Software Engineering 2

Travlendar+

Design Document

Menchetti Guglielmo
Norcini Lorenzo
Scarlatti Tommaso

November 26, 2017

Contents

1 Introduction

1.1 Purpose

This document is aimed to provide an overview of the *Travlendar+* application, explaining how to satisfy the various project requirements stated in the RASD.

This document is mainly intended for developers and testers and its purpose is to provide a functional description of the main architectural components, their interfaces and their interactions, along with the design patterns and algorithms to be implemented.

1.2 Scope

Travlendar+ is a calendar based application whose goal is to help registered Users organize their day by scheduling their appointments and providing the best solutions in terms of mobility.

Users are able to manage their schedule, adding, deleting or modifying events. The system is in charge to check the feasibility of the schedule and to warn Users in case a conflict occurs. Furthermore, for each event, the application provides a list of available travel options, taking into account also Users preferences. Where possible, the system allows Users to book rides or to buy tickets for travel means relying on third party services.

Travlendar+ is structured in a multitier architecture. More specifically, the *Business logic* layer has the task of computing schedulability checks and interacts with external third-party services, through the use of interfaces, allowing Users to book rides or buy tickets. This layer is connected with the *Data* layer, in which are stored all the Users data (credentials, schedule, preferences). The *Presentation* layer, is build through the *thin Client* paradigm in which the client needs to perform close to no computation, allowing a more portable system.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- *Client:* is a piece of computer hardware or software that accesses a service made available by a server.
- *Server:* a computer program or a device that provides functionality for other programs or devices, called "clients".

- *Reverse Proxy:* proxies that forward requests to one or more ordinary servers which handle the request.
- *Firewall:* is a network security system that monitors and controls incoming and outgoing network traffic based on predetermined security rules.
- *Port:* is an endpoint of communication in an operating system.

1.3.2 Acronyms

DBMS	Data Base Management System
HTTP	Hyper Text Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
API	Application Program Interface
IEEE	Institute of Electrical and Electronics Engineers
OS	Operative System
UI	User Inteface
UX	User Experience
REST	REpresentational State Transfer
SPA	Single Page Application
MVC	Model View Controller
ORM	Object-Relational Mapping

1.3.3 Abbreviations

- [**R.n**]: n-th functional requirement in the RASD

1.4 Reference Documents

- RASD document
- Mandatory project assignment

1.5 Overview

The rest of the document is organized in this way:

- **Architectural Design:** shows the main components of the systems and their relationships. This section will also focus on design choices, styles, patterns and paradigms.
- **Algorithm Design:** presents and discuss in detail the algorithms designed for the system functionalities.

- **User Interface Design:** provides further details on the user interface defined in the RASD document through the use of UX modeling.
- **Requirements Traceability:** shows how the requirements in the RASD are satisfied by the design choices of the DD.
- **Implementation, Integration and Test plan**: shows the order in which the implementation and integration of subcomponents will occur and how the integration will be tested.

2 Architectural Design

2.1 Overview

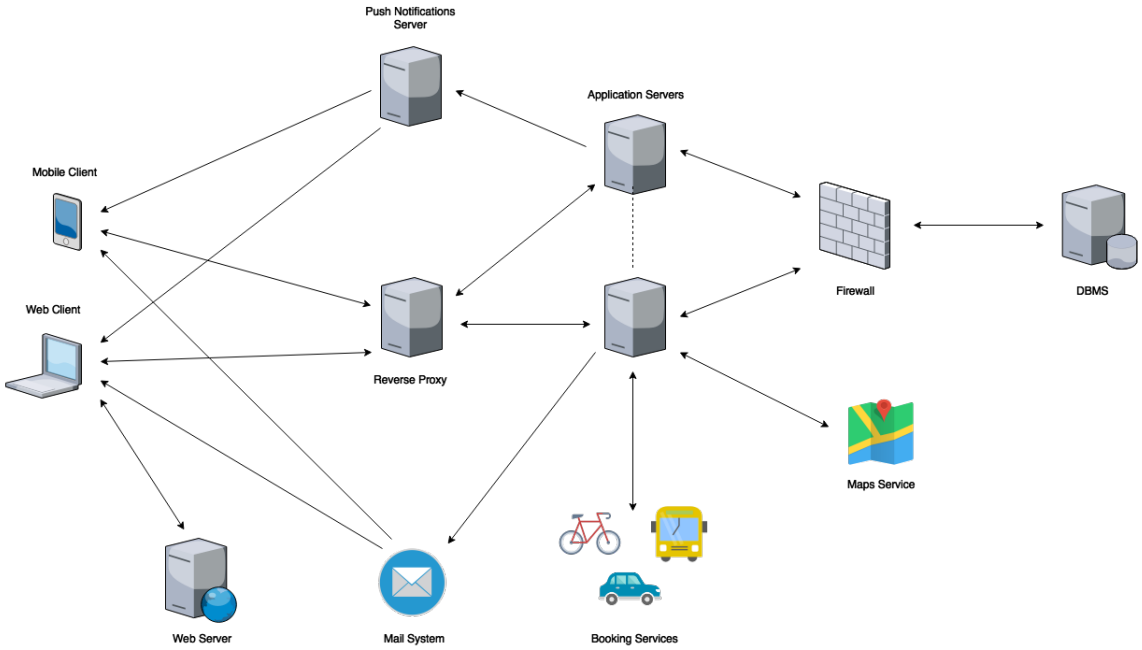

Figure 1: System overview

The above figure shows an high level overview of the System. Further details on the System components and their interactions will be explained in detail in the following sections.

2.2 Component View

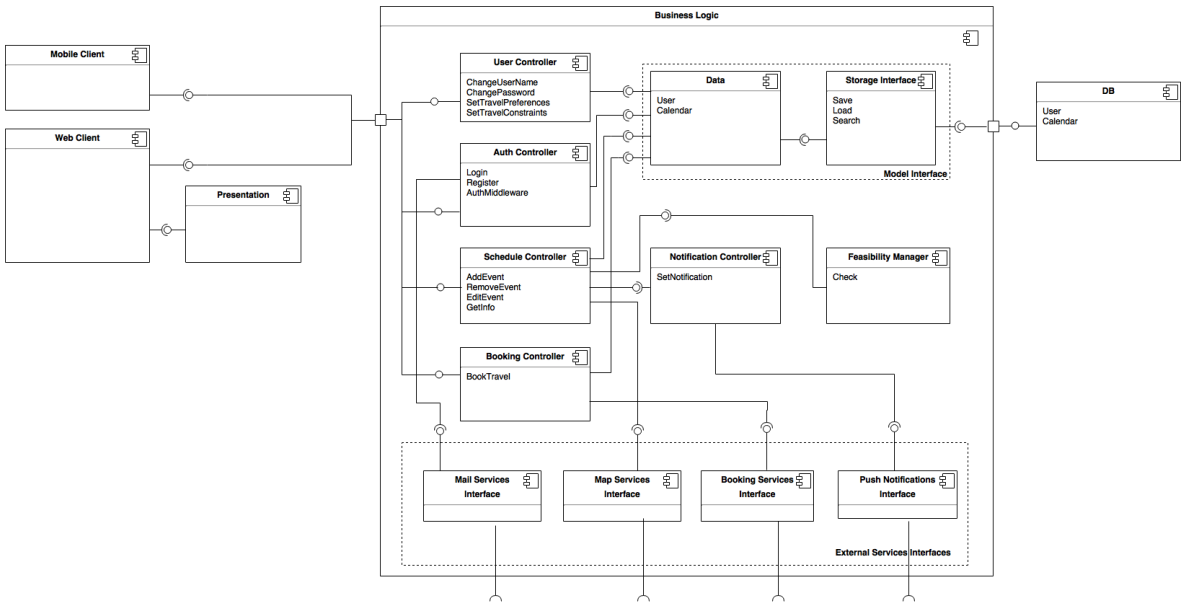

Figure 2: Component diagram

The UML component diagram aims at capturing the internal modular structure of components, showing how they are connected together in order to form larger components. Components are wired together by using an *assembly connector* to connect the required interface of one component with the provided interface of another component. Let's have a closer look at each component:

Mobile Client, Web Client

These two components represent the client machines that access to the API of the Business Logic container. They both do not have any notable functionality to be outlined, due to the fact that they are implemented as thin clients as explained below.

The Web Client, accessing trough a browser, needs the Presentation component in order to display the web pages of the application. This layer only provides the structure of the User Interface without accessing data and application logic.

User Controller

This component embeds all the operations that affects the user-related data. It exposes methods to change account credentials and to edit travel means preferences and constraints. Furthermore, it manages the data stored in the DB using the interface of the Model Interface.

Auth Controller

All the system authentication tasks are contained in this component. It's responsible both for the log in and the registration process and uses the Model Interface to retrieve and store data in the DB. It also uses the Mail Service Interface in order to communicate with the Mailing Service to send registration emails.

Schedule Controller

The Schedule Controller is the core component of the Business Logic container. Most of the business logic operations are grouped in this container. It exposes to the clients all the methods to manage their schedule and their events. It uses the Feasibility Manager to guarantee that the operations don't create any conflict in the schedule and the Model Interface to retrieve and store data in the DB.

Booking Controller

This component, using the Booking Services Interface, exposes methods to buy tickets or to book rides for public means, taxis or car and bike sharing. Again, this component uses the Model Interface to store Booking related information on the DB.

Notification Controller

The Notification Controller provides clients with the opportunity to be notified by the system in case of the occurrence of an upcoming event. To accomplish this result it uses the Push Notification Interface, forwarding notification requests to a Push Notification Server.

Feasibility Manager

This component is dedicated to check the consistency of the schedule, i.e.

detect the presence of conflicts. It is used by the Schedule Controller whenever a method tries to change the schedule adding, editing or deleting events.

External Services Interfaces

Each component of this group is tasked with calling the API of the related third party service. These interfaces are indispensable for the system: they make every other component available to interact in both directions with external services.
In particular the interfaces needed by the System are:

- *Booking Service Interface*: interacts with the third-party booking services allowing the user to book rides and buy tickets for a specific travel.
- *Mail Service Interface*: is responsible of the interaction with the mail service, used to send email confirmation to the User during the registration phase.
- *Map Service Interface*: communicates with the map service and is responsible of the information retrieval concerning travel time and available travel means.
- *Push Notification Interface*: interacts with the push notification service and is responsible of the notification for an incoming event to the User.

Model Interface

The Model Interface includes two sub-components. The Data component provides the set of Classes corresponding to the tables contained in the Database. The Storage Interface provides the methods for querying the Database.

Data Base

This component represent the DBMS, which provides the interfaces to retrieve and store data. In the data base, for each user, credentials and application data are safely and securely stored.

2.3 Deployment View

Here the deployment diagram of the whole system is shown. Its main aim is to specify the distribution of components capturing the topology of the

system's hardware.

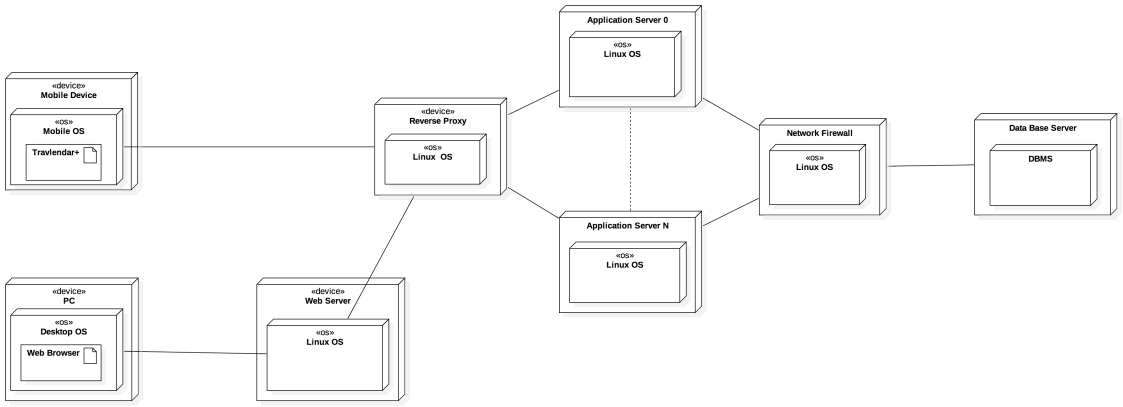

Figure 3: Deployment diagram

As we can see from the diagram, the system is structured in a multitier architecture. The specific role of the every node will be clarified below:

Clients

The first tier is composed by clients machines, which can be either mobile or desktop. In the first case, the client will be able to access *Travlendar+* functionalities through the dedicated native application, in the second case, by means of any web browser.

Web Server

A web server is used to store, process and deliver web pages to desktop clients. The communication between client and server takes place using the Hypertext Transfer Protocol (HTTP).

Reverse Proxy

We chose to deploy a reverse proxy on a Linux machine with Nginx server installed on it, in order to safely increase parallelism of requests and scalability of our application. This architecture leads to several advantages:

- Load balancing, dividing requests among different servers
- Can optimize content by compressing it in order to speed up loading times

- Event-driven architecture, increasing parallelism by not locking the CPU
- Very conservative memory-wise

Application Servers

This is the middleware level of the architecture: all the business logic of the system is contained in these application servers.

Network Firewall

The access to the Database is mediated by a network firewall in order to avoid unauthorized access to the data and the credentials of the user.

Data Base Server

This is the last layer of the architecture: all the data are stored in a Data Base Server equipped with a relational DBMS. Furthermore, credentials are not stored in plain text but hashed and salted: this method add a new layer of security to safeguard Users' credentials.

2.4 Runtime View

2.4.1 Registration Runtime View

The Guest has to register to the service before accessing the functionalities of the application.
The Guest fills a form containing the necessary information.
The information is serialized and then sent to the Application Server through an HTTP POST request.
The Authentication Controller handles the request, verifies if the information is correct and complete, then, through the model interface, creates a new entry in the User table on the Database.
At this point the User Account has been created but it has to be confirmed.
An email with a confirmation URL is sent by the external mailing service.
Once the Guest clicks on the provided link, the User entry on the Database is updated and the Account is set as active.

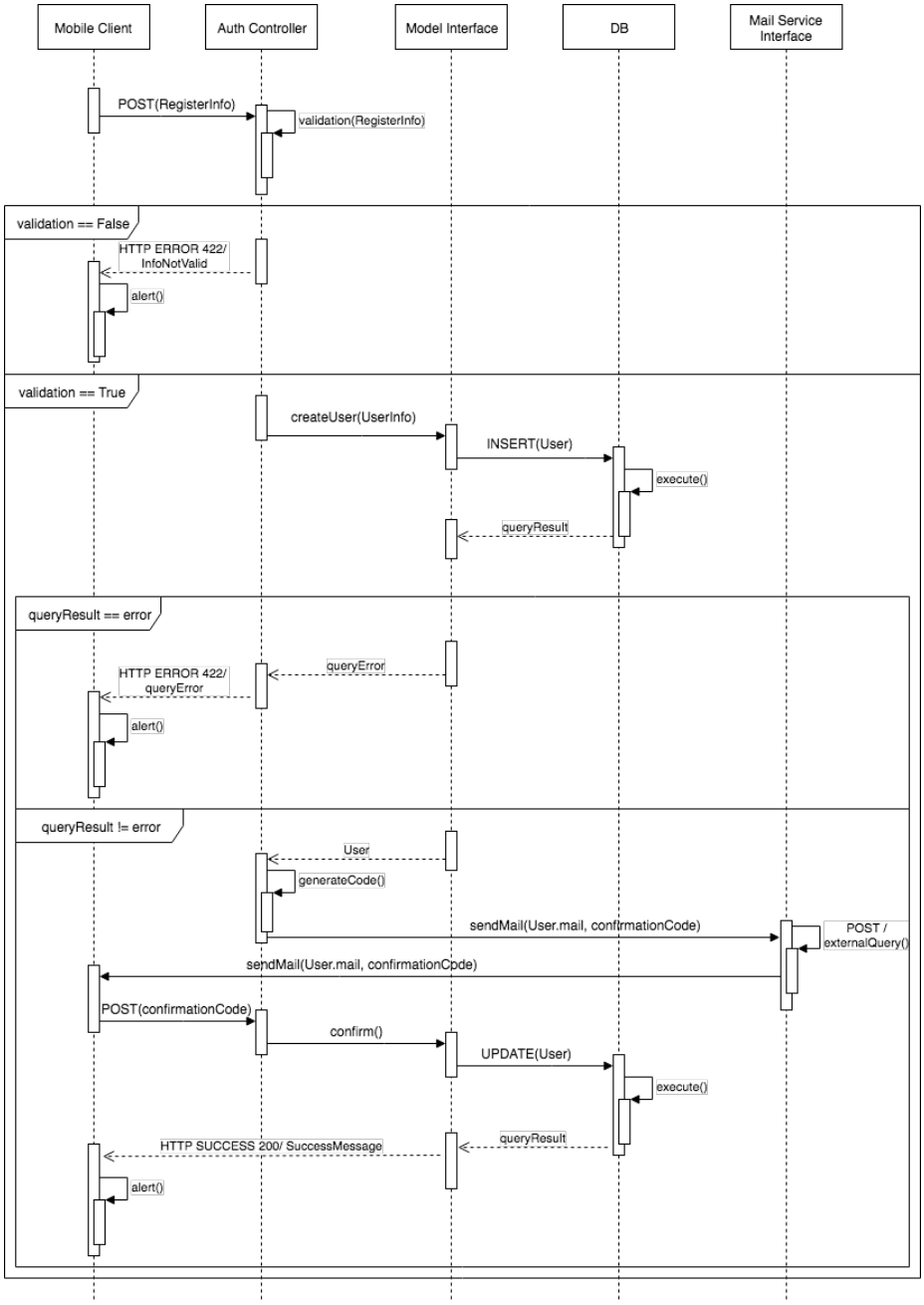

Figure 4: Registration Runtime View

2.4.2 Login Runtime View

The User has to login before accessing the functionalities of the application. The User submits the login information through an HTTP POST request. The request is handled by the Authentication Controller that validates the request and checks if the Database has an entry for the requested account. If the Account is present and the credentials are correct the Authentication Controller generates an Access Token, sets it as the current access Token for the specified Account and returns it to the Client.

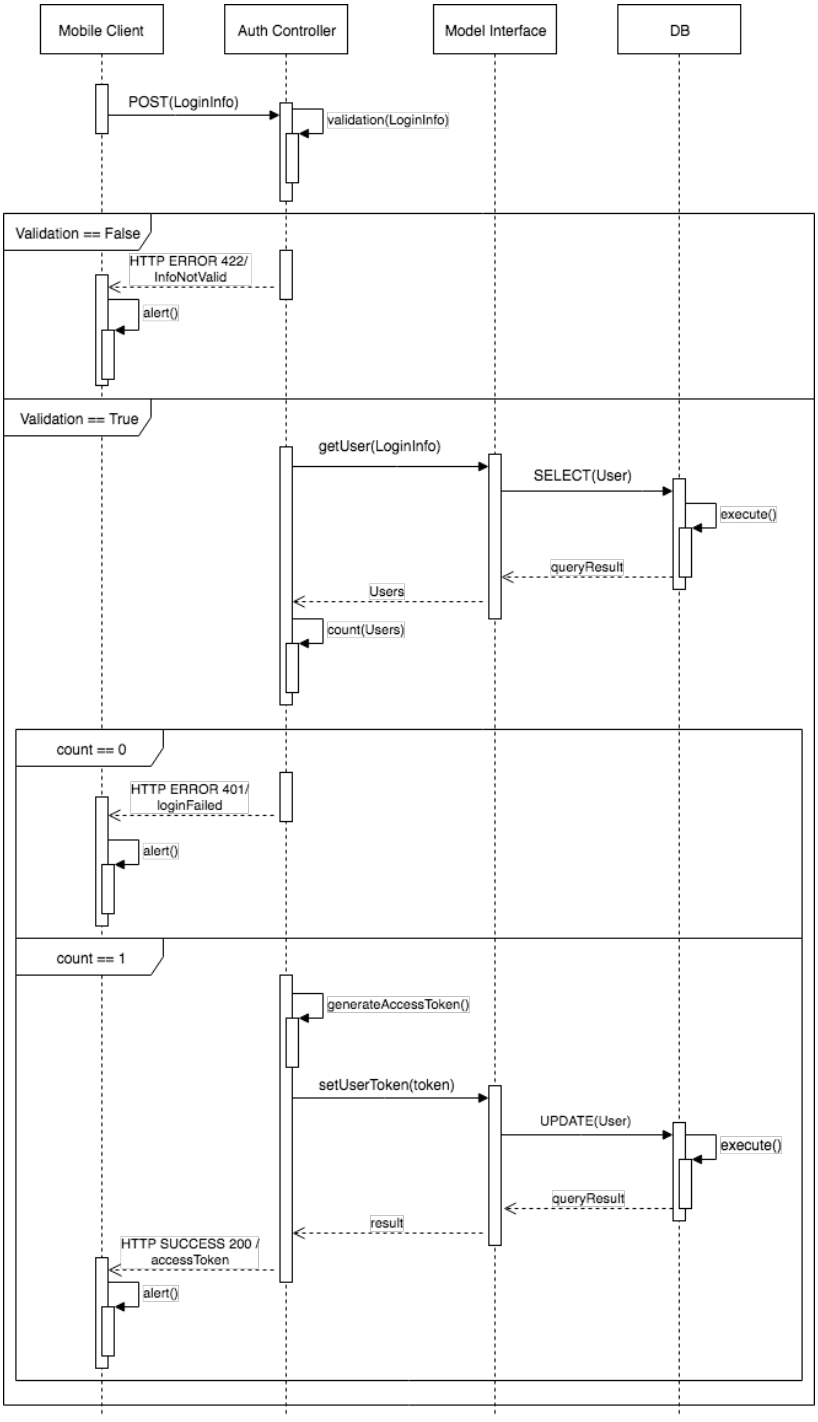

Figure 5: Login Runtime View

2.4.3 Add Event Runtime View

The User opens the event creation section of the application, fills the Event information and submits the request to create a new Event to the Schedule Controller along with the Event Info and the Access Token.
The Schedule Controller passes the Access Token to the Authentication Controller that checks if the Client provided a valid Token.
If the Authentication succeeds the control goes back to the Schedule Controller; the Schedule Controller passes the control to the Feasibility Schedule that checks if the new Event causes conflicts and if the Event is reachable.
If there are no conflicts and there are viable Travel options the User is prompted to choose one of the proposed options. Once User chooses, the Schedule Controller creates a Travel entry and an Event entry on the Database by means of the Model Interface.

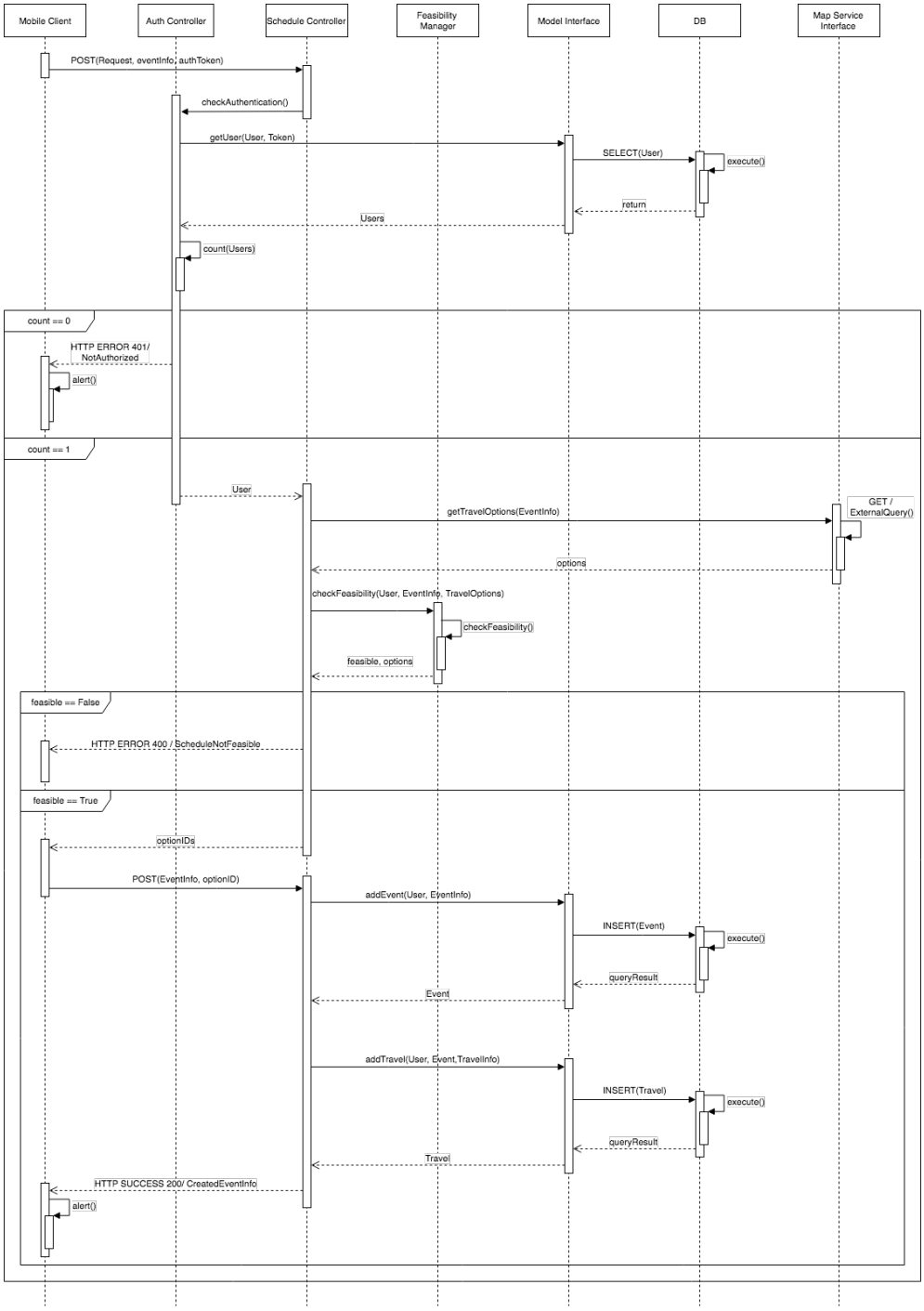

Figure 6: Add Event Runtime View

2.4.4 Book Travel Runtime View

The User requests a Booking on the Client application, the request is sent to the Booking Controller that checks with the Authentication Controller if the Client is authenticated.
If the Authentication succeeds the Booking Controller requests the Booking information for the requested Travel from an external booking service through the Booking Interface and returns such informations to the Client. The User is then prompted to confirm the purchase.
If the User confirms the purchase, the Booking Controller sends a confirmation to the External Booking Service and updates the Travel entry on the Database with the new Booking.
The whole booking process is also conditioned on the fact that the User provides credentials for the third party service.

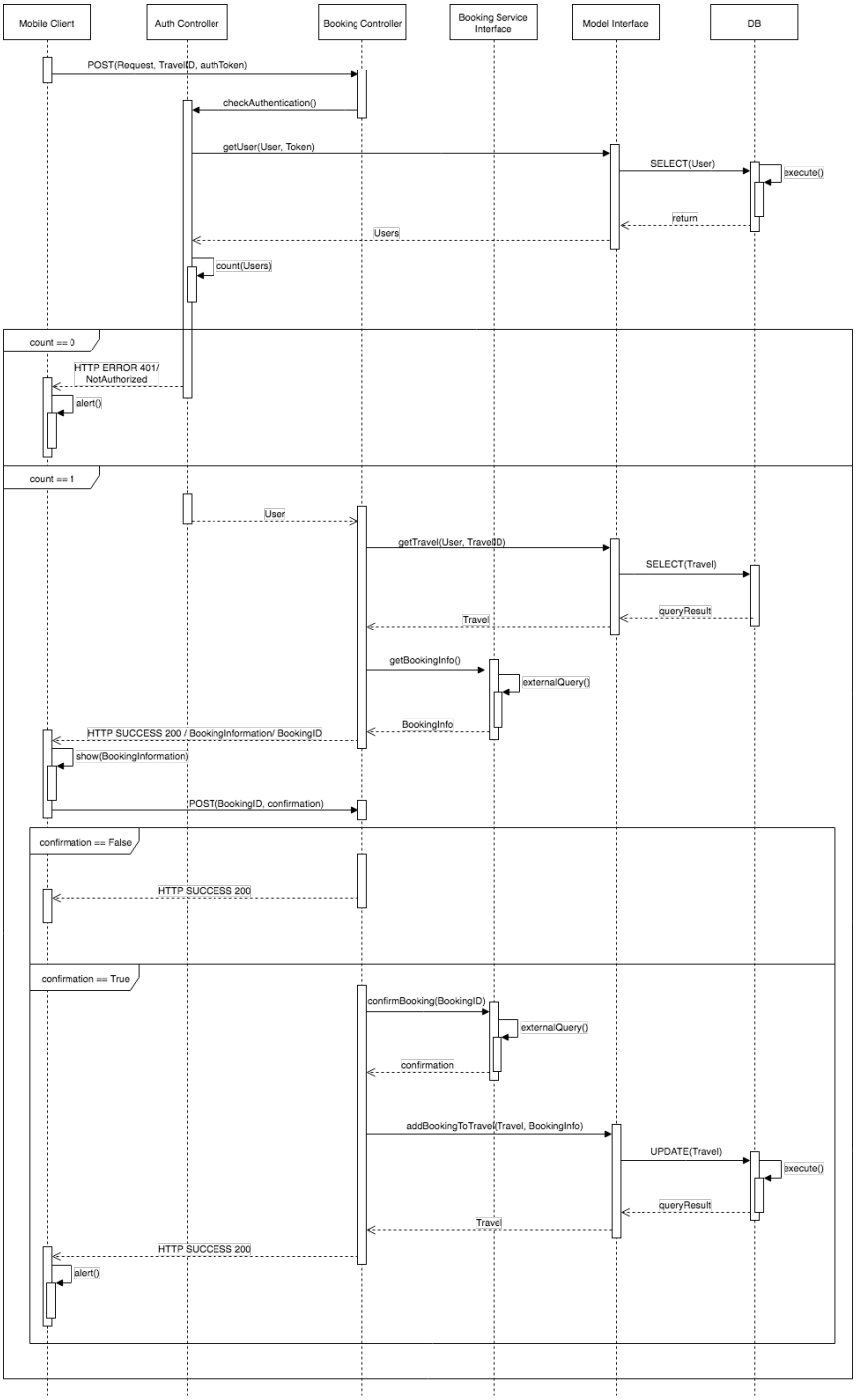

Figure 7: Book Travel Runtime View

2.4.5 Notification Runtime View

Once there is a change in the Calendar, the Schedule Controller notifies the Notification Controller of the changes occurred.
The Notification Controller gathers the new notifications information and configures the Push Notification Service.
The Push Notification Service will then notify the Client at specified times.

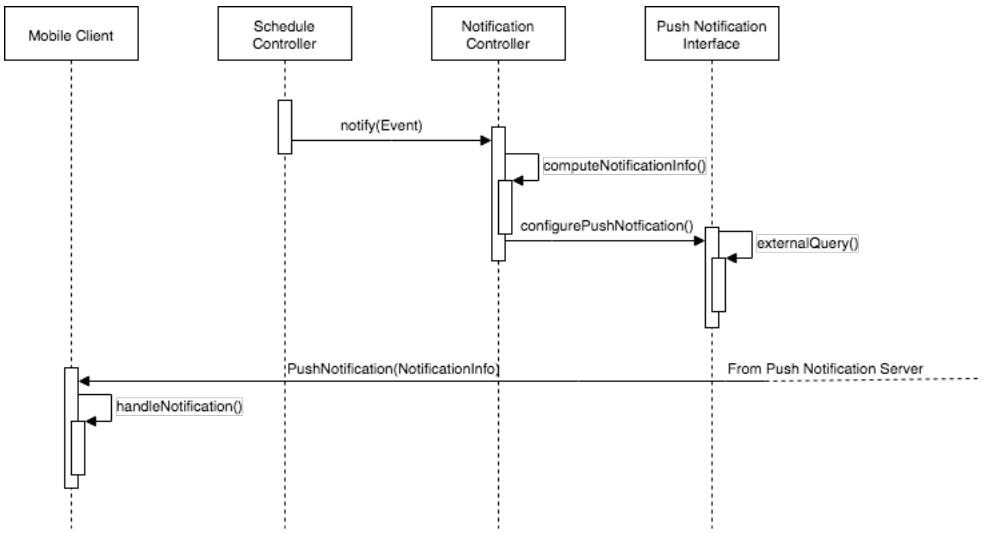

Figure 8: Notification Runtime View

2.4.6 Edit User Preference Runtime View

The User opens the User Information section of the application in editing mode, changes the preferences and submits the changes to the User Controller through a POST request. Assuming the request is authorized by the Authorization Controller, the User Controller checks whether the Preferences are satisfiable (i.e. are not impossible to respect) and using the Model Interface, updates the stored User Information.

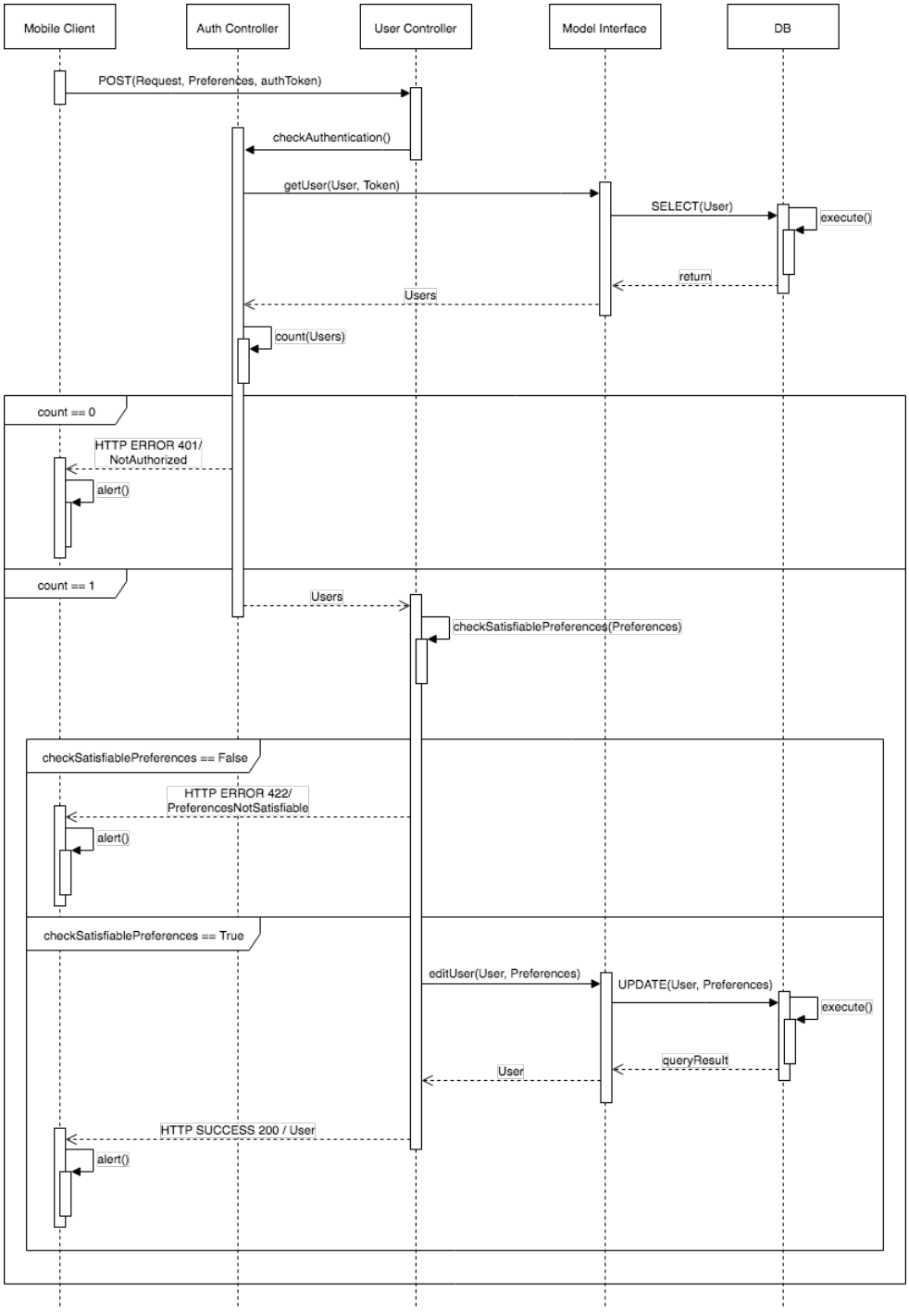

Figure 9: Set Preference Runtime View

2.5 Component Interfaces

2.5.1 REST API

Authentication Controller

The Authentication Controller provides methods for login and registration. This Controller also acts as a middleware between the clients' requests and other services that require authentication.
The Authentication Controller exposes the following methods:

Registration

endpoint	*/auth/register
method	POST
url params	
data params	mail: [alphanumeric] password : [alphanumeric] name: [text] surname: [text]
success response	code: 200 Content : {message: "Registration successful"}
error response	code: 422 UNPROCESSABLE ENTRY Content : {error: "Registration Data not correct"}
Notes	Allows a Client to request the registration of a new User

Login

endpoint	*/auth/login
method	POST
url params	
data params	mail: [alphanumeric] password : [alphanumeric]

success response	code: 200 Content : {accessToken: [alphanumeric]}
error response	Code: 422 UNPROCESSABLE ENTRY Content : {error: "Login Data not correct"} Code: 401 UNAUTHORIZED Content : {error: "wrong Mail or Password"}
Notes	Allows a Client to obtain an authentication Token

Activate Account

endpoint	*/auth/activate
method	GET
url params	activationCode: [alphanumeric]
data params	
success response	code: 200 Content : {message: "Account activated"}
error response	Code: 404 NOT FOUND Content : {error: "Incorrect activation code"}
Notes	Allows a Client to activate the account

Schedule Controller

The Schedule Controller handles all Schedule and Events related operations such as add, remove and edit. The use of the exposed methods is conditional on the User authentication which is handled by the Authentication Controller.

Add Event

endpoint	*/schedule
method	POST

url params	
data params	title: [alphanumeric] startTime : [Date] endTime: [Date] (optional) duration: [integer] (optional) frequency: [integer] travel: [Travel] type: [alphanumeric] description: [text]
success response	Code: 200 Content : {message: "Event created"}
error response	Code: 401 UNAUTHORIZED Content : {error: "User not logged"}
Notes	Allows a Client to create a new Event in the Schedule associated with a specific Account

Compute Travel Options

endpoint	*/schedule
method	POST
url params	
data params	startTime : [Date] endTime: [Date] (optional) duration: [integer] (optional) frequecty: [integer] type: [alphanumeric]
success response	Code: 200 Content : {travel: Array<Travel>}

error response	Code: 401 UNAUTHORIZED Content : {error: "User not logged"}
Notes	Allows a Client to obtain information related to the available Travel means

Remove Event

endpoint	*/schedule/{id}
method	DELETE
url params	accessToken: [alphanumeric]
data params	
success response	Code: 200 Content : {message: "Event removed"}
error response	Code: 404 NOT FOUND Content : {error: "Event not found"} Code: 401 UNAUTHORIZED Content : {error: "User not logged"}
Notes	Allows the Client to remove an Event from the Schedule associated with a specific Account

Edit Event

endpoint	*/schedule/{id}
method	PUT
url params	

data params	accessToken: [alphanumeric] (optional) title: [alphanumeric] (optional) startTime : [Date] (optional) endTime: [Date] (optional) duration: [integer] (optional) frequecty: [integer] (optional) type: [alphanumeric] (optional) description: [text]
success response	Code: 200 Content : {message: "Event edited"}
error response	Code: 404 NOT FOUND Content : {error: "Event not found"} Code: 401 UNAUTHORIZED Content : {error: "User not logged"}
Notes	Allows the Client to edit an Event from the Schedule associated with a specific Account

Get Schedule

endpoint	*/schedule
method	GET
url params	accessToken: [alphanumeric] startTime : [Date] endTime : [Date]
data params	
success response	Code: 200 Content : {events: Array<Event>}

error response	Code: 404 NOT FOUND Content : {error: "Travel not found"} Code: 401 UNAUTHORIZED Content : {error: "User not logged"}
Notes	Allows a Client to obtain a List of the events in the Schedule associated with an Account

Booking Controller

The Booking Controller provides the necessary methods for retrieving informations about available tickets to buy or rides to book from third party services (e.g. public transportation and taxis).
The use of such methods is conditional on the User authentication which is handled by the Authentication Controller, and if necessary, on the the authentication with the third party service.

Request Booking Info

endpoint	*/booking/info
method	GET
url params	travelID: [integer] (optional) bookingServiceToken: [alphanumeric]
data params	
success response	Code: 200 Content : {booking: Array<Booking>}
error response	Code: 401 UNAUTHORIZED Content : {error: "User not logged"}
Notes	Allows a Client to obtain informations about Booking options for a specific Travel

Confirm Booking

endpoint	*/booking/confirm
method	POST
url params	
data params	bookingID: [alphanumeric] bookingServiceToken: [alphanumeric]
success response	Code: 200 Content : {booking: Booking}
error response	Code: 401 UNAUTHORIZED Content : {error: "User not logged"} Code: 403 FORBIDDEN Content : {error: "User not authenticated with the external service"}
Notes	Allows the Client to confirm the purchase of a specific Booking

User Controller

The User Controller provides the necessary methods to obtain and update User related information such as Authentication credentials and Preferences.

Edit User Information

endpoint	*/user/{id}
method	PUT
url params	
data params	accessToken: [alphanumeric] (optional) newPassword: [alphanumeric] (optional) oldPassword: [alphanumeric] (optional) name: [text] (optional) surname: [text] (optional) preferences: [Array<Preference>]

success response	Code: 200 Content : {message: "Personal information edited}
error response	Code: 401 UNAUTHORIZED Content : {error: "User not logged"} Code: 403 FORBIDDEN Content : {error: "User ID provided does not match the authentication Token" || "Current password provided is incorrect} Code: 422 UNPROCESSABLE ENTRY Content : {error: "Preferences not satisfiable"} Code: 404 NOT FOUND Content : {error: "User not found"}
Notes	Allows a Client to edit User related information and set User's Preferences.

Get User Information

endpoint	*/user/{id}
method	GET
url params	accessToken: [alphanumeric]
data params	
success response	Code: 200 Content : {user: User}
error response	Code: 401 UNAUTHORIZED Content : {error: "User not logged"} Code: 403 FORBIDDEN Content : {error: "User ID provided does not match the authentication Token" || "Current password provided is incorrect} Code: 404 NOT FOUND Content : {error: "User not found"}

Notes	Allows a Client to obtain the informations related to the User associated with the provided Token.

2.6 Architectural Styles and Patterns

In this section the most relevant architectural design choices are shown, along with a satisfactory explanation of their role and their advantages.

Multitier architecture

A multitier architecture is a client–server architecture in which presentation, application processing, and data management functions are physically separated. The most common multi-tier architecture is the three-tier, composed by the following three layers: *Presentation tier, Domain logic tier, Data storage tier*. We added a fourth tier, the *Web tier*, in order to handle requests from web users. A multi-tier application architecture provides a model with several advantages: developers can create flexible and reusable applications and acquire the option of modifying or adding a specific layer, instead of reworking the entire application. Going deeper in details:

- *Presentation tier:* this is the topmost level of the application. The main function of the interface is to translate tasks and results in something that the user can understand. It's the only tier directly accessible by the user.

- *Web tier:* this layer is used only by the web clients in order to retrieve the client side code and static resources for the web application.
  In our case such application follows the Single Page Application (SPA) paradigm meaning that it interacts with the user by dynamically rewriting the current page rather than loading entire new pages from a server. This means that once the application is retrieved from the Web Server the Client will interact directly with the Domain/Business logic tier.

- *Domain/Business logic tier:* the logical tier is separated from the presentation tier and, as its own layer, it controls an application's functionality by performing detailed processing. In our architecture, it embeds both the application servers and the reverse proxy, which is used to handle client requests and to balance the workload.

- *Data storage tier:* this tier includes the data persistence mechanisms and the data access layer that encapsulates the persistence mechanisms and exposes the data.

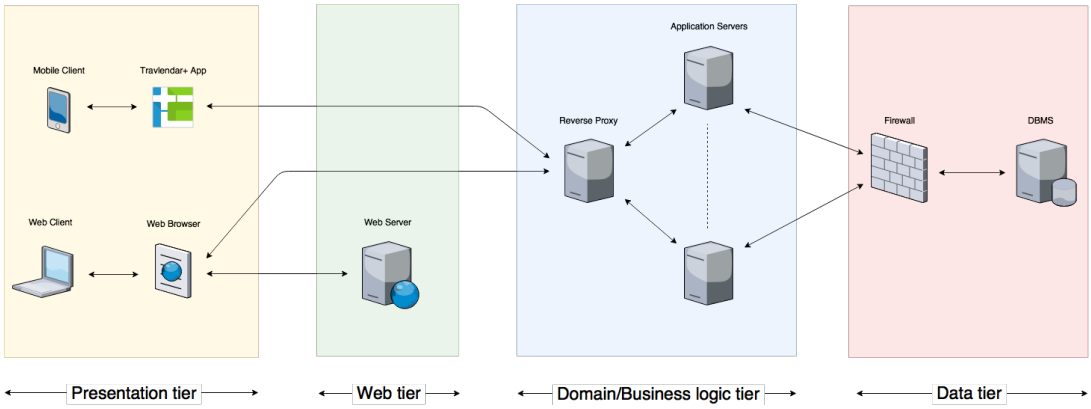

Figure 10: Multi-tier architecture of the system

Thin Client

The thin client paradigm is implemented with relation to the interaction between user's machine and the system. A thin client is a minimal sort of client, which needs to perform close to no computation, but only handling communications. The main benefits of this architecture is that it's easier to keep data synchronized across multiple clients and since there is less logic implemented on the client, new implementations in different client platforms require less effort.

	Relies on local store	Relies on local CPU
Fat Client	Yes	Yes
Hybrid Client	No	Yes
Thin Client	No	No

As we can see from the table above, a thin client does not rely neither on local storage, nor on local cpu. All the application logic is on the application servers, which have sufficient computing power and are able to manage concurrency issue efficiently. Furthermore, updates to the software are easier.

RESTful

The communication between the thin client and the application servers is set up according to the REpresentational State Transfer set of constraints. Data exchange is also made through the HTTP protocol, using SSL to provide encryption of sensible data which are encoded in the JSON format.
Complying with the REST constraints allows not only for simple component design, but also for intermediate layers such as firewalls or proxies. That is because the stateless nature of the system means that every request sent by the client contains all the necessary information for processing, not relying on server-side information. Furthermore, using a RESTful service allows to design a uniform interface, easily allowing for future scalability.

Relational DBMS

The Data layer consists of a relational DBMS. The relational model is a widely used design and there are several valid options, both open source and not, for what concerns the choice of the Management System.

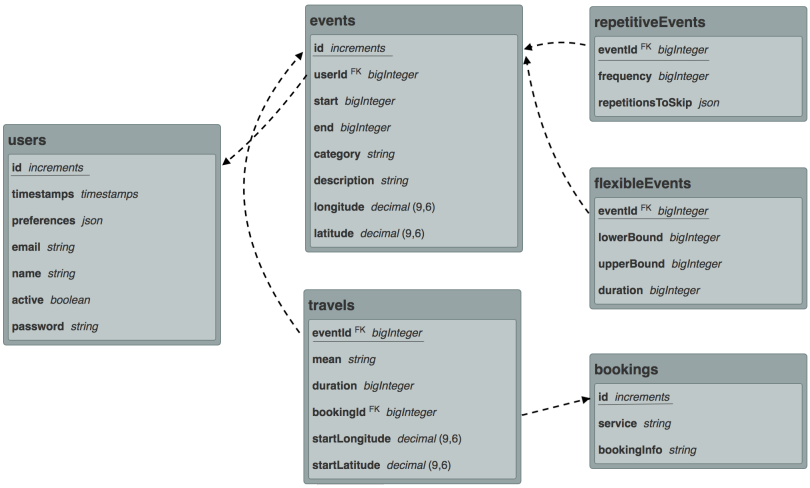

Figure 11: Database Schema Example

2.7 Other Design Decisions

Model View Controller

The frontend application is built following the Model-View-Controller (MVC) architectural pattern. By separating the application in three macrocomponents, MVC allows for full encapsulation, with all the related advantages:

each component can be changed without hassle, since they only need to present a coherent interface to the other ones, and it is easy to perform integration testing even if one or more components haven't been fully implemented. As has been said so far, MVC follows the *separation of concerns* principle.

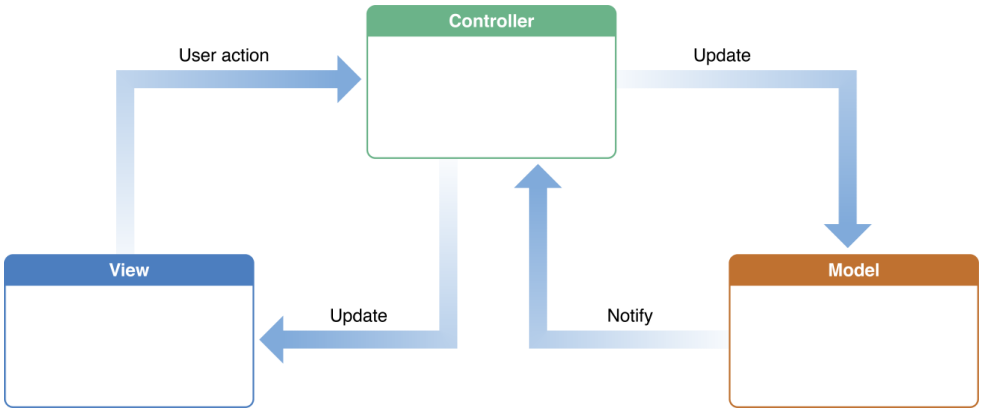

Figure 12: Model View Controller

Object-Relational Mapping

The Business Logic Layer interacts with the data stored in the Data Layer according to an Object-Relational Mapping paradigm.
ORM is a technique that allows to query and manipulate data from a database using an object-oriented paradigm.
This leads to more reusable and cleaner code adding a layer of abstraction over the Database Queries.

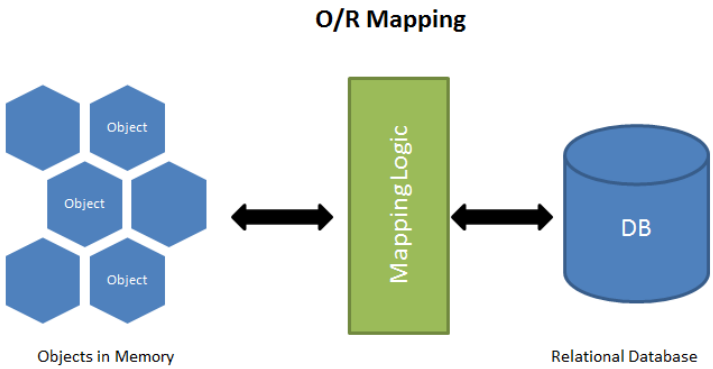

Figure 13: ORM Mapping

3 Algorithm Design

3.1 Chek feack feasibility for standard and flexible Events

Data:
schedule: User's schedule
newEvent: Event to be added to the schedule
travelOptions: Travel options available
Result:
feasible: *true* if the resulting schedule is feasible *false* otherwise
travelOptions: the Travel options that allow for a feasible schedule
changes: a dictionary that for each travel option contains the necessary changes for the schedule to be feasible.

```
changes ← Empty Dictionary
for option ∈ travelOptions do
    feasible ← true
    conflicts ← detectConflicts(newEvent, schedule, option)
    for event ∈ conflicts do
        optionChanges ← changes.forkey(option)
        feasible, newChange ← adjustTimes(event,newEvent,
         optionChanges)
        optionChanges.append(newChange)
        if ¬ feasible then
            travelOptions.remove(option)
            changes.removekey(option)
            break
        end
    end
end
if travelOptions.length > 0 then
    return True, travelOptions, changes
else
    return False
end
```

This algorithm describes how the feasibility is checked for standard and flexible Events.
Given a Schedule, an Event that has been added or modified and a list of possible Travel means, the algorithm detects for each option a list of conflicts, if any of the events causing the conflict is flexible the algorithm tries to adjust the start and the end of such events in order to remove the conflict.

If none of the events are flexible or it's not possible to remove the conflict the Travel option is discarded.
At the end if there are still Travel options available the algorithm returns them along with the necessary changes to make the schedule feasible.

3.2 Check feasibility for repetitive Events

Data:
schedule: User's schedule
newEvent: Event to be added to the schedule
option: choosen Travel option
Result:
repetitionsToRemove: a list that contains the repetitions to remove for the schedule to be feasible.
changes: a list that contains the necessary changes for the schedule to be feasible.

```
changes ← Empty List
repetitionsToRemove ← Empty List
for repEvent ∈ newEvent.repetitions do
    feasible ← true
    conflicts ← detectConflicts(repEvent, schedule, option)
    temporayChanges ← Empty List
    for event ∈ conflicts do
        feasible, newChange ← adjustTimes(event, repEvent,
          temporaryChanges)
        temporaryChanges.append(newChange)
        if ¬ feasible then
            repetitionsToRemove.append(repEvent)
            break
        end
        concat(changes, temporaryChanges)
    end
end
return repetitionsToRemove, changes
```

This algorithm describes how the feasibility is checked for repetitive Events. Given a Schedule, a repetitive Event that has been added or modified and a chosen Travel mean, the algorithm detects for each repetition a list of conflicts, if any of the events causing the conflict is flexible the algorithm tries to adjust the start and the end of such events in order to remove the conflict.

If none of the events are flexible or it's not possible to remove the conflict the repetition is discarded.
At the end the algorithm returns the repetitions to remove along with the necessary changes to make to the schedule feasible.

3.3 Adjust times for flexible Events

Data:
newEvent: Event to be added to the schedule
event: Event in conflict
changes: current Changes
Result:
feasible: *true* if the conflict is solved *false* otherwise
changes: changes to add to solve the conflict

```
change ← Empty List
e1, e2 ← applyChanges(newEvent, event, changes)
sTime ← e1.start - option.duration
eTime ← e1.end
if eTime > e2.startTime && eTime < e2.endTime && sTime >
 e2.start && sTime < e2.end then
 | return False
end
if e1.flexible then
 | if sTime > e2.start && sTime < e2.end then
 | | e1.start = tryToMoveSTART(e2.end + option.duration, e1)
 | | if sTime > e2.end then
 | | | return True, changes
 | | end
 | end
 | if eTime > e2.startTime && eTime < e2.endTime then
 | | e1.endtime = tryToMoveEND(e2.start , e1)
 | | if eTime < e2.start then
 | | | return True, changes
 | | end
 | end
end
if e2.flexible then
 | same process but modifying e2
end
return False
```

This algorithm corresponds to the *adjustTimes* subrouting called by the two previous algorithms.
This routine takes as input two conflicting events (plus some additional changes that may have been computed in advance) and if possible adjusts the times of the two in order to remove the conflicts.

4 User Interface Design

4.1 UX Diagram

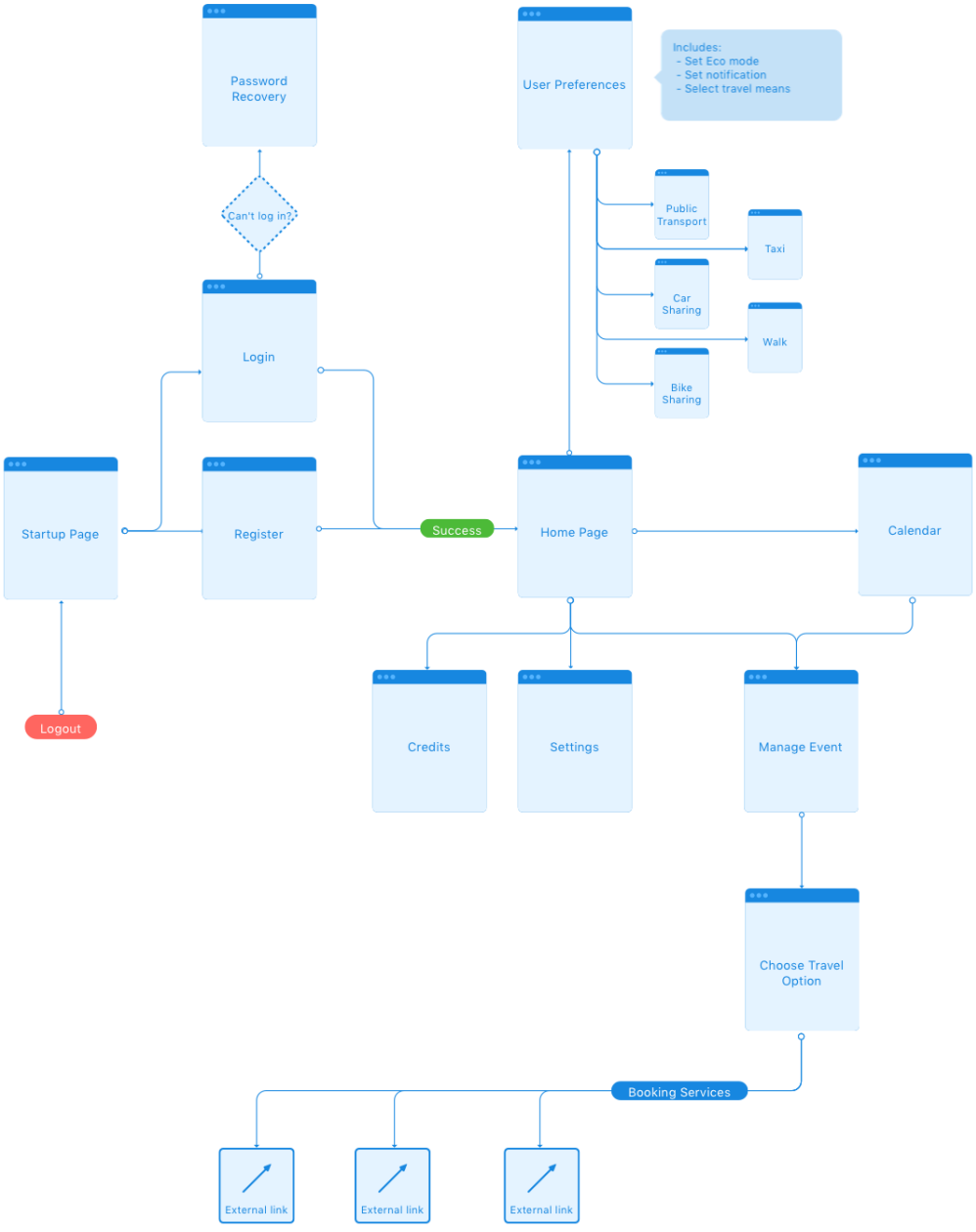

Figure 14: UX diagram

For the User Interface Design, we refer to section 3.1.1 of the RASD document. In this section, we enrich what was shown in the RASD by means of a User Experience diagram, which is principally thought to explain as clearly as possible the relationships between the various sections of the application.

As we can see from the diagram, the leftmost part is merely devoted to the necessary operations to log into the application:

- *Startup Page*
- *Login*
- *Register*
- *Password Recovery*

The main window of the application is the *Home Page*, which occupies a central position in the diagram. In this window user related information and daily schedule are displayed, any other section is reachable trough a sidebar. The sections arranged around the *Home Page* are:

- *User Preferences:* users are able to customize their travel experience.
- *Credits:* references to the creators of the application.
- *Settings:* mail, password and username can be edited.
- *Calendar:* displays a view of the user calendar per month. If a day has at least one event scheduled, a bullet is shown on the day cell.

As already outlined in the RASD, from the *Home Page* or the *Calendar* section is possible add, edit or delete an event in the dedicated *Manage Event* section. Once specified all the relevant information, the application will lead to the *Choose Travel Option* section, where a list of available travel means is provided together with links to external booking services to purchase tickets and rides.

5 Requirements Traceability

In this section we show how the components of the system architecture are involved in satisfying all the requirements outlined in the RASD document.

Component (DD)	**Requirements (RASD)**
Auth Controller	**[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.4]** The System must allow the User to log in using his/her personal credentials.
User Controller	**[R.3]** The System must store all User credentials. **[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.
Mail Services Interface	**[R.6]** The System must send a confirmation email if username, email or password is changed.
Schedule Controller	**[R.7]** The User must be allowed to create events, specifying all the mandatory fields. **[R.8]** The User must be allowed to edit or delete a specific event in his/her schedule. **[R.9]** The System allows the User to provide optional event information: membership category and a brief description. **[R.10]** The System must allow the User to view all the events for a specific day. **[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.
Map Services Interface	**[R.13]** The System must compute travel time between appointments.

Feasibility Manager	**[R.11]** The System must check if the event created or edited by the User is feasible. **[R.14]** The System must guarantee a feasible schedule, that is, the User is able to move from an appointment to another in time. **[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.
Notification Controller	**[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.
Booking Controller	**[R.28]** The System must allow the User to select a specific travel means.
Booking Services	**[R.29]** The System must provide an interface for third party services allowing the User to authenticate with the service.

6 Implementation, Integration and Test Plan

As previously illustrated, the System consists of many components that can be divided into the following categories:

- **Frontend components**: Client application
- **Backend components**: user controller, authorization controller, schedule controller, booking controller, notification controller, feasibility manager, model interface, mail service interface, map service interface, booking service interface and push notification interface
- **External components**: booking, map, mail and notification components provided by third-party services and the DBMS

In order to implement, integrate and test the System, a *bottom-up strategy* will be used. In particular, will be first implemented, integrated and tested the components of the same subsystem, then the subsystems will be integrated and tested together in order to verify that the System behaves according to specifications.
It has to be noticed that the components of the external subsystem don't need to be tested because the external services are considered reliable. Then, they will be used in place of stubs in order to test the interfaces.
Hence, the implementation and integration process will be performed in two levels:

1. Implementation and integration of different components of the same subsystem
2. Integration of different subsystems

For what concerns the first step, component integration of the same subsystem will be applied only for the backend subsystem. More specifically, the component to be integrated and tested together in this category are:

- User controller and model interface, forming the *User subsystem*
- Authorization controller, model interface and mail service interface, forming the *Auth subsystem*
- Notification controller, push notification interface and schedule controller, forming the *Notification subsystem*
- Schedule controller, model interface, feasibility manager and map service interface, forming the *Schedule subsystem*

- Booking controller, model interface and booking service interface, forming the *Booking subsystem*

For the second step the only subsystem integration that will be performed is between the backend and the frontend. The reason of this choice is due to the fact that the frontend and the external subsystems are independent and, as previously said, the external services will be used in place of stubs to test the interfaces but not tested by themselves.
Hence, the integrations between the backend subsystems and frontend subsystem that will be performed are:

- Client application and User subsystem
- Client application and Authorization subsystem
- Client application and Schedule subsystem
- Client application and Booking subsystem

After the application of this procedure, all the subsystems are integrated and tested together.

6.1 Sequence of Component Integration

The following diagrams describe the process of implementation, integration and testing. The arrow goes from the used component/subsystem to the component/subsystem that uses it.

Integration of the backend subsystem

All the components are first implemented and unit tested. Then some of the components are integrated and integration tests will be performed. The integrations are the following:

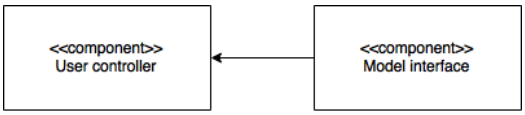

Figure 15: User subsystem

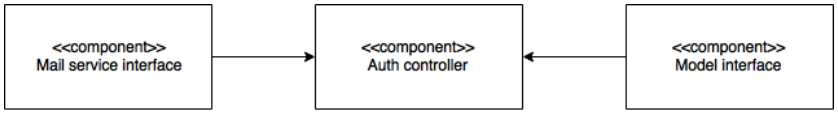

Figure 16: Authentication subsystem

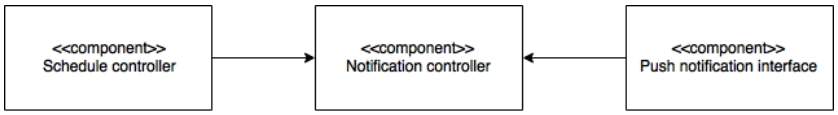

Figure 17: Notification subsystem

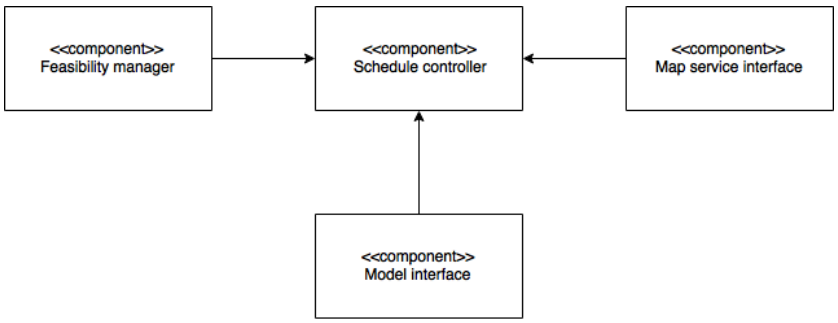

Figure 18: Schedule subsystem

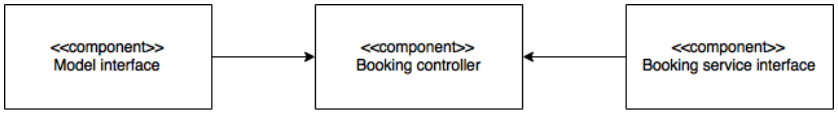

Figure 19: Booking subsystem

Integration of backend and frontend

Once all the components of the backend are fully implemented and tested, the frontend will be integrated and tested with the backend. In particular, the subsystems to be integrated with the *Client subsystem* are the *User subsystem, Auth subsystem, Schedule subsystem* and *Booking subsystem.*

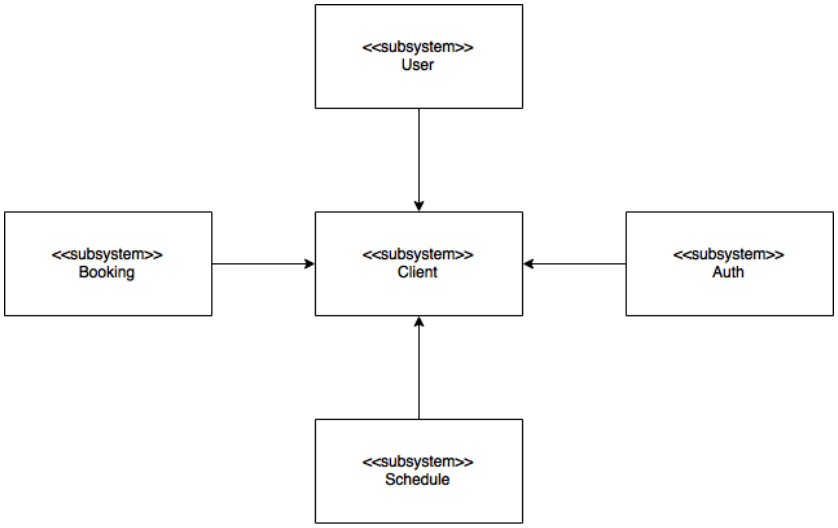

Figure 20: Client subsystem integration

System integration

At the end of the procedure explained before, the *Frontend*, *Backend* and *External Subsystems* are integrated and tested together.

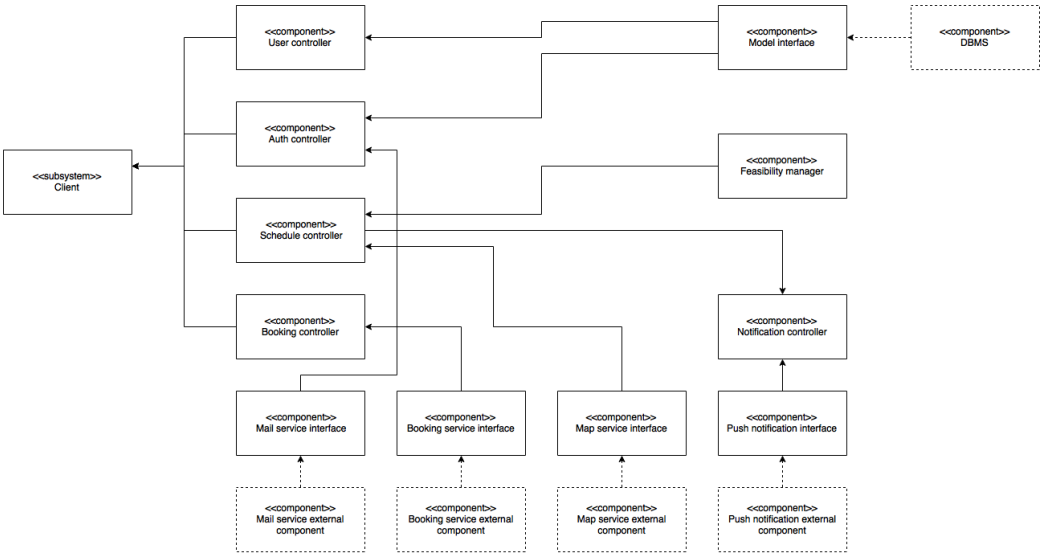

Figure 21: Subsystems integration

7 Effort Spent

The effort spent from each member of the team to build the Design Document can be summarized as follow:

Guglielmo Menchetti

Date	**Task**	**Hours**
31/10/17	First meeting	2
06/11/17	Introduction	2,5
10/11/17	Overview	3
13/11/17	Scope and Definitions	2
16/11/17	Component View Review	1
18/11/17	Runtime Views	3,5
20/11/17	Component Interfaces	1,5
22/11/17	Integration and Test Plan	4
23/11/17	Architectural Syles and Patterns	0,5
24/11/17	Integration and Test Plan Review	3,5
25/11/17	Final Review	2,5
		Total
		25,5

Lorenzo Norcini

Date	**Task**	**Hours**
31/10/17	First meeting	2
07/11/17	Overview	1,5
09/11/17	Component Diagram	2
10/11/17	Deployment Review	1
14/11/17	Runtime View	4,5
17/11/17	Component Interfaces	3,5
19/11/17	Algorithm Design	5
20/11/17	Architectural Syles and Patterns	2
21/11/17	Algorithm Design Review	2
22/11/17	Intro Review	1,5
25/11/17	Final Review	2,5
		Total
		27,5

Tommaso Scarlatti

Date	**Task**	**Hours**
31/10/17	First meeting	2
07/11/17	Overview and Deployment	4
12/11/17	UX Diagram	3,5
15/11/17	Component View	2
18/11/17	Architectural Styles and Patterns	3,5
19/11/17	Introduction Review	2
21/11/17	Requirements Traceability	3
22/11/17	Runtime View Review	1,5
23/11/17	Component Interface	2,5
24/11/17	Algorithm Design Review	2,5
25/11/17	Final Review	2,5
		Total
		27