Politecnico di Milano

Dipartimento di Elettronica, Informazione e Bioingegneria

Travlendar+

D.D.

Design Document

Alberto Floris

Claudio Montanari

Luca Napoletano

Contents

Chapter 1

Introduction

1.1 Purpose

This document represents the *Design Document* of the *Travlendar+* application. The main purpose of this document is to describe the application with respect to the design of the entire system, detailing the components of *Travlendar+*, the main interactions of various components and the physical architecture of the system.

1.2 Scope

Travlendar+ is aimed to be a 360 degree agenda for everyone's daily life. Firstly, the user will be able to define his appointments and a wide range of preferences. Preferences will include: type of transport, kind of appointment, eco-friendly profile and maximum walk range.

The System will be able to schedule the user appointments taking into account his preferences and making sure it will not be late. In addition, it will propose various mobility options considering time constraints (speed, traffic...), weather constraints (e.g. when it rains the user would avoid bikes trip), day constraints (strikes, festivity...).

Finally, the System will permit the user to buy public transport rides, rent or locate the nearest car or bike. This will be done considering potential discount and daily, weekly or season travel card.

Nowadays a system that integrates all these features doesn't exist, neither a supporting framework, this allow us to design the entire system from scratch.

1.3 Definitions, Acronyms, Abbreviations

The main definitions, acronyms and abbreviations used in the whole document are reported in this section of the *Design Document*.

1.3.1 Definitions

- **The Software:** when referring to *The Software*, this document refers to the entire *Travlendar+* infrastructure, at implementation and design level.
- **Application:** in the document is used like a synonymous of Software.

- **The Service:** when referring to *The Service*, this document refers to the service provided by the *Travlendar+* Software, at market level.
- **User:** the final person who use the *Travlendar+* software.
- **Registered User:** a user who has registered himself within the *Travlendar+* System.
- **Logged In User:** a registered user who is logged in the *Travlendar+* System. This kind of user will be one of the main actors interacting with the Software.
- **Time Slot:** period of continuous time when a task can be schedule.
- **Task:** an appointment the user has to do. A task always has associated a location, a time slot and a priority. Each task can have one or more of these behaviors:
  - **Fixed time:** a task with fixed time possesses a user specified time slot, that the System has to consider when producing the day's schedule.
  - **Flexible time:** a task with flexible time does not possess a fixed user specified time slot, instead it possesses a wider time frame so that the System could choose when to schedule the task within the given time frame constraint, which is user specified.
  - **Variable time:** a task with variable time does not possess a user specified time slot, so the System will instead schedule the task relating to the other time constraints.
  - **Fixed Day:** a task with fixed day possesses a user specified day within the week or the month, that the System has to consider when producing the day's schedule.
  - **Flexible Day:** a task with flexible day does not possess a fixed user specified day, instead it possesses a set of days within the week or the month so that the System could choose when to schedule the task within the given set, which is user specified.
  - **Variable Day:** a task with variable day does not possess a user specified day, so the System will instead schedule the task relating to the other time constraints within a user specified deadline.
  - **Fixed period:** the task repeat itself exactly after the given period.
  - **Flexible period:** the task repeat itself within the given flexible period, which is an interval of one or more days length.
  - **Not Periodic:** the task is not repeated.
- **Periodic task:** a task that possesses a fixed or flexible period.
- **Tasks Conflict:** given two user tasks, there will be a conflict if the two time slot associated to those tasks overlaps.
- **Task priority:** when referring to task priority we intend to refer to a user defined property which represent how much is important to follow that task constraints. The task priority can vary from 0 (lowest priority) to 5 (highest priority).
- **Location:** the place where the user either is or wants to go.

- **ZTL:** an area where some kind of private vehicles, such as cars or motorbikes, cannot enter. If the user has specific documentation, he can enter in the ZTL with his private vehicles.
- **Private Vehicle:** every kind of vehicle owned by the user, such as a car, a motorbike or a bike.
- **Shared based Vehicle:** vehicles that the user can book, and then rent, for a trip. To use these kind of vehicle the user needs a valid account of the vehicle sharing service he wants to use.
- **Public Transport Vehicle:** vehicles belonging to the public transport such as: bus, metro, tram and taxis.
- **Ticket:** a valid public transport document to use that service.
- **Break Time:** the time chosen by the user in which he doesn't want to have appointments.
- **User Preferences:** all the in-application preferences that user can choice. These preferences are:
  - Use of the private vehicle.
  - Use of the public transport services.
  - Use of either the Car-Sharing or the Bike-Sharing services.
  - Max range of walking.
  - Break time.
  - Eco profile.
  - Public transport allowed time slot.

  These kind of preferences can be used either as global preferences or as task preferences. In the former case they are treated for all the tasks, in the latter they are treated only for the task which has them. If a task has some preferences associated, then the system will consider instead of the global ones.
- **Travel Solution:** the proposed solution for the user that allows him to reach the destination in the way the user has specified through the preferences, which includes: travel indications for each travel mean involved, starting and arrival time.
- **Scheduling:** when referring to schedule we mean the sequence of user tasks for each day, and the set of travel solutions between them. These travel solutions will be suggested by the *Travlendar+* software, as described under the **Goals** section.

1.3.2 Acronyms

- R.A.S.D: Requirements Analysis and Specifications Document
- D.D: Design Document
- T.C.P: Transmission Control Protocol
- H.T.T.P.S: Hypertext Transfer Protocol Secure

- R.E.S.T: Represantional State Transfer
- J.S.O.N: JavaScript Object Notation
- D.B.M.S: DataBase Management System
- E.R: Entity/Relation Diagram
- U.X: User Experience
- D.M.Z: Demilitarized Zone

1.4 Document Structure

The *Design Document* is devoted to explain, with a good level of detail, the chosen architecture to develop easily the entire system. So, the document is divided into seven chapters, each one with the intent to explain a view on the *Travlendar+* System.

1. Introduction: chapter devoted to describe the general purpose of both the document and the *Travlendar+* application
2. Architectural Design: chapter devoted to show the architectural chosen for *Travlendar+*, the main physical components that will be used in the development phase. Moreover this section tries to explain the main modules used in each component and in each layer of the application.
3. Algorithm Design: chapter used to explain, in a general form, the main algorithmic part of the *Travlendar+* system, that is the explanation of the schedule algorithm used to construct the user's calendar.
4. User Interface Design: chapter devoted to explain the main user interfaces used in the client layer, to handle properly the user's input and to show information to him.
5. Requirements Traceability: section used to describe each application component with respect to the functional requirements which is satisfied by them.
6. Implementation, Integration & Test Plan: section used to describe the plan used when each component will be develop and how it will be integrated with the others components to simplify the testing.
7. Effort Spent: chapter devoted to explain how the *Travlendar+* team has worked, on which part of the document and how many hours each team component has spent in the creation of the document.

1.5 Revision History

This part of the *Design Document* contains the versions involved in the development of the final one.

Version 0.1:	Index created
Version 0.2:	First write of the Introduction Section
Version 0.3:	Added Component Diagrams, Deployment Diagram, Runtime View and Overview Pictures
Version 0.4:	First write of the Architectural Design Section
Version 0.5:	Added Mock-Ups on the document and writing of the *User Interface Design*
Version 0.6:	Added the *Algorithmic Design* part
Version 0.7:	Revision of the Algorithm and of the Deployment View, added the Requirement Traceability

1.6 References

- *Travlendar+* Requirements Analysis & Specifications Document, Version 1.2
- Specification documents: *Assignments AA 2017-2018.pdf*
- IEEE Standard for Information Technology - Systems Design - Software Design Description

Chapter 2

Architectural Design

2.1 Overview

Travlendar+ is an application that schedules the user's appointments, taking into account various tasks types, traffic information and so on. To do that as well as possible, the system is composed by four tiers, which are: the client layer, the web server layer, the application server layer and the data layer.

Travlendar+ uses these four layers to guarantee the best flexibility of the application, indeed the user can access to his calendar through either the web or the mobile application. Moreover, these four tiers permit to have a system that is easily maintained by developers.

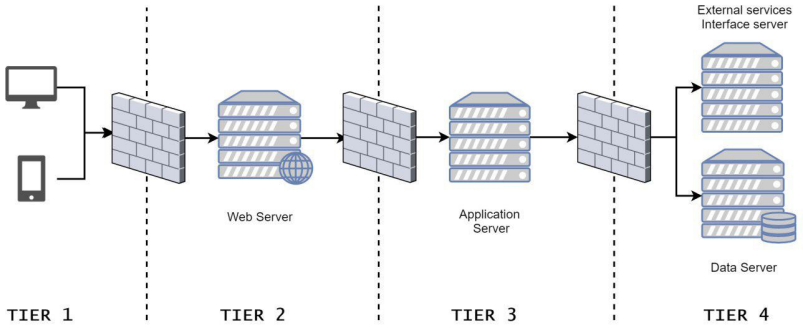

Figure 2.1: Physical architecture of the *Travlendar+* System

The application is divided into seven main components, the *web client* and the *mobile application client*, the *Application Façade*, the *Presenter Input* and the *Presenter Output*, the *Business Logic* and finally the *Data Layer*.

When a client wants to connect to the application, through a web page (or a mobile application interface) sends an https request, according to the REST protocol. This request is received by the *Travlendar+* web server, that contains the *Application Façade* component. The web server is used in order to create a secure zone for the application server, in this way several hacks can be avoided. The web server takes the HTTPS request, translate it in a socket request and then send it to the application server, through the *Presenter Input* component. This component is used to call the correct functionalities of the *Business Logic*, that is the core of the entire system. This layer is used to add a new user to the system, to control the login system, to create a calendar,

to add/modify/delete a task and to compute a travel. The *Business Logic* component indeed does the required operations on the user datas, then interacts with the *Data Layer* component to store these information or to receive some information either from the internal database or from an external service. For this reason the *Data Layer* component is used to interact with both the internal DBMS and the external APIs services.

Finally, once the user's request is satisfied, the *Business Logic* send the useful data to the *Application Façade* component through the internal socket connection, provided by the *Presenter Output* interface, and then the data are translated in a JSON file, and this is sent to the client.

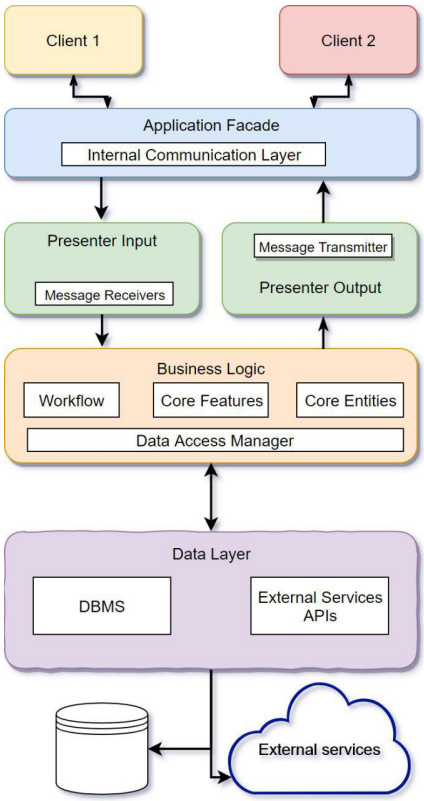

Figure 2.2: Overview of the *Travlendar+* System

2.2 Component View

This section details the internal modules used in each component seen previously. In particular, we focus on the main subcomponents of the business logic layer, as described in the previous chapter concerning the architecture overview. This because the main functionalities of our software are almost all exploited at this level, whereas the other layer's components are mainly designed in order to guarantee non functional features such as scalability or security.

At component level, the software can be divided betweeen *front-end components* and *back-end components*; as a five-layers architecture, the *Travlendar+* frontend is mainly represented by the client layer, which, as described in the *RASD* Document, will be implemented as both a mobile application and a web application, usable via browser.

2.2.1 Front-end system

In this subsection we mainly focus on the analysis of a possible generic client application, which can be implemented as a native application but also as a browser-oriented front-end. Nowadays it is relatively simple to develop *Javascript* based applications which uses web-oriented technologies without having to deal with common web browsers. this is the reason why is natural to describe the two client interfaces using a single component diagram: the overall structure is similar, and the main differences are implementation related and thus not discussed in this document.

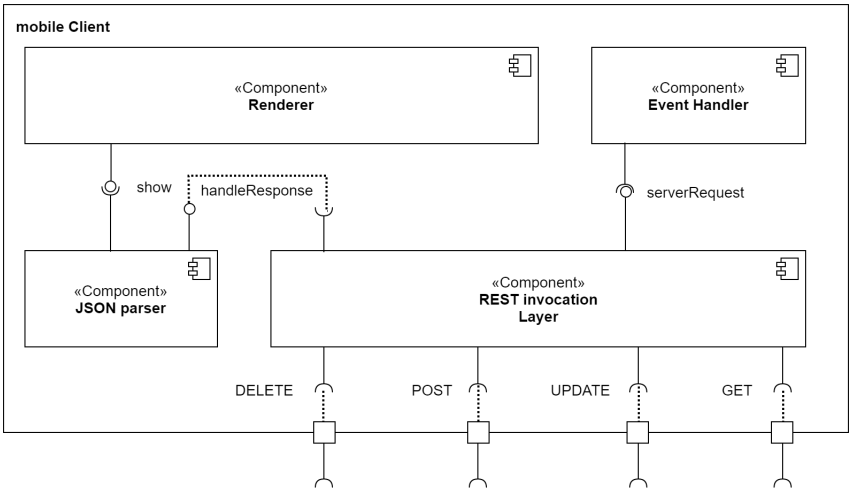

Figure 2.3: UML component diagram about front-end components

As seen in Figure 2.3, the main components are:

- **Renderer**: this component is meant to show data from the back end and display the interface from whom the user interacts with the application. in particular, the renderer is designed to be observed by other components, as in a traditional *observable - observer pattern*: this choice has been made in order to make the renderer as much indipendent as possible with respect to the overall application, and vice versa, thus permitting to completely rewrite both the renderer and the application without modifying the other.

- **JSON parser**: this component is meant to correctly interpret objects from server, so that the renderer can receive data without having to deal with the technologies used in other components. In particular, this component will receive JSON objects, it will interpret them and then call the correct functions from the renderer in order to populate the user interface.

- **Event handler**: this component will handle user imputs and call the network interface consequently. in particular, as seen when dealing about the renderer, the *observable - observer* approach will be used, so that the event handler *observates* the graphical objects representation inside the renderer, and response in a proper way when a certain event is thrown.

- **REST invocation layer**: this component will handle the communication with the back-end. because of the chosen client-server approach, the front-end will

mainly communicate with the back end using requests that are followed by proper responses. that is the reason why it is simple to design the communication between the client and the server using HTTPS and REST requests. the responses will then handled by the JSON parser, which will also update the graphic interface.

2.2.2 Business logic entities

In this subsection the core business logic entities will be presented. though they are not equally implemented between front-end and back-end, the high level design is similar. in this mean we can consider them proper of either the front end and the back end. the chosen design consider to include only data inside these entities, leaving almost any kind of operations outside them. as can be seen in Figure 2.4, the most complex entity is the *Task* entity, who is related directly or indirectly to every other.

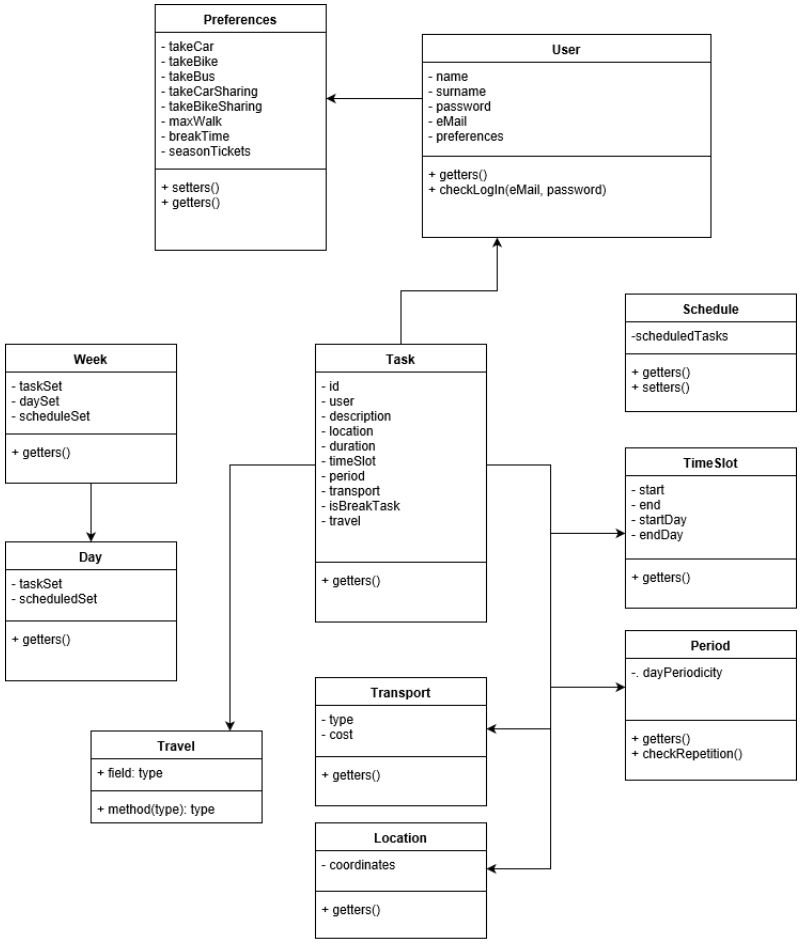

Figure 2.4: UML class diagram regarding business entities

2.2.3 Back-end system

As claimed previously, the server software architecture is fairly more complex than the client software. the main reasons about this unbalance are simple:

- the client hosts only the front-end, without operating any kind of business logic functionality.

- the back-end is built over three physical tiers, mainly for security and scalability reasons.

- the main idea between the back-end architectural design is to guarantee maintainability: all the software components will have to be as much modular and indipendent as possible, thus requires to design proper adapters between components that certainly increases the overrall complexity.

further information about this choice regarding the physical architecture will be presented in the chapter regarding the deployment view, leaving this section discuss only about how the software architecture is designed a-priori.

Application Façade is a macro component that handles client requests and submit them to the application logic in a proper way. this component is required because of the choice to rely on HTTPS when dealing about the communication between front-end and back-end: it is then necessary a *web server* component who handles rest request, and also an *internal communication layer* who deals about translating https requests in TCP messages to be sent to the business logic components. this approach is preferable instead of permitting to the front end to directly send request to the business logic in order to avoid misuses.

Presenter layer is the macro component which is in charge of receiving TCP messages from the application Façade and calling the business logic layer in order to satisfy the request. it is so mandatory to define a *communication interface* component that mantains an open socket connection with the application Façade. the other two components we decided to define are:

- **Message Receivers** are components which deals with receiving a message from the communication interface and calling the proper method inside the business logic. in this term, we can consider these components as a sort of middle ground between the presenter layer and the business logic layer. Further information about how these components properly works can be founded in the subsection regarding the *selected architectural styles and patterns*.

- **Message Transmitters** which are dual with respect to the message receivers: they observe the business logic and activate themselves when a certain operation is performed, thus transforming the result data in a proper message to be sent to the application Façade via the internal communication interface.

Business Logic layer is the main topic of this section. it can be divided in several different components which performs specific operations. here we present the main functionalities and the overall structure in terms of subcomponents, leaving the the subsection regarding the *selected architectural styles and patterns* to describes how these subcomponents are meant to be implemented.
the business logic layer is composed by:

- **Business WorkFlow**: this is the component which takes in charge external requests and coordinates the activity of other business logic components, so allowing the core features of *Travlendar+* to cooperate and produce a correct output. This means that the *business components* interfaces itself with all the main components

of our business logic architecture. nonetheless, he can also interface with the *Message Transmitter* previously described, and is called by the *message receivers* in order to properly handle messages from the frond-end.

- **Scheduler**: is the component in charge of schedule the user's task, which is the main core feature of the *Travlendar+* software. It can be divided in other three subcomponents:
  - *Week Scheduler*: this is the component that deals about dividing the weekly task who receives in input into the corresponding day, in order to be scheduled. the week scheduler takes care of both variable and flexible day tasks, where day fixed tasks need little computation. to accomplish this goal we decide to use a stochastic approach, based on simple travel time estimations, leaving the heavy part of the scheduling algorithm to the day scheduler.
  - *Day Scheduler*: this component is necessary to organize tasks between a single given day. it uses a particular version of the *branch and bound* algorithm in order to return the optimal scheduling between the given day. it also relies on the *Cost Evaluator* component in order to calculate the cost, in terms of money and time, of each solution.
  - *Cost Evaluator*: this component takes care of calculating the cost of each travel solution. in order to do so, it needs to communicate with the *data access manager* within the business logic, and so retrieve information such as traffic or route options from the external services which the application relies on.

  Further description about how the scheduler works can be found in *Chapter 3*, which describes how the scheduling algorithm works.

- **Data Access Manager**: this is the component in charge of extracting data from either external services and the *Travlendar+* database. note that, in order to be as modular as possible, this component does not directly interact with these data source, but relies on other minor components at data level. thus, his main goal is to build an high level representation of these data without having the other components to work about how to interpret. In this way we can claim this component as an *adapter* between the business logic and the data layer.

- **General Preferences Manager**: this component main goal is to manage the user preferences, as described in the *RASD* Document, and then check whether certain solutions are admissible or not. thus, it needs to communicate both with the data access manager and the *Business WorkFlow*, in order to interpret "complex" preferences such as an *eco friendly* profile in much more simple instructions about how to build the schedule.

- **Task Fetcher**: as the other component, the main goal of the *task fetcher* is either to create an high level representation of data from the data layer. in particular, the task fetcher needs to build the task entities using data from the access manager, from those extracting only the ones who are relevant to the scheduling, modifying them when needed and performing other similar operations.

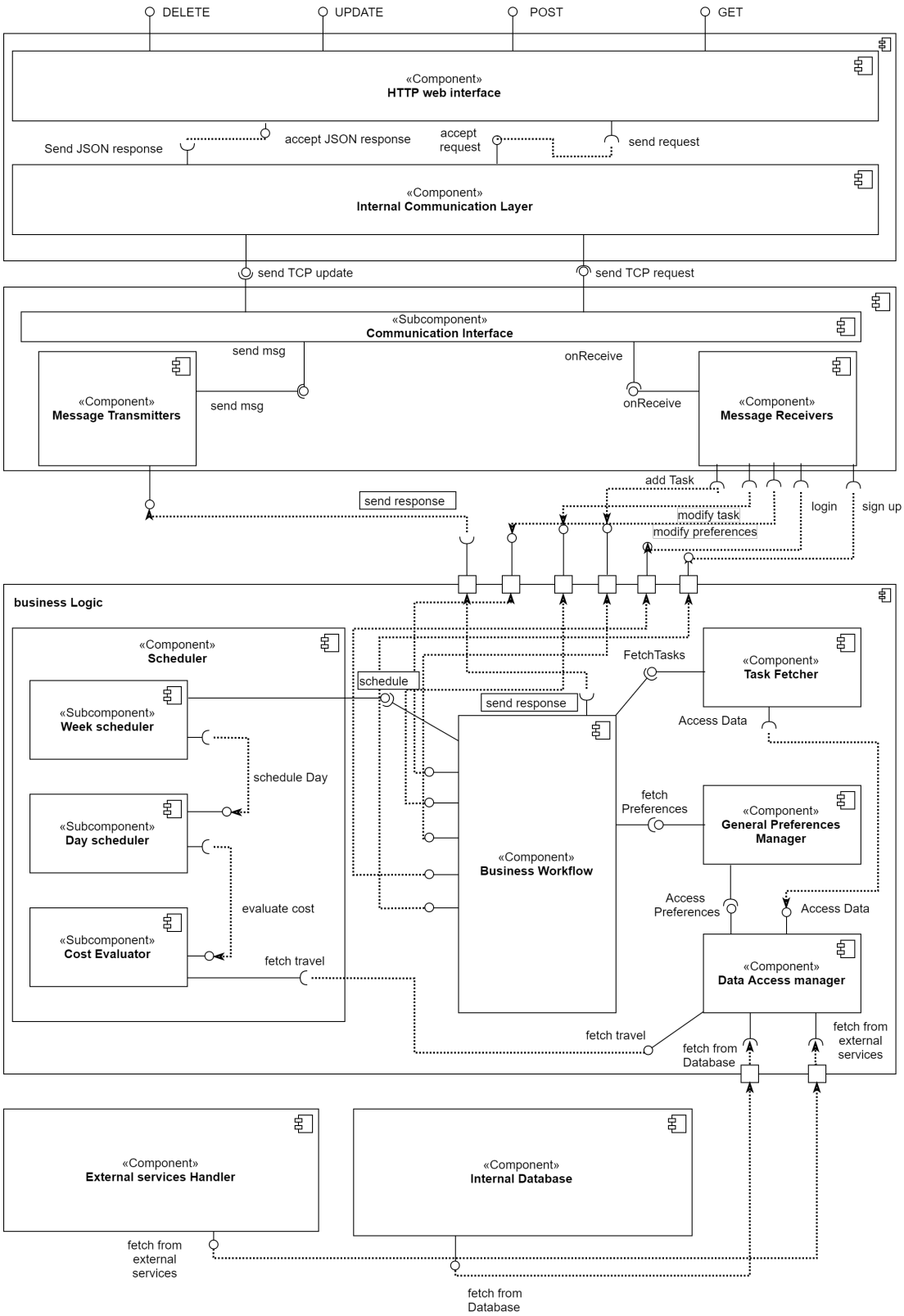

Figure 2.5: UML component diagram about back-end components

2.3 Deployment View

This part of the *Design Document* is devoted to explain at the best the four tiers that compose the *Travlendar+* system.

First of all, the application is created to work either on a web browser or on a mobile phone through an application. To have a system which is easily maintainable, both the clients will implement the *REST* protocol, with an *HTTPS* communication, in this way the mobile application and the web server can change their internal component, and the only thing to take into account is to implement the correct *RESTful* services.

The clients send requests to the *Travlendar+* web server, in which there is the *Server Façade*, a component use to receive messages from the Internet and to send them to the *Application Server*.

The *Application Server* is the server that contains all the *Business Logic* components. This layer runs the main functionalities of the *Travlendar+*, and also is used to access to the internal database, to store useful user's data. Notice that the *Application Server* is in a *demilitarized* zone (DMZ), in order to avoid possible hacks from the Internet. In this zone is placed also the *Database Server*, in this way the important data are stored in a secure way.

Finally the *Application Server* is also used to communicate with the *External Services*. These services are helpful to the core features of *Travlendar+*. Indeed, the application needs to know the best travel between two tasks, with additional information (such as the estimated travel time and the expected traffic); the travel means, with their time tables, and finally the various tickets' costs.

2.3.1 Tier architecture

As seen before, *Travlendar+* relies on a 4 tier architecture. this choice surely either increases the overall cost of deployment (in terms of money or time) and the maintenance cost, but is surely useful in terms of the main software attributes as described in the *RASD* Document, which are:

- **Security**: the aim of this architecture is to protect both data and business logic from malicious intrusions. the way this requirement is accomplished is by using two "access" servers to the local network which hosts the application: the *Aapplication Server Façade* and the *External Services Server*. these two machines are protected each one by a firewall who denies access to all connection except the ones coming to the https port and the ones used by the APIs. In this situation is simpler to imagine these two servers as separate machines, but it is also possible to host the relative software components under a single machine. In both ways, the fact that the local area network is not directly accessible from the world wide web is accomplished.

- **Scalability**: using a 4 tier architecture, we clearly distinguish machines devoted to communication with the web (for example, the web server that communicates with the client), machines devoted to operations and business logic, and machines devoted to data. that means that is is fairly simple to expand the network who deals about business logic features without changing the network interfaces to the world wide web: for example, it can be useful to add more serviced devoted to business logic when a certain amount of active users is reached. in this way, there's

no need to modify the entire architecture to do it, but only the components who interface themselves to the business logic.

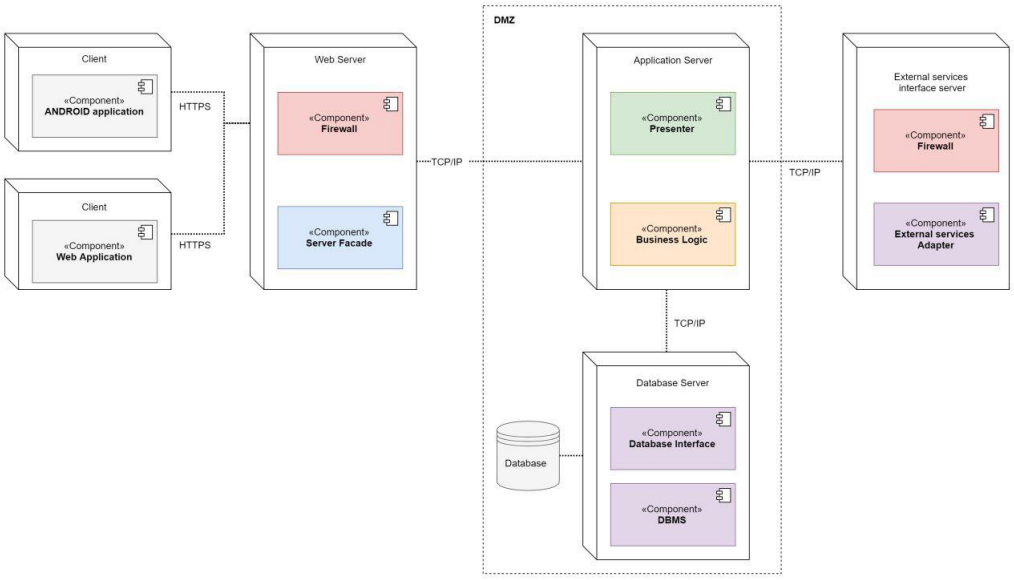

Figure 2.6: Deployment Diagram for *Travlendar+* System

2.3.2 Back-end infrastructure

the back-end infrastructure will be, as described in the *RASD* Document, based on server powered by a UNIX operating system, for example GNU/Linux or Free BSD: these operating systems are well known as great platform when dealing with web and application servers. Moreover, it is important to specify what approach can be used in order to expand the infrastructure when a certain number of active users is reached. A possible way for expanding the platform is to build a server farm containing several clones of the application server; these can work each one on a certain number of requests, so that the whole system is easily upgradable but also fault tolerant. Nonetheless, nowadays several container services are available on the market: these services takes care of the physical structure of the back end infrastructure, leaving to the customer only software maintenance.

2.4 Database View

In this part of the *Design Document* is reported the design and the main entities used by *Travlendar+* System.

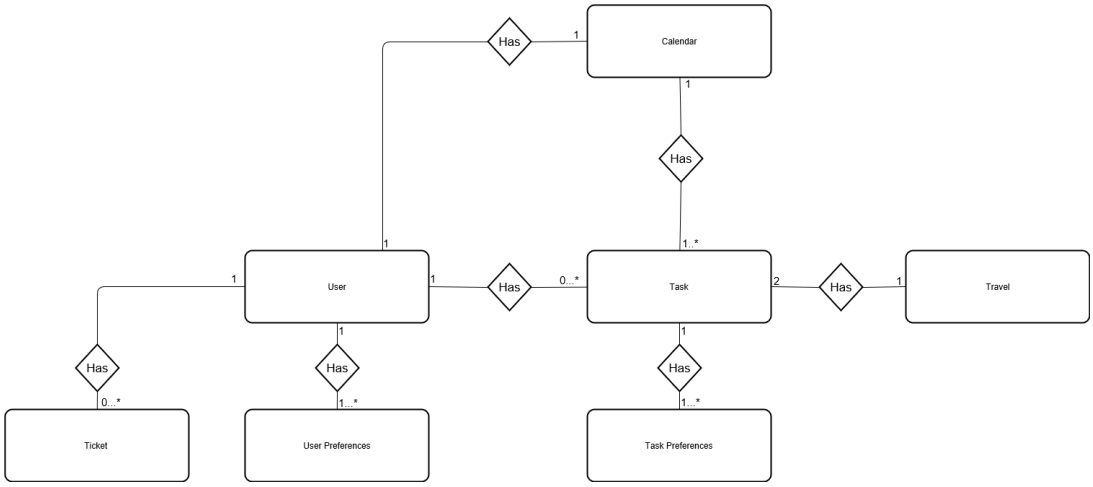

Figure 2.7: E-R Diagram

As can be notice, the Database is composed by six main entities, useful for the correct behavior of the *Travlendar+* System.

The main entities is the *User* one, that is used to store properly the main information about the user's accounto into the System, and it is also used for the Log In procedure. An user can have at least one preference, stored in the User Preferences table. Moreover, to taking into account the cost of a travel, the *Travlendar+* System needs to know if the user has some tickets yet, because in this situation the travel cost can be null. For this reason, the system stores also some information about the user's tickets. Obviously, the user's needs to schedule his calendar, to do that the *Travlendar+* System needs to know the tasks the user has to do, and also their preferences, in this way the system can compute a feasible calendar for him, and then it stores this calendar into a proper entity in the Database System.

To have more information about the attributes of each tables, here is reported a picture to explain well this:

User	
PK	**E-Mail**
	Name
	Surname
	Password
	User Residence

Ticket	
PK	**TicketID**
PK	**E-Mail**
	availableTravels

User Preference	
PK, FK	**E-Mail**
	takeCar
	takeBus
	takeCarSharing
	takeBikeSharing
	walk
	hasSeasonTicket

Figure 2.8: Attributes of the user-related tables

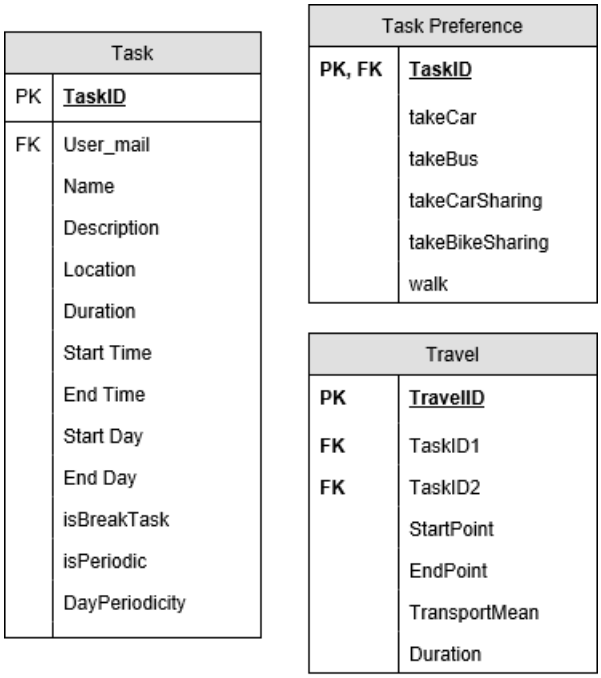

Figure 2.9: Attributes of the task-related tables

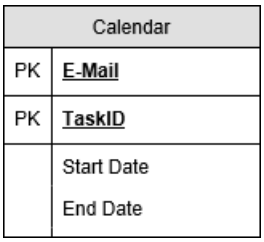

Figure 2.10: Attributes of the calendar-related table

2.5 Runtime View

In this section the run-time view of our system will be described with respect to its main functionalities. We mainly focused our attention on critical aspects such as the creation of a new task and the management of shared based vehicles.

2.5.1 Sign up and Log in

The sign up process is not a critical one but we have decided to include its description since it involves all the main layers of our application. The process starts with the Client sending a sign up request, which is made up of a mail and password to the Application Façade; this request is then packed into an HTTPS message and sent to the Presenter which is in charge of parsing the message and dispatch it to the Business Logic. At this point the data received will be checked and if everything its approved, a new user will be registered and a confirmation mail will be sent to the corresponding email, otherwise an error message will be sent to the Client passing through the Presenter and

the Application Façade. The same is valid for the Log In process but in such a case if the data are correct then the Client will be validated and logged into the system.

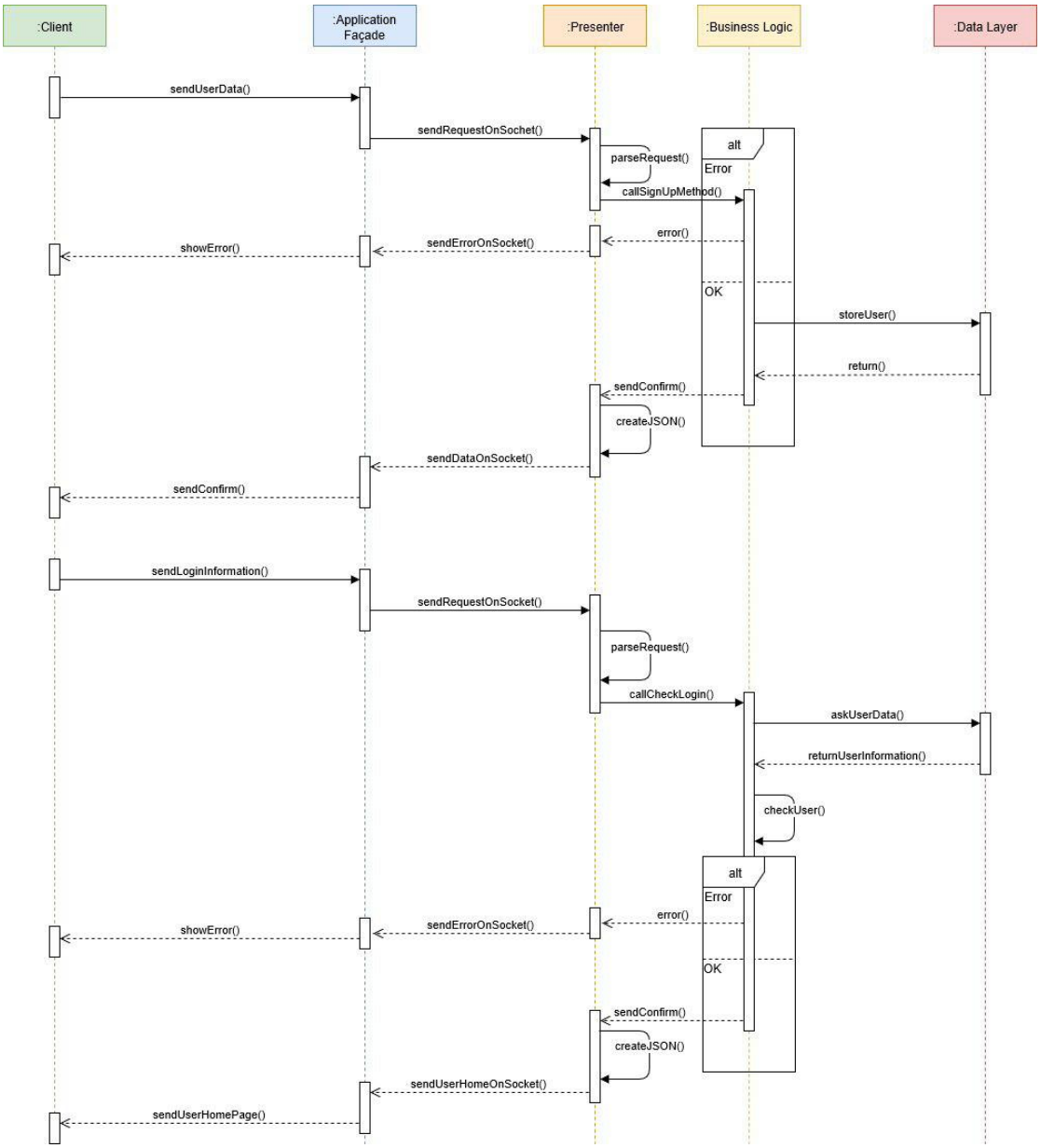

Figure 2.11: Sequence Diagram for the Sign up of a new User

2.5.2 Add a Task

The creation of a new Task starts with the Client sending all the information required for the creation of a new Task (see section 2.2.2) to the Application Façade. This information will be properly packed in an HTTPS message and sent to the Presenter. At this point the message will be parsed and dispatched to the Business Logic component in charge for the Business WorkFlow; now the system will try to schedule the newly inserted task in the calendar (loaded from the Database) and, if no conflicts occur, the new schedule will be sent to the user through the Presenter and the Application Façade. If the calendar is

approved by the client then it will be stored in the Database. Notice that a very similar process is performed when the Client require to modify a task preference or wants to remove a task.

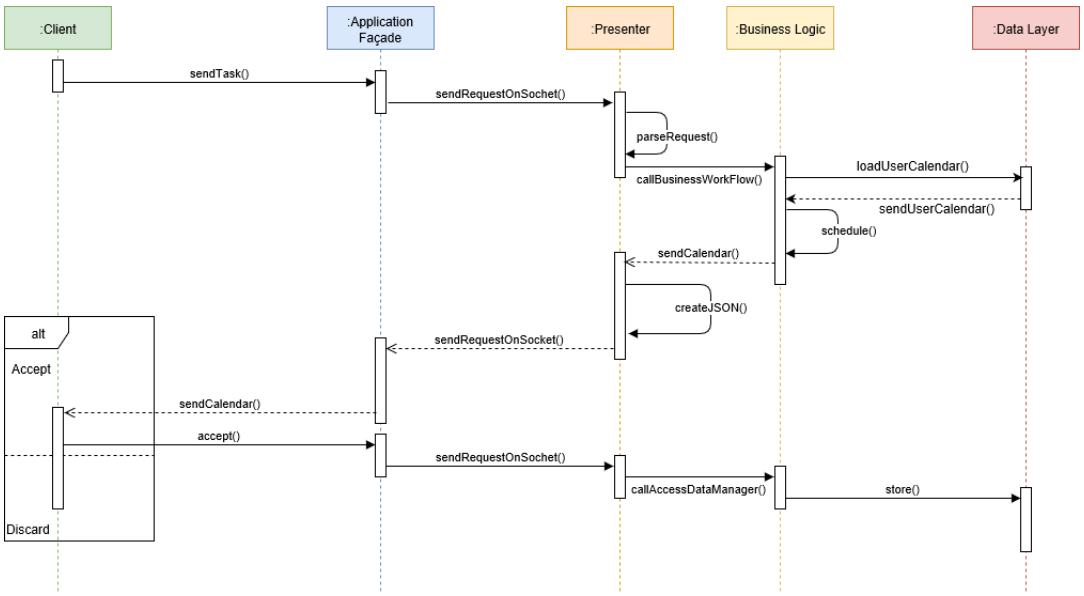

Figure 2.12: Sequence Diagram for the creation of a new task

2.5.3 Reminder

The usage of every kind of shared based vehicle is supported by a reminder system, as previously detailed in the *RASD* Document of the project. For this reason a component of the Business Logic will be in charge to ask the calendar of every user to the Data Layer and then check the availability of such vehicles 30 minutes before the travel starts. If there are vehicles available within the maximum walk range, then the user will be notified of the closest one; otherwise a reschedule of the day will occur and the user will be notified of the potential changes.

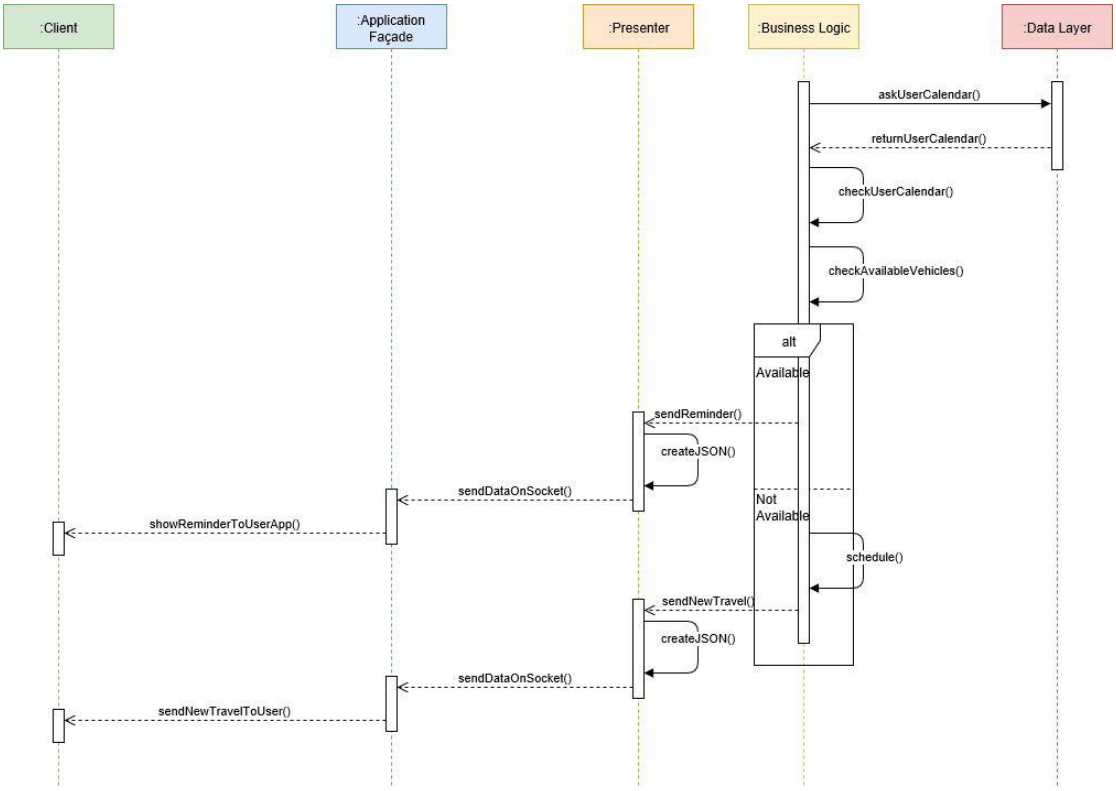

Figure 2.13: Sequence Diagram for checking availability of shared based vehicles

2.6 Component Interfaces

Client Interfaces

The Client has some internal interfaces in order to render pages, to take the user's input events and to compute some information. To do these stuffs, the client needs to have two main modules, one to handle the server's data and another one to handle the user's input. Indeed the client has:

- Parser: an interface used to receive the server's data, written in JSON format, and to translate them into a HTML pages.
- Render: a component used to render the HTML pages previously built.
- User Event Handler: an interface used to take the user's input to send to the server.
- REST Invocator: interface used to package the user's data to send them through the *REST* protocol.

Server Façade Interfaces

The Server Façade layer takes the client's REST requests and send them to the Application Server. In order to create a easily maintainable web server, this layer has:

- HTTP Web Interface: interface used to handle properly the client's requests and also to send back information to it.
- Internal Communication Layer: used to send, through TCP connection, the client's requests to the Application Server.

Presenter Interfaces

When a client's request comes in the server, the first layer it meets is the Server Façade, that is used to create a demilitarized zone for the Application Server. Indeed, once the request comes in the Server Façade, it sends a TCP request to the Application Server. Then, the last one, to work properly, needs a layer used to call the correct business logic functionality. So, the Presenter Layer has

- Communication Interface: module used to handle the connection with the Server Façade component, through a TCP connection.
- Message Receiver: module that calls the properly business logic function to manage the user's request.
- Message Transmitters: module uses to send back data from the business logic to the client.

Business Logic Interfaces

This module represents the main part of the *Travlendar+* application. It contains the core features and the functionalities used to schedule, as well as possible (obviously according to the given preferences), the user's tasks. To do that, it needs:

- Business WorkFlow: the first interface called by the presenter, it is used to control the operations flow to schedule the tasks in a feasible calendar, and also it sends the reminders to the user in order to let him know when he needs to start his trip, or if something changes (probably because of the unavailability of the shared-based vehicle).
- Scheduler: the main component calls by the Business WorkFlow, used to schedule the various tasks. Since the complexity of this one, it is divided into three main modules:
  - Cost Evaluator: module used to compute the Day schedules *cost*, in this way the application can know which day schedule is the best.
  - Day Scheduler: module used to create a single day calendar, with both the tasks and the travels to reach each task.
  - Week Scheduler: module used to divide the tasks in the correct day, in order to simplify the schedule problem.
- Task Fetcher: interface used by the Business WorkFlow for the tasks loading. It interacts mainly with the Access Data Manager to load the tasks from the Database. Moreover this component is used to handle properly the tasks' specific preferences.

- Access Data Manager: interface that connects the Business Logic Layer with the Data Layer. Indeed is used to do the Input/Output operations. Also, this module is used to ask external services, such as the map service or the transport mean one; in this way the Business Logic can schedule the user's various trips, according to the information given by the external services.
- General Preferences Manager: module used to handle the general preferences, which are the preferences that the user specifies for all tasks.

Data Layer Interfaces

This is the layer which is used to store the users' information. This layer has, as main interface, the connection with the DBMS in order to load or store useful information for the *Travlendar+* application.

2.7 Selected Architectural Styles & Patterns

The aim of this section is to present the main architectural decisions about the software architecture. In particular, given the extreme simplicity of the client application, the back end architecture will be mainly discussed.

As discussed before in the document, all software components made extensive use of simple patterns such as *observable - observer* and *adapter* patterns. In general, the chosen approach is to make the Business Logic as much stateless as possible: in this mean, the core components are designed to work as much functionally as possible, leaving the *Business WorkFlow* in charge of managing operations in their entirety. this architecture, at back-end level, resembles a lot the *Boundary - Control - Entity* pattern, where the boundary is represented by the *Presenter*, the core entities are mainly designed only to contain data and not to contain methods, and the control is represented by *Business WorkFlow* and other core components. A stateless approach is simpler in terms of maintainability, resource management, scalability.
In the next two subsection two interesting topic will be presented: the first one is regarding how to manage incoming messages in the presenter layer without defining complex message visitors that are difficult to expand and to maintain; the second one is a summary about pattern that are useful when implementing the business logic.

2.7.1 Message Receivers

the main idea about the message receivers is accomplish the same result obtained using the *visitor* pattern without actually using it. this because of the lack of scalability and flexibility imposed by that pattern; the idea is then to build a class similar to the one readable in the next page. the idea is to define a general receiver which *decorates* (see the *Decorator* Pattern) itself. that can be done in many ways, that are almost equivalent in this context. then, a subclass of the Receiver can be defined for each message that needs to be managed. both the superclass and the subclass possess a

```
public onReceive(message);
```

method, but when the superClass handle a generic message, the subClass handle a very specific message.

```
public class SubReceiver<T extends Message> extends Receiver{
    public onReceive(T message){
        // do something
    }
}

public abstract class Receiver{
    Receiver next;

    public onReceive(Message message){

        next.onReceive(message);
    }
}
```

As can be seen in the code, the `onReceive(messgae)` method of the subclass handle the message in its own way, but the method inside the superclass simply call the same `onReceive(message)` method in the decorated object. In this way, a chain of Receivers where each one decorates the other can be defined, and the top Receiver serves as entry point for the chain, that is explored until a specific Receiver that handle the message is found. if the receiver is not found, an exception can be thrown.

this pattern not only is more maintainable than the normal *visitor*, but also gives the possibility to modify at run time what messages can be received and what can not.

2.7.2 Business Logic Design Patterns

As described before, the business logic architecture is designed to be as much stateless as possible. Moreover, another design choice is to make all the business logic entities immutable once created; this can be simply done through the *factory* pattern, which is widely used to build entity objects from the data retrieved by the *data access manager*. In fact, the *task fetcher* and the *general preferences manager* are actually factories.
another important pattern is the *decorator*, which can be used to represent user preferences as complex combinations of more "elementary" entities, both for the task or for the general preferences. Eventually, another important problem that needs to be dealt with is how to actually manage preferences: our idea is to implement a *strategy* pattern, where each preference represent a strategy on how to build the correct travel and how to verify if the travel is well built or not. this can be done at cost calculation level, inside the scheduler.

Chapter 3

Algorithm Design

In this chapter the attention will be focused on the design of the most critical aspects of the *Travlendar+* system from the algorithmic point of view. For this reason we decided to support our arguments with some *Flow Diagrams* that clearly explain the main steps of the most complex algorithmic parts.
The most critical part of the system is clearly the scheduling algorithm, thus this part is divided in two main phases: the first one aimed at assigning a day to each task and the second one aimed at assigning a time slot to each task.

3.0.1 Week Scheduler

In the first stage the algorithm will receive as input a set of tasks with any kind of time behaviour but to be scheduled within the same week; on the passed set will be firstly performed a filtering operation which separates the tasks with fixed day behaviour from the others with flexible and variable time behaviour. At this point the ones with flexible and variable time behaviour will be assigned a fixed day, trying to uniformly distribute each task with respect to the day load of task and travel time. This will be done using a probabilistic approach based on a probability distribution function such as:

$$P(t \in d) = \frac{\tau(d) + \beta \sum_{i \in T(d)} i \to B(d) + \gamma t \to B(d)}{\sum_{j \in D} \tau(j) + \beta \sum_{j \in D} \sum_{k \in T(j)} k \to B(j) + \gamma \sum_{j \in D} t \to B(j)} \tag{3.1}$$

where:
Ω is the set of the task in the current week
$T(d)$ is the set of task in a given day d
D is the set of days for the current week
$t \in \Omega$
$d \in D$
$\tau(d)$ is the sum of the duration time of every task in a given day d
$B(d)$ is the barycenter of the task location for a given day d
$s \to B(d)$ is the distance between a task s and the baycenter for a given day d
β, θ are constants

After this stage every task will have a fixed day, thus they will be scanned in order to find evident conflicts between fixed time tasks.

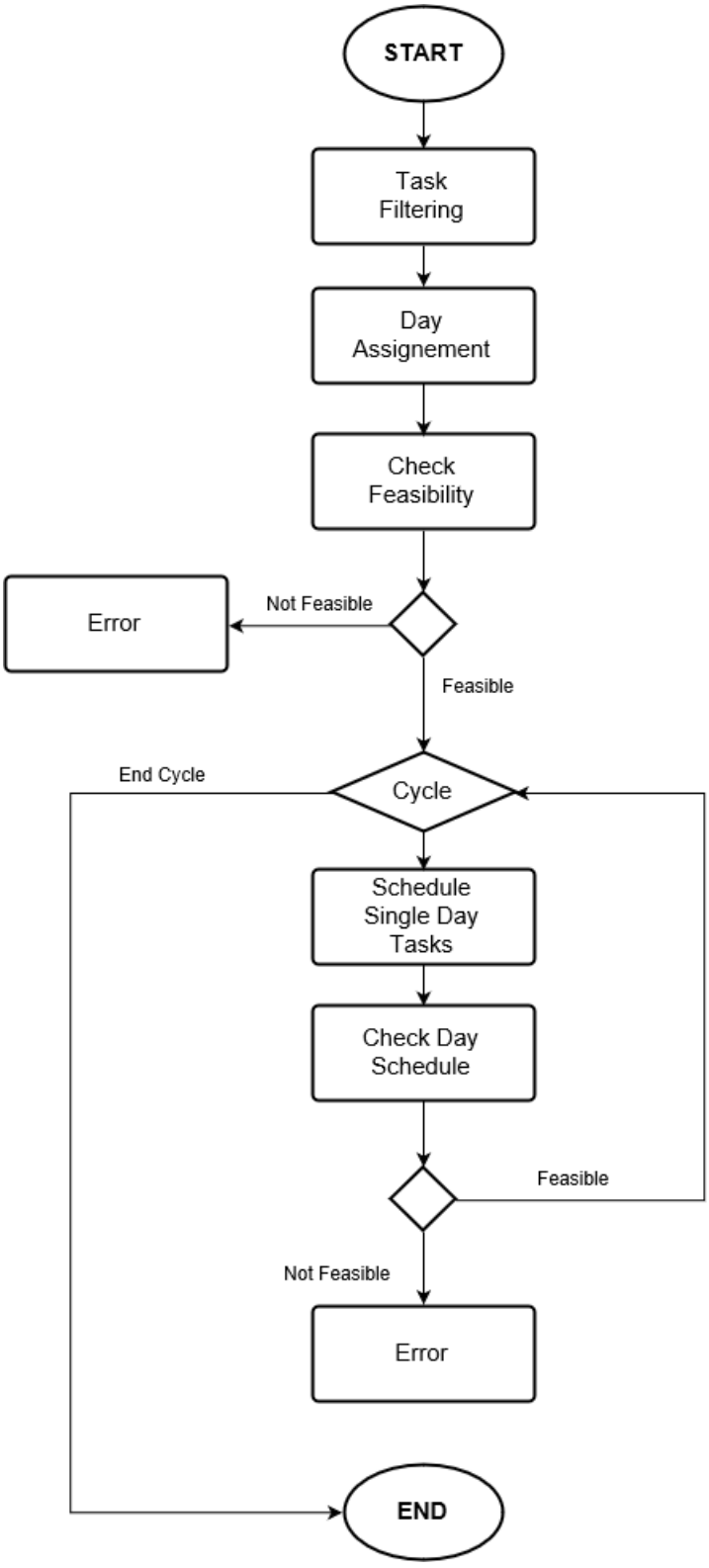

Figure 3.1: *Flow Diagram* for the day assignment problem

If so, the conflicting tasks will be postponed in the next feasible day at first, if even this solution result unfeasible, then they will be signaled with an error; once the feasibility

check ends the algorithm will start to cycle over each day, invoking the algorithm that schedule tasks in a given day (which will be detailed later). The resulting day schedule will be double checked and, if there are no errors, the scheduling proceeds with the next day, until the end of the week is reached. Notice that, if a conflict occurs in the scheduled day, then the algorithm will, at first, try to postpone one of the conflicting task to the next feasible day, then it will end signaling the conflicting tasks.

3.0.2 Day Scheduler

Now that from an high point of view it's clear how the scheduling algorithm operates, we will better detail how the assignment of time slot to each task in a given day works. The time scheduler algorithm will try to organize the task in tree main phases: at first the ones with a fixed time slot, then the ones with flexible time slot and finally the ones with a variable time slot. In each of these phases the scheduling will be based on the Branch and Bound algorithm thus, given a certain task set every possible combination will be explored and evaluated if necessary. For this reason, in the worst case the overall complexity is in the order of $O(\exp(n))$ with n the number of the tasks in the given set. But in average it's proofed, under certain assumption, that has a sub-exponential complexity (see the Reference documents section). If during the Branch and Bound the algorithm discovers that a task inevitably conflicts with the others, then depending on the day behaviour of the task different actions will be taken: if day fixed then the algorithm terminates and signals the conflicting task, otherwise it will postpone the task to the next feasible day and continue the day scheduling process, until all the tasks are parsed.

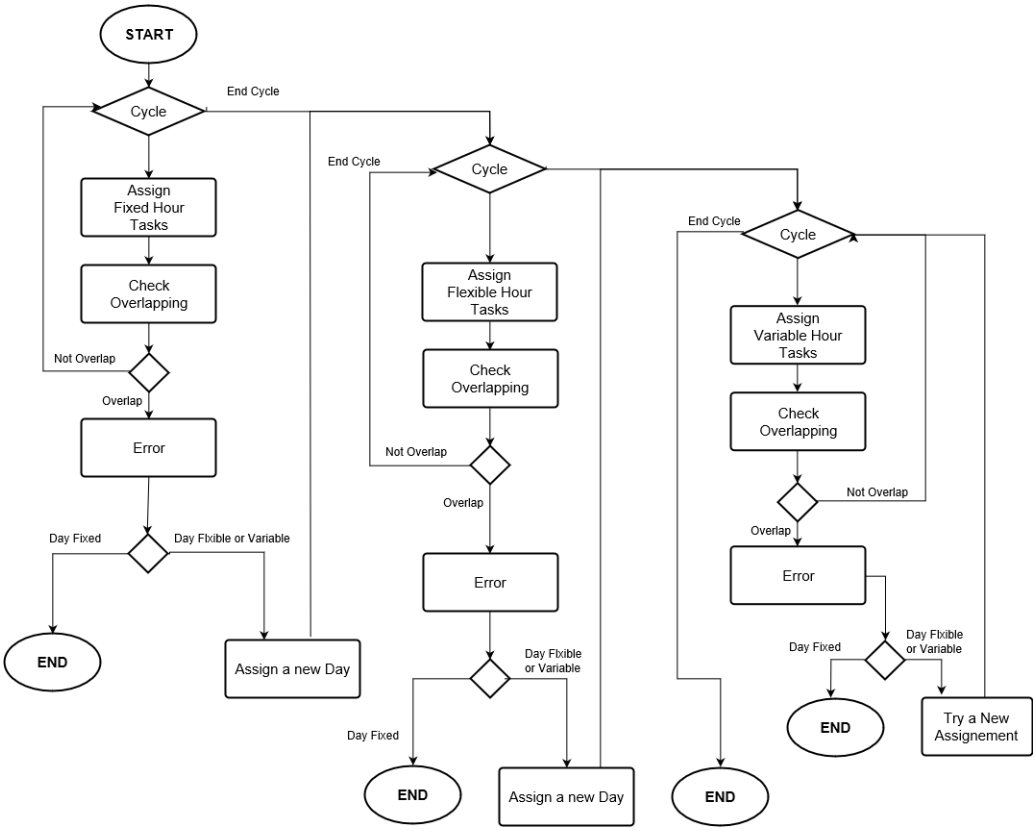

Figure 3.2: *Flow Diagram* for the schedule algorithm on a single day

Chapter 4

User Interface Design

In this chapter we discuss in detail the design of the User Experience by the *UX Diagram* and some mock-ups. The main intention during this stage of the project has been to extremely simplify the interactions between the user and the system, trying to guarantee the most intuitive and smooth experience.
The first page the user will interact with is the Home Page obviously, from this point it is possible to log in or sign up into the system and also see a description of the main features of the *Travlendar+* application. Immediately after the log in phase the user will be redirected to his personal page. Here, he can perform several actions, such as:

- See the scheduled tasks
- See a detailed description of the current day schedule
- Have access to both the sharing based services and the public transport services
- Add a new task
- Select a task to modify it or to simply look at more detailed information about it
- Change the global preferences
- See his profile
- Change the calendar granularity (by day, week, month)
- Log out from the system

In particular, when the user wants to start create a new task a pop up will be displayed and will ask the user to fill a number of fields to better detail the new task. A similar approach will be used to manage the selection and modification of a task. On the other hand, when the user wants to have access to global preferences a new page will be displayed where all the travel related preferences might be modified. Finally, when the user wants to buy a bus ticket or rent a shared based vehicle it will be redirected to the appropriate web page or application by mean of a button.

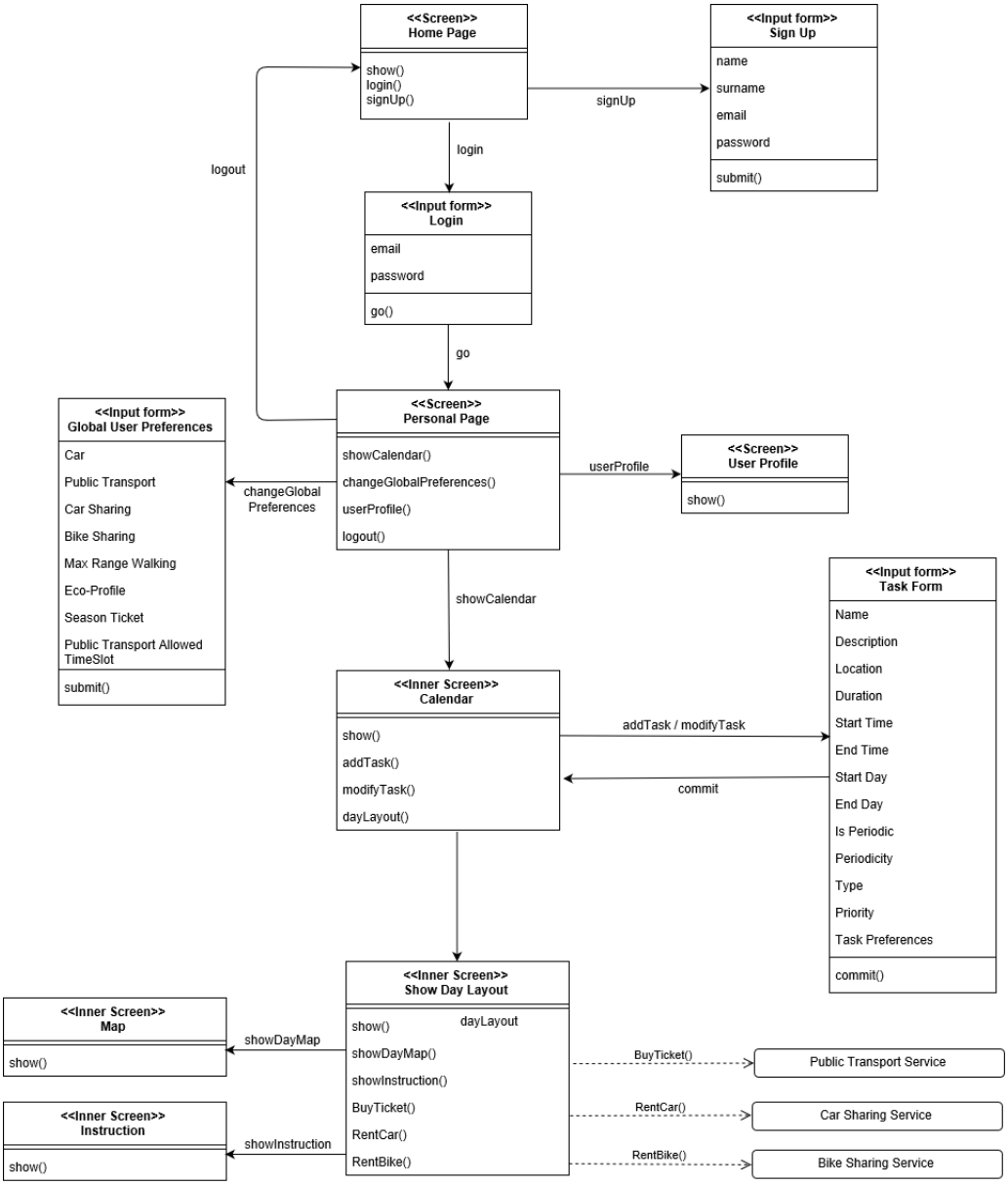

Figure 4.1: UX Diagram

As the reader may notice, the two pictures below represent a possible mock-ups for the Home Page and the Sign Up screen.

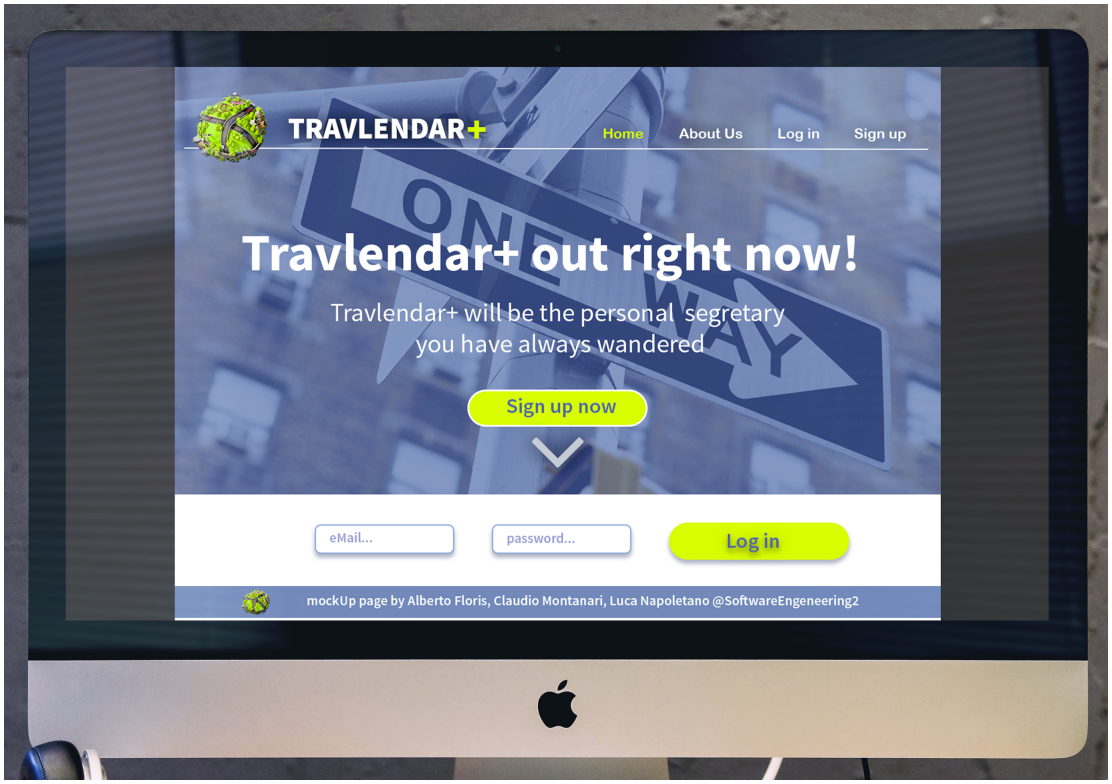

Figure 4.2: Mock Up for the Home page of the *Travlendar+* application

Figure 4.3: Mock Up for the Sign Up screen

The two pictures below instead, stand for the user Calendar Page. In the left part of the screen the user has a summary of the current day schedule enriched by two buttons that stands for maps directions and vehicle or ticket services.
On the right part is placed the calendar and just above the calendar the user has a menu bar for controlling the calendar layout, adding a task, change global preferences or log out of the system.

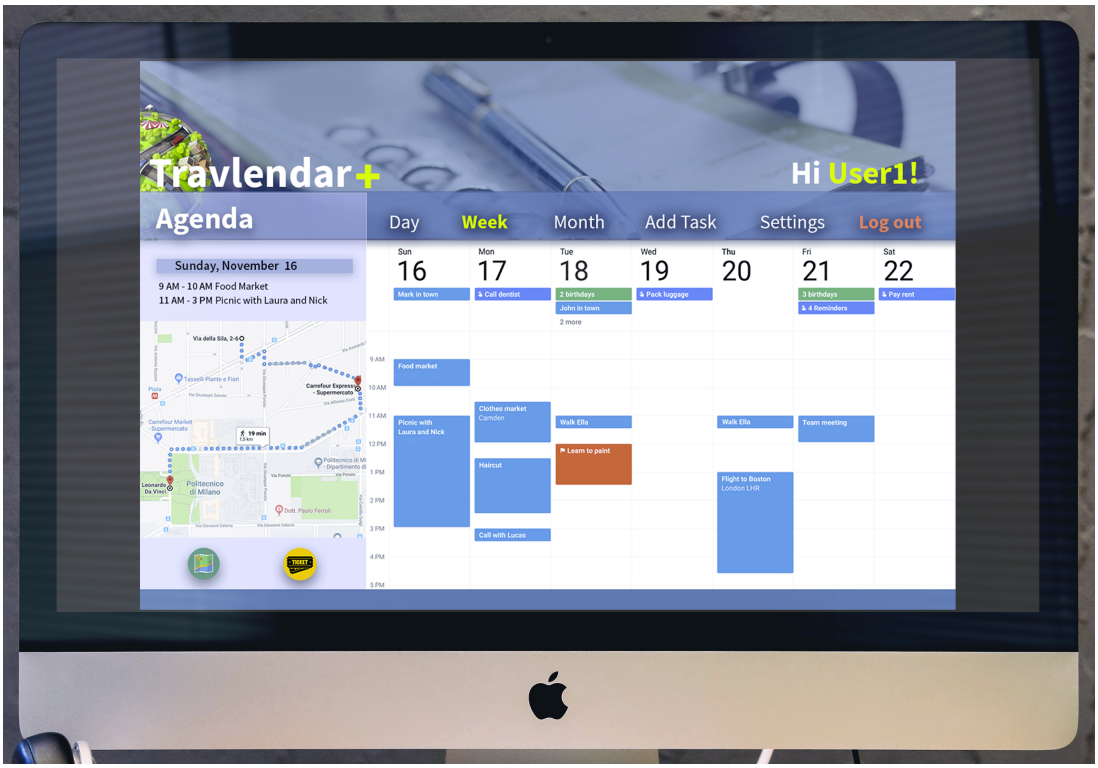

Figure 4.4: Mock Up for the Calendar page

Figure 4.5: Mock Up for the creation of a new task

Chapter 5

Requirements Traceability

In this part of the *Design Document* are reported the requirements, taken from the *RASD* Document (version 1.2), in order to create a connection with the features, the modules and the components described through the entire document.

Component	**Satisfied Requirements**
Sign Up System	R1, R16
Client	R14, R15
Data Access Manager	R13, R16, R17
Data Layer	R9, R11, R16, R17
Cost Evaluator	R17, R18
Business WorkFlow	R4, R6, R7, R8, R11
Scheduler	R3, R4, R5, R6, R7, R8, R10, R12, R18
Reminder	R2, R3

Table 5.1: Main features connected with the satisfied functional requirements

As can be notice from the table, there are eight main functionalities of the *Travlendar+* system that involved in the satisfaction of the requirement. First of all, there is the *Sign Up* functionalities, that is used to satisfy the requirement R1 (the user should specify a residence at registration time), in this way the system can know where the user starts his day and where he ends it one. This features also satisfy the requirement R16 (the system should permit a new user signing up through a proper page).

The *Client* component permit the satisfiability of the requirement R14 (The application should permit the visualization of three different calendar visualization: by month, by week or by day) and also the requirement R15 (The application should permit the selection of a single task, in order to change preferences concerning that task).

The *Data Access Manager* module is used to load and store, in a Database, the main useful business entities, so it satisfies the requirement R16, because the user data should be stored in the Database at signing up time; the requirement R13 (the system should permit the purchase of a either a public transport ticket or a shared-based vehicle service) and the R17 (The system, in order to choose the best travel option, has to

consider two main parameters: cost of the travel and the distance between the two locations), these ones are satisfied because the *Data Access Manager* layer is responsible of the communication with the external services, such as the map service or the transport service.

The *Data Layer* while is pretty similar to the *Data Access Manager*, but it is a layer connected directly to the DBMS service. So it satisfy the requirements R16 and R17, because this layer is useful to store both the user's data and some others data concerning the various travels; but also this layer is useful to satisfy the requirements R9 (Each Task can be repeated a finite or an infinite number of time) and R11 (The software should permit to modify both the general user's preferences and the tasks' preferences).

The *Cost Evaluator* modules is used to have a good value that describe well the "goodness" of a schedule, by the computation of some mathematical function to have a numerical value and evaluate it. Moreover this module is used to load the costs of the various public services (such as the public transport service or the shared-based vehicle). So this component satisfies the requirement R17, because it computes effectively the function used to evaluate a single schedule, and the requirement R18 (the system should suggest the best travel ticket, if the public transport service is a user's preference), because this component also compute how many times the user has to take the public transport means, and it knows how much a ticket cost.

The *Business WorkFlow* component is the main responsible of the correct management of the work flow, in order to compute properly the user's calendar. So the requirements it satisfies are: R4 (If there are two or more overlaps, then the application should ask to the user how it will handle this situation), R6 (The system should do a reschedule, if the user modifies, adds or deletes one or more tasks), R7 (If the user adds, deletes o modifies a task during the current day, the application may not be able to schedule a feasible calendar), R8 (If the user adds a task that isn't reachable in a feasible time, the system should send a warning to the user) and R11. All these functional requirements, as can be notice from the *RASD* Document, are all tied to the correct management of the application's (or user's) events.

The *Scheduler* component is used to schedule the user's tasks in a feasible way, and it is the main features of the *Travlendar+* application. Indeed it satisfies the main functional requirements: R3 (If there isn't a shared-based vehicle, the system should compute a feasible alternative), R4 (If there are two or more overlaps, then the application should ask to the user how it will handle this situation), R5 (The software should permit to the user to specify when he wants to come back at home), R6 (The system should do a reschedule, if the user modifies, adds or deletes one or more tasks), R7 (If the user adds, deletes o modifies a task during the current day, the application may not be able to schedule a feasible calendar), R8 (If the user adds a task that isn't reachable in a feasible time, the system should send a warning to the user), R10 (The application should taking into account if the user has taken his private vehicle during the day, in order to permit the user to retake his vehicle), R12 (The application should allow the user to insert tasks which are time-variable during the week) and R18 (the system should suggest the best travel ticket, if the public transport service is a user's preference). In this case, all the requirements are concerning the computation of a feasible calendar for the user.

Finally, the *Reminder* module is used to check, if a travel consider to take a shared-based vehicle, the feasibility of that schedules, in order to avoid the situation that, when a user needs to start his trip, there are no vehicle available. So this component satisfies the requirement R2 (The application must send a reminder to the user 30 minutes before

he should begin the trip. Moreover, the application has to verify if a Sharing-Based vehicle is effectively available, if actually it is a user preference) and R3 (If there aren't available shared-based vehicle, the system should compute a different travel solution that is feasible). Both of them are related to the reschedule of a trip if there are some issues concerning the shared-based services.

Chapter 6

Implementation, Integration & Test Plan

This chapter of the *Design Document* is devoted to explain the strategy that will be used while the *Travlendar+* System will be develop, in order to define the procedure to follow when each components and each modules will be created, tested and integrated in the entire application.

6.1 Integration Strategy

The integration of the *Travlendar+* components should start as soon as possible, in order to start the testing of both the single modules and the system's component. But, before starting the integration phase, there are some precondition that should be satisfied:

- The documents should be completed and written in a final version, both the *RASD* and the *Design Document*. Moreover, these documentation should be available to all the people involved in the project.
- The low-level stuffs should be available to the *Travlendar+* system. Indeed the *Business Logic* layer, to work properly, needs to have a *DBMS* ready and configured (so the *Data* layer should be done), and some *APIs* should be available, such as the map service, the transport service, the car-sharing and the bike-sharing services.

Once these precondition are satisfied, when a component will be developed, it can be tested alone, and if it will work, then it can be integrated to the others components, in order to verify first the correct functionalities of the single module, and then that it works properly in the entire system.

The implementation should start with the elementary component of the *Business Logic* layer, in this way the testing will be focused on the main, and most important, logic features. Indeed, the firsts components to develop are:

1. Core Entities, in order to have the correct data structures.
2. Access Data Manager, in this way the system can interact easily with a database.
3. Scheduler, the main function of the *Business Logic* layer, that will be developed in the follows order:

(a) Cost Evaluator, component used to check the goodness of a travel or a schedule.

(b) Day Scheduler, used to know if the algorithm works properly.

(c) Week Scheduler, used to divide test if the application can divide properly tasks through various days.

4. General Preferences Manager, that will interact with the scheduler to set up the user's preferences in a correct way.

5. Task Fetcher, indeed it should take various tasks to give them to the scheduler module

After the development of the *Business Logic* layer, the focus will be moved on the *Presenter Layer*, in order to start the testing of the communication as soon as possible. Moreover, after the implementation and the integration of the *Presenter Layer*, the Application Server can be set up to start its work.

Then the Application Server should interact with the internal Web Server, so the next move will be the development of the Application Façade layer, necessary to test the communication with the clients. In parallel, the clients can be developed, so the entire communication part of the *Travlendar+* application can be tested.

Here is reported the diagram that shows the testing dependencies. As a notation, the arrows that goes from a component A to a component B means that the component A should be done before the B one in order to integrate and to test properly the modules.

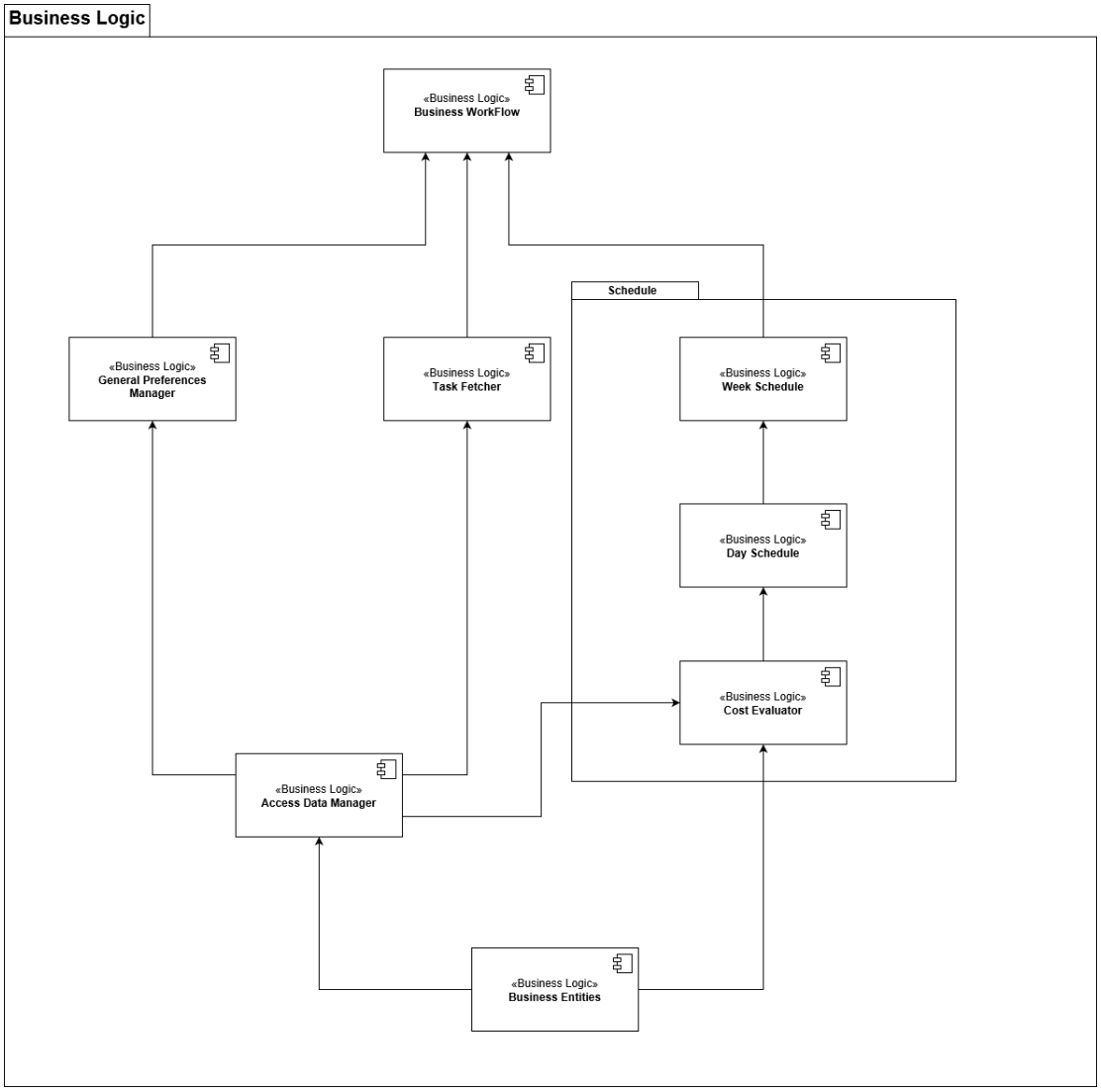

Figure 6.1: *Travlendar+* integration, implementation and testing diagram

6.2 Testing Description

Login System

Test ID	T1
Involved Components	Application Façade, Message Transmitter, Message Receiver, Business WorkFlow, DBMS
Input Specification	User's email and password
Output Specification	System should check if both email and password are correct
Exception	Either email or password are unvalid, so the system should show to the user an error page
Description	The system should be able to log in an user, by the verification of his email and password. This procedure is done by receiving a message in the web server, send it to the Application server through the *Message Receiver* component, analyzed by the *Business WorkFlow* component and the *DBMS* and then accept or discard, by send back a page through the Presenter's *Message Transmitter* component
Environmental needs	

Table 6.1: Testing Description of Login system

Sign Up System

Test ID	T2
Involved Components	Application Façade, Message Transmitter, Message Receiver, Business Work-Flow, DBMS
Input Specification	User's first name, last name, email and password
Output Specification	System should add the user to the internal database, and send back an email to him with a summary of the operation done
Exception	User enters a mail that is already registered in the system. So the system send back an error page and it doesn't register the user
Description	The user send his information required by the system to save him in the DBMS. This message goes from the client to the Application Server, which thanks to the *Business Logic* module can check the correctness of the information, and, if there aren't errors, store the user and give him an account
Environmental needs	

Table 6.2: Testing Description of Sign Up system

Add Task System

Test ID	T3
Involved Components	Application Façade, Message Transmitter, Message Receiver, Business WorkFlow, Scheduler, DBMS
Input Specification	Task's Information
Output Specification	System should create a calendar with the new task inserted
Exception	The newer task conflicts with some other older tasks, and the system cannot schedule a feasible solution for the new set of tasks. In some case this situation can happen because the newer task is unreachable in a feasible time from the others tasks, according to the given contraints
Description	The system should add the task to the DBMS system, and should schedule a feasible calendar with the newer task, according to the given time constraints, the travel constraints and the user's preferences
Environmental needs	T1: the user should be logged in the system

Table 6.3: Testing Description of Add Task System

Remove Task System

Test ID	T4
Involved Components	Application Façade, Message Transmitter, Message Receiver, Business WorkFlow, Scheduler, Access Data Manager, Task Fetcher, DBMS
Input Specification	Task ID that will be removed
Output Specification	System should remove the selected task
Exception	The given Task ID doesn't exist, so the system show an error page
Description	The user specifies the task he wants to remove, then he sends this information to the system. At this point, the *Business WorkFlow* components interacts with the *Access Data Manager* to search the selected task and remove it. Finally, the system asks to the user if he wants to reschedule or not the calendar
Environmental needs	T1: the user should be logged in the system

Table 6.4: Testing Description of Remove Task System

Modify Task System

Test ID	T5
Involved Components	Application Façade, Message Transmitter, Message Receiver, Business Work-Flow, Scheduler, Access Data Manager, Task Fetcher, DBMS
Input Specification	Task ID that will be modified
Output Specification	System should modify the settings associated to the selected task
Exception	The given Task ID doesn't exist, so the system show an error page. The newer preferences creates a non feasible calendar, so the system should return to the previous calendar and show to the user an error page
Description	The user specifies what kind of settings, associated to the selected task, he wants to modify. So the system has to store these information, and then, if necessary, it needs to reschedule the user's calendar
Environmental needs	T1: the user should be logged in the system T3, T4: the system should work properly with the add and the remove procedures of the Task management

Table 6.5: Testing Description of Modify Task System

Change User's Global Preferences System

Test ID	T6
Involved Components	Application Façade, Message Transmitter, Message Receiver, Business WorkFlow, Scheduler, Access Data Manager, Global Preferences Manager, DBMS
Input Specification	The preferences the user wants to modify and the values of them
Output Specification	System should store properly the new preferences and show a new calendar associated to the new preferences
Exception	The given Task ID doesn't exist, so the system show an error page. The newer preferences creates a non feasible calendar, so the system should return to the previous calendar and show to the user an error page
Description	The user specifies what kind of settings, associated to the selected task, he wants to modify. So the system has to store these information, and then, if necessary, it needs to reschedule the user's calendar
Environmental needs	T1: the user should be logged in the system T3, T4: the system should work properly with the add and the remove procedures of the Task management

Table 6.6: Testing Description of Change User's Global Preferences System

Reminder System

Test ID	T7
Involved Components	Application Façade, Message Transmitter, Business WorkFlow, Scheduler, Access Data Manager, DBMS
Input Specification	
Output Specification	System should send a message to the user's client in order to notify when he needs to start his travel or to warn the user if there is a trip change, because of the unavailability of a shared-based vehicle
Exception	
Description	The system, 30 minutes before a trip (based on either a car-sharing or a bike-sharing service) starts, check if there is a vehicle available in the user's location. If there is, the application sends a reminder to notify the user he should rent the vehicle, otherwise, the *Travlendar+* system tries to find a new way to reach the next location
Environmental needs	T1: the user should be logged in the system T6: the system should work properly with the user's preferences

Table 6.7: Testing Description of Modify Task System

Break Time Task

Test ID	T8
Involved Components	Application Façade, Message Transmitter, Message Receiver, Business Work-Flow, Scheduler, Access Data Manager, Global Preferences Manager, DBMS
Input Specification	The user should specify when he wants to do his break time, either a fixed timeslot or a flexible timeslot during the day. He can specify also more than one break times
Output Specification	System should schedule a new feasible calendar, according to the user's and the tasks' preferences, with the break times added properly
Exception	When the user adds the break time task, maybe it conflict with others tasks inserted yet. So the system, in this case, should notify the user and ask him to change some settings concerning the various tasks
Description	This kind of task is little different from the others, because it doesn't need a location, so the *Travlendar+* system can recommend a place where the user can go to eat. When a user add this kind of task, the system has to schedule a proper feasible calendar
Environmental needs	T1: the user should be logged in the system T3, T4, T5: the system should work properly with the management of the various tasks T6: the system should work properly with the user's preferences

Table 6.8: Testing Description of Modify Task System

Chapter 7

Effort Spent

This section of the document describes all hours of work for each team mate.

- Alberto Floris
  - [17/11/2017] Team Reunion: Software Architecture Design, Algorithm Design discussion (3h)
  - [18/11/2017] Team Reunion: Software Architecture Design, Algorithm Design discussion (6h)
  - [19/11/2017] Team Reunion: Algorithm Design discussion, Component Diagram (4h)
  - [19/11/2017] Individual Work: Physical Architecture Diagram (2h)
  - [20/11/2017] Individual Work: Algorithm Design, Interfaces Diagram (2h)
  - [21/11/2017] Individual Work: Component Interfaces Diagram (4h)
  - [24/11/2017] Individual Work: Component Interfaces Diagram (4h)
  - [25/11/2017] Team Reunion: Integration and Test plan discussion (3h)
  - [25/11/2017] Individual Work: Deployment Diagram, Algorithm Design (7h)
  - [26/11/2017] Individual Work: Component Diagram, Deployment Diagram, Architectural and Patterns Design (10h)
- Claudio Montanari
  - [17/11/2017] Team Reunion: Software Architecture Design, Algorithm Design discussion (3h)
  - [18/11/2017] Team Reunion: Software Architecture Design, Algorithm Design discussion (6h)
  - [19/11/2017] Team Reunion: Algorithm Design discussion, Component Diagram (4h)
  - [19/11/2017] Individual Work: Component Interfaces Diagram (2h)
  - [20/11/2017] Individual Work: Component Diagram, Component Interfaces Diagram (2h)
  - [21/11/2017] Individual Work: User Interfaces Diagram, Database Design (4h)

- [24/11/2017] Individual Work: Mock up, Business Entities Diagram (4h)
- [25/11/2017] Team Reunion: Integration and Test plan discussion (3h)
- [25/11/2017] Individual Work: Mockups Design, Runtime View (7h)
- [26/11/2017] Individual Work:

- Luca Napoletano
  - [17/11/2017] Team Reunion: Software Architecture Design, Algorithm Design discussion (3h)
  - [18/11/2017] Team Reunion: Software Architecture Design, Algorithm Design discussion (6h)
  - [19/11/2017] Team Reunion: Algorithm Design discussion, Component Diagram (4h)
  - [19/11/2017] Individual Work: Component Interfaces Diagram (2h)
  - [20/11/2017] Individual Work: Component Diagram, Component Interfaces Diagram (2h)
  - [21/11/2017] Individual Work: User Interfaces Diagram, Database Design (4h)
  - [24/11/2017] Individual Work: Sequence Diagram (4h)
  - [25/11/2017] Team Reunion: Integration and Test plan discussion (3h)
  - [25/11/2017] Individual Work: Algorithm Diagrams, First Write of Architectural Design section (7h)
  - [26/11/2017] Individual Work: Completed Introduction and Architectural Design Overview, done Integration, Implementation and Testing Plan, Requirements Traceability and Document Review (10h)