feasible Events
Indifferent	Request with invalid data	- Response status code is 422 - Response message is "The given data was invalid"

Available Booking options [API 13]		
state	request	assertions
Indifferent	Request with valid data	- Response status code is 200 - Response message is "Request successful" - Response data is not empty
Indifferent	Request with invalid Event identifier	- Response status code is 404 - Response message is "Event not found" - Response data is empty
Indifferent	Request with invalid start travel coordinates	- Response status code is 404 - Response message is "Error in Uber services: no available travels" - Response data is empty

Create Booking [API 14]		
state	request	assertions
Indifferent	Request with valid data	- Response status code is 200 - Response message is "Request successful" - Database has an entry with the created booking - Database has travel with bookingId field same as the booking identifier created - Booking information in the database are the same as the one created
Indifferent	Request with valid data but requested too early	- Response status code is 400 - Response message is "Is not possible to book a ride until 60 minutes before the start of the event"
Indifferent	Request with valid data but requested too late	- Response status code is 400 - Response message is "Is not possible to book a ride after the start of the event"

Retrieve Current Booking [API 16]		
state	request	assertions
There is the requested active booking in the database	Request with valid booking identifier	- Response status code is 200 - Response message is "Current ride available" - Response data is instance of class Booking
There is not the requested booking in the database	Request with invalid booking identifier	- Response status code is 400 - Response message is "Booking doesn't exists" - Response data is null

Delete Current Booking [API 15]		
state	request	assertions
There is the requested active booking in the database	Request with valid token	- Response status code is 200 - Response message is "Booking deletion successful"
There is not the requested booking in the database	Request when the resource doesn't exists	- Response status code is 403 - Response message is "User has no active booking"

5.3 iOS Integration testing

In this section we provide the integration tests between the iOS app and the back end.

User Authentication		
state	request	assertions
Valid username and password set	Request with provided credentials	- Response status code is 200 - User token is not nil

Get Events		
state	request	assertions
Added two events to a specific day	Request with the start and the end of the day as parameters	- Response status code is 200 - Events are 2

Get Travels		
state	request	assertions
Event with travel options set	Request with the position of the event and the start position of the travel	- Response status code is 200 - Travels of selected event is not nil - Number of travels is greater than 0

Get Days with Events		
state	request	assertions
At least one event is present in the calendar of the test user	Request with the credentials of the test user	- Response status code is 200 - Days with events is not nil - Number of days with events is greater than 0.

Add Event		
state	request	assertions
Event to be added fields are correct	Request with event to be added fields as parameters	- Response status code is 200

Save Preferences		
state	request	assertions
Required preferences fields are correctly filled	Request with preferences as parameters	- Response status code is 200

Save Settings		
state	request	assertions
Required settings fields are correctly filled	Request with settings as parameters	- Response status code is 200

Change Password		
state	request	assertions
Old Password and New Password are set correctly	Request with passwords as parameters	- Response status code is 200 - User new password is effectively the new one set

6 REST API

This section describes the first version (v1) of the implemented REST API. The functionalities provided are the ones specified in the Design Document. Here with describe exactly how such specifications have been implemented and may be used as **usage documentation**.

6.1 User Management API

The following endpoints are used to manage User related information.
Apart for the **Registration** and **Login** endpoints all the others are privileged endpoints and require authentication.

Registration/Create new User

endpoint	api/{api version}/user
method	POST
url params	
data params	mail: [alphanumeric] password : [alphanumeric] name: [text]
success response	Code: 200 Content : {"message": "User creation successful", "user": [User]}
error response	Code: 422 UNPROCESSABLE ENTRY Content : {"message": "The given data was invalid", errors: *details about the errors *} Code: 409 CONFLICT Content : {"message": "Data not correct , possible mail duplicate" }

function	Creates a new User account

Request example	POST /api/v1/user HTTP/1.1
	Host: addr:8080
	User-Agent: *
	Content-Type: application/json
	Accept: application/json
	
	{
	"name": "travlendar",
	"email": "travlendar@travlendar.com",
	"password": "travlendar"
	}

Response example	{ "message": "User creation successful",
	"user": {
	"name": "travlendar",
	"email": "travlendar@travlendar.com",
	"updated_at": "2018-01-02 14:07:29",
	"created_at": "2018-01-02 14:07:29",
	"id": 969
	}
	}

Login/Get authorization token

endpoint	*/oauth/token
method	POST
url params	
data params	client_id : [numeric] client_secret: [alphanumeric] grant_type: [text] username: [alphanumeric] password: [alphanumeric]

success response	Code: 200
	{
	"token_type": [text],
	"expires_in": [numeric],
	"access_token": [alphanumeric],
	"refresh_token": [alphanumeric]
	}

error response	Code: 401 UNAUTHORIZED Content : { "error": "invalid_client", "message": "Client authentication failed" } { "error": "invalid_credentials", "message": "wrong user credentials." }
function	Returns an authentication Token

Request example	POST /oauth/token HTTP/1.1
	Host: addr:8080
	User-Agent: *
	Content-Type: application/json
	Accept: application/json
	
	{
	"client_id" : 2,
	"client_secret": "6Y8LNZ3d8lopzBoTMBZ1...",
	"grant_type": "password",
	"username": "travlendar@travlendar.com",
	"password": "travlendar"
	}

Response example	{
	"token_type": "Bearer",
	"expires_in": 31536000,
	"access_token": "eyJ0eXAiOiJKV1QiLCJ...",
	"refresh_token": "def502000c7aa1035278..."
	}

Get User information

endpoint	api/{api version}/user
method	GET
url params	
data params	
success response	Code: 200 Content : {"message": "Request successful", "user": [User]}
error response	Code: 401 UNAUTHORIZED Content : { "message": "Unauthenticated" } Code: 500 INTERNAL SERVER ERROR Content : { "message": "Request failed" }
function	Returns the current User's information
Request example	GET /api/v1/user HTTP/1.1 Host: addr:8080 User-Agent: * Content-Type: application/json Accept: application/json Authorization: Bearer eyJ0eXAiOiJKV1QiLC...

Response example	{
	"message": "Request successful",
	"user": {
	"id": 969,
	"created_at": "2018-01-02 14:07:29",
	"updated_at": "2018-01-02 14:07:29",
	"preferences": null,
	"email": "travlendar@travlendar.com",
	"name": "travlendar",
	"active": true
	}
	}

Update User

endpoint	api/{api version}/user/{id}
method	PUT
url params	
data params	name : (optional) [text] oldPassword : (optional) [alphanumeric] newPassword : (optional) [alphanumeric]
success response	Code: 200 Content : {"message": "User successfully modified", "user": [User]}
error response	Code: 401 UNAUTHORIZED Content : { "message": "Unauthenticated" } Code: 422 UNPROCESSABLE ENTRY Content : {"message": "The given data was invalid", errors: *details about the errors *} Code: 403 FORBIDDEN Content : { "message": "The password provided does not match the current one" } Content : { "message": "Provided User id and User token do not match" }\
function	Updates User's name and password information

Request example	PUT /api/v1/user/969 HTTP/1.1
	Host: addr:8080
	User-Agent: *
	Content-Type: application/json
	Accept: application/json
	Authorization: Bearer eyJ0eXAiOiJKV1QiLC...
	
	{
	"name": "test_account"
	}

Response example	{
	"message": "User successfully modified",
	"user": {
	"id": 969,
	"created_at": "2018-01-02 14:07:29",
	"updated_at": "2018-01-02 14:07:29",
	"preferences": null,
	"email": "travlendar@travlendar.com",
	"name": "test_account",
	"active": true
	}
	}

Update Preferences

endpoint	api/{api version}/preferences
method	PUT
url params	

data params	"transit" : {
	"active" : [boolean]
	"maxDistance": [integer]
	}
	"uber":{
	"active" : [boolean]
	"maxDistance": [integer]
	}
	"walking":{
	"active" : [boolean]
	"maxDistance": [integer]
	}
	"driving":{
	"active" : [boolean]
	"maxDistance": [integer]
	}
	"cycling":{
	"active" : [boolean]
	"maxDistance": [integer]
	}

success response	Code: 200 Content : {"message": "Preferences successfully modified", "user": [User]}
error response	Code: 401 UNAUTHORIZED Content : { "message": "Unauthenticated" } Code: 422 UNPROCESSABLE ENTRY Content : {"message": "The given data was invalid", errors: *details about the errors *}

function	Updates User's Preferences

Request example	PUT /api/v1/preferences HTTP/1.1
	Host: addr:8080
	User-Agent: *
	Content-Type: application/json
	Accept: application/json
	Authorization: Bearer eyJ0eXAiOiJKV1QiLC...
	
	{
	"transit" : {
	"active" : false ,
	"maxDistance": 10000
	},
	"uber":{
	"active" : false ,
	"maxDistance": 10000
	},
	"walking":{
	"active" : true ,
	"maxDistance": 10000
	},
	"driving":{
	"active" : false ,
	"maxDistance": 10000
	},
	"cycling":{
	"active" : false ,
	"maxDistance": 10000
	}
	}

Response example	{
	"message": "Preferences successfully ...",
	"user": {
	"id": 969,
	"created_at": "2018-01-02 14:07:29",
	"updated_at": "2018-01-02 14:07:29",
	"preferences": {
	"transit" : {
	"active" : false,
	"maxDistance": 10000
	},
	"uber":{
	"active" : false,
	"maxDistance": 10000
	},
	"walking":{
	"active" : true,
	"maxDistance": 10000
	},
	"driving":{
	"active" : false,
	"maxDistance": 10000
	},
	"cycling":{
	"active" : false,
	"maxDistance": 10000
	}
	},
	"email": "travlendar@travlendar.com",
	"name": "test_account",
	"active": true
	}
	}

Delete User

endpoint	api/{api version}/user
method	DELETE
url params	
data params	
success response	Code: 200 Content : {"message": "User deletion success"}
error response	Code: 401 UNAUTHORIZED Content : { "message": "Unauthenticated" } Code: 500 INTERNAL SERVER ERROR Content : { "message": "User deletion failed" }
function	Deletes the current User account
Request example	DELETE /api/v1/user HTTP/1.1 Host: addr:8080 User-Agent: * Content-Type: application/json Accept: application/json Authorization: Bearer eyJ0eXAiOiJKV1QiLC...

Response example	{
	"message": "User deletion successful",
	}

6.2 Schedule Management API

The following endpoints are used to manage Event related information.
These are all privileged endpoints and require authentication.

Get Schedule/Events

endpoint	api/{api version}/event
method	GET
url params	from : [integer] to : [integer]
data params	
success response	Code: 200 Content : {"message": "Request successful", "events": [Array<Event >]}
error response	Code: 401 UNAUTHORIZED Content : { "message": "Unauthenticated" } Code: 422 UNPROCESSABLE ENTRY Content : {"message": "The given data was invalid", errors: *details about the errors *}
function	Returns a list of Events within the specified time bounds
Request example	GET /api/v1/event?from=0&to=1514677000 HTTP/1.1 Host: addr:8080 User-Agent: * Content-Type: application/json Accept: application/json Authorization: Bearer eyJ0eXAiOiJKV1QiLC...

Response example	{
	"message": "Request successful",
	"events": [
	{
	"id": 14527,
	"userId": 969,
	"title": "Lesson",
	"start": 1514676600,
	"end": 1514677000,
	"category": "school",
	"description": "a brief description",
	"longitude": 45.478054,
	"latitude": 9.227298,
	"travel": null,
	"flexible_info": null,
	"repetitive_info": null
	}
	]
	}

Get Active Days/Days with Events

endpoint	api/{api version}/days
method	GET
url params	from : [integer] to : [integer] epoch: [boolean] timezone: [integer]
data params	
success response	Code: 200 Content : {"message": "Request successful", "days": [Array<Integer/String>], "timezone": [String]}
error response	Code: 401 UNAUTHORIZED Content : { "message": "Unauthenticated" } Code: 422 UNPROCESSABLE ENTRY Content : {"message": "The given data was invalid", errors: *details about the errors *} Code: 400 BAD REQUEST Content : {"message": "Invalid timezone "}
function	Returns a list of Events within the specified time bounds
Request example	GET /api/v1/days?from=0&to=1514677000 HTTP/1.1 Host: addr:8080 User-Agent: * Content-Type: application/json Accept: application/json Authorization: Bearer eyJ0eXAiOiJKV1QiLC...

Response example	{
	"message": "Request successful",
	"days": [
	1514674800
	],
	"timezone": "Europe/Paris"
	}

Create new Event

endpoint	api/{api version}/event
method	POST
url params	
data params	title: [alphanumeric] startTime : [integer] endTime: [integer] type: [alphanumeric] description: [text] travel: [boolean] mean: [text] (optional) travelDuration: [integer] (optional) distance: [integer] (optional) startLatitude: [numeric] (optional) startLongitude: [numeric] flexible: [boolean] (optional) duration: [integer] (optional) lowerBound: [integer] (optional) upperBound: [integer] repetitive: [boolean] (optional) until: [integer] (optional) frequency: [integer] adjustements : [vector]

success response	Code: 200 Content : {"message": "Event creation successful", "events": [Array<Event >]}
error response	Code: 401 UNAUTHORIZED Content : { "message": "Unauthenticated" } Code: 422 UNPROCESSABLE ENTRY Content : {"message": "The given data was invalid", errors: *details about the errors *} Code: 400 BAD REQUEST Content : {"message": "Feasibility check failed", "feasibility": *feasibility information *} Code: 500 INTERNAL SERVER ERROR Content : {"message": "Event creation failed "}
function	Creates a new Event

Request example	POST /api/v1/event HTTP/1.1
	Host: addr:8080
	User-Agent: *
	Content-Type: application/json
	Accept: application/json
	Authorization: Bearer eyJ0eXAiOiJKV1QiLC...
	
	{
	"title": "Lesson",
	"start": 1514676600,
	"end": 1514677000,
	"longitude": 45.478054,
	"latitude": 9.227298,
	"description": "a brief description",
	"category": "school",
	"repetitive": false,
	"flexible": false,
	"travel": false,
	"adjustements": []
	}

Response example	{ "message": "Event creation successful", "events": [{ "id": 14815, "userId": 969, "title": "Lesson", "start": 1514676600, "end": 1514677000, "category": "school", "description": "a brief description", "longitude": 45.478054, "latitude": 9.227298, "flexible_info": null, "travel": null }] }

Get Event

endpoint	api/{api version}/event/{id}
method	GET
url params	
data params	
success response	Code: 200 Content : {"message": "Request successful", "event": [Event]}
error response	Code: 401 UNAUTHORIZED Content : { "message": "Unauthenticated" } Code: 403 FORBIDDEN Content : {"message": "The requested resource does not belong to the current User or does not exists" }
function	Returns the requested Event
Request example	GET /api/v1/event/14815 HTTP/1.1 Host: addr:8080 User-Agent: * Content-Type: application/json Accept: application/json Authorization: Bearer eyJ0eXAiOiJKV1QiLC...

Response example	{
	"message": "Request successful",
	"event": {
	"id": 14527,
	"userId": 969,
	"title": "Lesson",
	"start": 1514676600,
	"end": 1514677000,
	"category": "school",
	"description": "a brief description",
	"longitude": 45.478054,
	"latitude": 9.227298,
	"travel": null,
	"flexible_info": null,
	"repetitive_info": null
	}
	}

Delete Event

endpoint	api/{api version}/event/{id}
method	
url params	
data params	repetitions: [boolean]
success response	Code: 200 Content : {"message": "Event deleted "}

error response	Code: 401 UNAUTHORIZED
	Content :
	`{ "message": "Unauthenticated" }`
	Code: 403 FORBIDDEN
	Content :
	`{"message": "The requested resource does not belong to the current User or does not exists" }`
	Code: 500 INTERNAL SERVER ERROR
	Content :
	`{"message": "Event deletion failed "}`

function	Returns the requested Event
Request example	DELETE /api/v1/event/14815 HTTP/1.1 Host: addr:8080 User-Agent: * Content-Type: application/json Accept: application/json Authorization: Bearer eyJ0eXAiOiJKV1QiLC...
Response example	`{"message": "Event deleted "}`

Generate valid Event

endpoint	api/{api version}/generator
method	POST
url params	
data params	title: [alphanumeric] startTime : [integer] endTime: [integer] type: [alphanumeric] description: [text] travel: [boolean] mean: [text] (optional) startLatitude: [numeric] (optional) startLongitude: [numeric] flexible: [boolean] (optional) duration: [integer] repetitive: [boolean] (optional) until: [integer] (optional) frequency: [integer]
success response	Code: 200 Content : {"message": "Request succesful", "event": [Event], "options": Array<Travel+adjustements> }
error response	Code: 401 UNAUTHORIZED Content : { "message": "Unauthenticated" } Code: 422 UNPROCESSABLE ENTRY Content : {"message": "The given data was invalid", errors: *details about the errors *}
function	Creates a new Event

Request example	POST /api/v1/generator HTTP/1.1
	Host: addr:8080
	User-Agent: *
	Content-Type: application/json
	Accept: application/json
	Authorization: Bearer eyJ0eXAiOiJKV1QiLC...
	
	{
	"title": "Lesson",
	"start": 1513980000,
	"end": 1513993600,
	"longitude": 11.244536,
	"latitude": 43.800797,
	"description": "first lesson",
	"category": "school",
	"travel": true,
	"flexible": true,
	"repetitive": false,
	"startLongitude": 11.230715,
	"startLatitude": 43.749687,
	"duration": 300
	}

Response example	{
	"message": "Request successful",
	"event": {
	"userId": 969,
	"title": "Lesson",
	"start": 1513980000,
	"end": 1513993600,
	"longitude": 11.244536,
	"latitude": 43.800797,
	"category": "school",
	"description": "first lesson",
	"id": -1,
	"flexibleInfo": {
	"eventId": -1,
	"lowerBound": 1513980000,
	"upperBound": 1513993600,
	"duration": 300
	}
	},
	"options": [
	{
	"travel": {
	"mean": "driving",
	"duration": 1662,
	"distance": 9569,
	"startLatitude": 43.749687,
	"startLongitude": 11.230715
	},
	"adjustements": {
	"-1": [
	1513993200,
	1513993600
	]}}]}

6.3 Booking Management API

The following endpoints are used to manage Booking information.
These are all privileged endpoints and require authentication.

Available Booking Options

endpoint	api/{api version}/available
method	GET
url params	event_id: [integer] start_latitude: [integer] start_longitude: [integer]
data params	
success response	Code: 200 Content : {"message": "Request successful", "available": Array<Bookings >}
error response	Code: 404 Content : { "message": "Error in Uber services: no available travels" "available": Array<> }
function	Returns the available booking options

Request example	GET /api/v1/available HTTP/1.1
	Host: addr:8080
	User-Agent: *
	Content-Type: application/json
	Accept: application/json
	Authorization: Bearer eyJ0eXAiOiJKV1QiLC...
	
	{
	"event_id": "500",
	"start_latitude": "45.485976",
	"start_longitude": "9.204145"
	}

Response example	{ "message": "Request successful",
	"available": [
	{
	"service": "uber",
	"bookingInfo": {
	"product_id": "d9af4271-d500-4dbf...",
	"request_id": null,
	"type": "Nearest UberBLACK",
	"duration": 900,
	"distance": 2752,
	"price_low": 13,
	"price_high": 13,
	"start_latitude": 45.485976,
	"start_longitude": 9.204145,
	"end_latitude": 45.476851,
	"end_longitude": 9.225882
	}
	},
	{
	"service": "uber",
	"bookingInfo": {
	"product_id": "78756b99-0f94-4541...",
	"request_id": null,
	"type": "UberLUX",
	"duration": 960,
	"distance": 2752,
	"price_low": 18,
	"price_high": 18,
	"start_latitude": 45.485976,
	"start_longitude": 9.204145,
	"end_latitude": 45.476851,
	"end_longitude": 9.225882
	}
	}
	]
	}

Create Booking

endpoint	api/{api version}/book
method	POST
url params	event_id: [integer] product_id: [alphanumeric] token: [alphanumeric] start_latitude: [floating] start_longitude: [floating]
data params	
success response	Code: 200 Content : {"message": "Request successful", "booking": [Booking] }
error response	Code: 400 Content : { "message": "Is not possible to book a ride after the start of the event" ␣ ␣"booking": ␣[] ␣ } Code: 400 Content : { "message": "Is not possible to book a ride until 60 minutes before the start of the event" ␣ ␣"booking": ␣[] ␣ } Code: 404 Content : { "message": "A problem occurred during the booking creation" ␣ ␣"booking": ␣[] ␣ }
function	Returns the current booking

Request example	POST /api/v1/book HTTP/1.1
	Host: addr:8080
	User-Agent: *
	Content-Type: application/json
	Accept: application/json
	Authorization: Bearer eyJ0eXAiOiJKV1QiLC...
	
	{
	"event_id": "500",
	"product_id": "78756b99-0f94-4541...",
	"token": "KA.eyJ2ZXJzaW9uIjoyLCJp...",
	"start_latitude": "45.485976",
	"start_longitude": "9.204145"
	}

Response example	{
	"message": "Request successful",
	"booking": {
	"service": "uber",
	"bookingInfo": {
	"product_id": "d9af4271-d500-4dbf...",
	"request_id": c1608441-9780-4851...,
	"type": "Nearest UberBLACK",
	"duration": 900,
	"distance": 2752,
	"price_low": 13,
	"price_high": 13,
	"start_latitude": 45.485976,
	"start_longitude": 9.204145,
	"end_latitude": 45.476851,
	"end_longitude": 9.225882 },
	"id": 427
	}
	}

Delete Booking

endpoint	api/{api version}/book
method	DELETE
url params	token: [alphanumeric] booking_id: [integer]
data params	

success response	Code: 200
	Content :
	{
	"message": "Booking deletion successful"
	}

error response	Code: 403
	Content :
	{
	"message": "User has no active booking"
	}
	
	Code: 404
	Content :
	{
	"message": "Booking deletion failed"
	}

function	Delete the current booking from the DB and the service

Request example	DELETE /api/v1/book HTTP/1.1
	Host: addr:8080
	User-Agent: *
	Content-Type: application/json
	Accept: application/json
	Authorization: Bearer eyJ0eXAiOiJKV1QiLC...
	
	{
	"token": "KA.eyJ2ZXJzaW9uIjoyLCJp...",
	}

Response example	{
	"message": "Booking deletion successful",
	}

Show Current Booking

endpoint	api/{api version}/book/current/booking
method	GET
url params	token: [alphanumeric]
data params	

success response	Code: 200
	Content :
	{
	"message": "Current ride available",
	"current": [Booking]
	}

error response	Code: 403
	Content :
	{
	"message": "User has no active booking"
	"current": []
	}
	
	Code: 403
	Content :
	{
	"message": "Current Uber booking not present in database"
	"current": [] }

function	Returns the current booking

Request example	GET /api/v1/book/current/booking HTTP/1.1
	Host: addr:8080
	User-Agent: *
	Content-Type: application/json
	Accept: application/json
	Authorization: Bearer eyJ0eXAiOiJKV1QiLC...
	
	{
	"token": "KA.eyJ2ZXJzaW9uIjoyLCJp..." ,
	}

Response example	{
	"message": "Request successful",
	"current": {
	"service": "uber",
	"bookingInfo": {
	"product_id": "d9af4271-d500-4dbf...",
	"request_id": c1608441-9780-4851...,
	"type": "Nearest UberBLACK",
	"duration": 900,
	"distance": 2752,
	"price_low": 13,
	"price_high": 13,
	"start_latitude": 45.485976,
	"start_longitude": 9.204145,
	"end_latitude": 45.476851,
	"end_longitude": 9.225882 },
	"id": 427
	}
	}

7 Effort Spent

We worked a lot together in order to take care of every relevant aspect that a multi-tier architecture requires. In details, each group memeber focused on the following aspects, working around 80-90 hours:

- **Gugliemo Menchetti**: worked mainly on the external API integration on the server side. In details, he implemented the interfaces for the following services:
  - Google Maps
  - Mapbox
  - Uber
  - Gmail

  He implemented the logic concerning the booking functionalities of the application. He performed the unit and integration tests for the interfaces and the booking API.
- **Lorenzo Norcini**: developed the back end part of the system. He defined the database schemas and he implemented the application logic for what concerns the menagment of the schedule, the feasibility manager and the design of the API endpoints. He also performed unit and integration tests for the components he developed.
- **Tommaso Scarlatti**: developed the iOS application, designing the user interface, the logo and the UX workflow. He managed the HTTP requests on the client side, taking into account possible exceptions due to the asynchronous response. He implemented the location search through the Apple MapKit framework and the in-app Uber login. He also wrote the system tests, testing the integration of the client application and the APIs provided by the server.

Furthermore, we worked together on the following aspects:

- Decide among the requirements defined in the RASD which one to implement in the prototype
- External services to be used
- Database structure details
- System integration
- ITD design