POLITECNICO
MILANO 1863

Software Engineering 2 Project - Travlendar+

Design Document - V1

January 1, 2018

Authors:

Francisco Cristóvão
Samsom Tsegay Beyene

Contents

1 Introduction **3**
- 1.1 Purpose . . . 3
- 1.2 Scope . . . 3
- 1.3 Definitions, Acronyms, Abbreviations . . . 3
  - 1.3.1 Definitions . . . 3
  - 1.3.2 Acronyms . . . 4
  - 1.3.3 Abbreviations . . . 4
- 1.4 Revision History . . . 4
- 1.5 Reference Documents . . . 4
- 1.6 Document Structure . . . 4

2 Architectural Design **5**
- 2.1 Overview: High-level components and their interaction . . . 5
- 2.2 Component View . . . 7
  - 2.2.1 Server detailed analysis . . . 7
  - 2.2.2 Database detailed analysis . . . 9
- 2.3 Deployment View . . . 9
- 2.4 Runtime View . . . 10
  - 2.4.1 User Login . . . 11
  - 2.4.2 Create Appointment . . . 12
  - 2.4.3 Change Preference . . . 13
- 2.5 Component Interfaces . . . 14
- 2.6 Selected architectural styles and patterns . . . 15
  - 2.6.1 Client/Server Architecture . . . 15
  - 2.6.2 Service Oriented Architecture . . . 16
  - 2.6.3 Layered Architecture . . . 16
  - 2.6.4 3-Tier Physical Architecture . . . 16
  - 2.6.5 Model-View-Controller . . . 17
  - 2.6.6 Remote Facade . . . 17
  - 2.6.7 Data Mapper . . . 17
  - 2.6.8 REST . . . 17
  - 2.6.9 Other design decisions . . . 17

3 Algorithm Design **18**
- 3.1 Appointment Placement Algorithm . . . 18
  - 3.1.1 Fixed Appointment Placement Algorithm . . . 18
  - 3.1.2 Flexible Appointment Placement Algorithm . . . 20
- 3.2 Reachability Checker Algorithm . . . 20
- 3.3 Best Path Computation Algorithm . . . 21

4 User Interface Design **21**
4.1 Mockups 21
4.2 UX Diagram 22

5 Requirements Traceability **23**
5.1 Functional Requirements 23
5.2 Non-Functional Requirements 24

6 Implementation, Integration and Test Plan **25**
6.1 DBMS 25
6.2 Server 26
6.3 User Application 26

7 Effort Spent **27**

8 Used Tools **28**

9 References **28**

1 Introduction

1.1 Purpose

The main purpose of the Software Design Document (or just Design Document) is to provide a more technical and detailed description about the way Travlendar+ is designed and planned, identifying its main components and the interfaces between them. It also guides the software development team and other interested parties through the architecture of the software project, stating what has to be implemented and how to do it.

1.2 Scope

Travlendar+ is a calendar-based application that provides the user a convenient way of organizing his/her daily schedule, maximizing its productiveness and minimizing the worthless time of his/her day. This application was not only thought for the regular businessman/businesswoman, who travel in between meetings the whole day and have no time to spare, but also for the parents with a more regular daily schedule, who just want to get the best out of their time while being able to pick their kids from school and take them to other activities, always being on time. Of course the system will fully support the features of a regular calendar application (booking of appointments at a specific time and location), but in a "smart" way, being able to detect and warn the user if a new appointment is not feasible because it has a conflict (the start of it doesn't allow the needed travel time after the end of the last appointment) and arranging all of the appointments in the best possible way. The application is meant to be used in the City of Milan, and so it will take advantage of the wide range of travel means and services already existing in the city, from public transports to shared bikes and cars. With the information gathered from those services, it will be able to suggest the best travel mean for the user to move between appointments, based on the available travel time, total cost, current weather and even user preferences.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

Visitor: A person who uses Travlendar+ for the first time, and is not yet registered.
User: A person who uses Travlendar+.
System: defines the overall set of software components that implement the required functionality.

1.3.2 Acronyms

API: Application Programming Interface
DD: Design Document
ATM: Azienda Trasporti Milanesi
UX Diagram: User Experience Diagram
DB: Database
SQL: Structured Query Language
ER: Entity Relationship Model
MVC: Model-View-Controller
SOA: Service Oriented Architecture
REST: Representational State Transfer
JSON: JavaScript Object Notation
JSP: JavaServer Pages
EJB: Enterprise JavaBeans
JEE: Java Platform, Enterprise Edition

1.3.3 Abbreviations

1.4 Revision History

Version 1.1:

- Added a small description of the Algorithm to compute the best path to get to an appointment, referring to the documentation of the algorithm that shall be used.
- Added a small detail to the explanation of why we chose the Client/Server Architectural Style.

Version 1.0: Initial Release

1.5 Reference Documents

- Assignment document: Mandatory Project Assignments.pdf
- Requirements Analysis and Specification Document produced before

1.6 Document Structure

Other than this introductory chapter, this DD is organized in seven more chapters. Chapter two is meant to **provide different types of views over the system**:

- A high-level overview of how the system is architected.

- A description of the main components of the system, their structure and how they interact with each other.
- A description of the static deployment view of the system (how the components are deployed in the system's infrastructure).
- A description of the system's behavior and interactions in run-time conditions.
- A list of the selected architectural styles and patterns used in the design of the system, as well as the reasons that justify the choice of those patterns.

In the third chapter the most **relevant algorithms** are analyzed and discussed with the appropriate detail and depth, in order to describe the way the system's most critical operations are driven and executed.

The fourth chapter deals with the **user interface design**. This chapter mainly refers to the mockups provided in the RASD, but it will also include some details on the user interaction with the UI, illustrated by a UX Diagram.

The fifth chapter explains how the **requirements defined in the RASD are fulfilled by the design decisions** that were taken, and how these **requirements map** to the design elements and decisions defined in the DD.

In the sixth chapter, it is provided a **implementation, integration and test plan**, defining the order in which the different subcomponents of the system will be implemented, the order in which they will be integrated and how this integration will be tested alongside with the development of the system.

In the seventh chapter the **effort spent by each of the group members** is described by specifying the number of hours each member of the group worked on the development of this document and, on the final chapter, **the tools we used to develop this DD are specified**.

2 Architectural Design

2.1 Overview: High-level components and their interaction

In the following paragraphs it will be presented a general overview of how the system is architected, especially focused on the different logically separated layers. As described on the RASD, Travlendar+ is supposed to be fully scalable and portable, so a layered architecture is the one that best fits these requirements. Given that the system only provides an Application interface, there's no need for a fourth layer isolating the web server from the application server.

With this in mind, the system will have a three-layer architecture, organized as shown below:

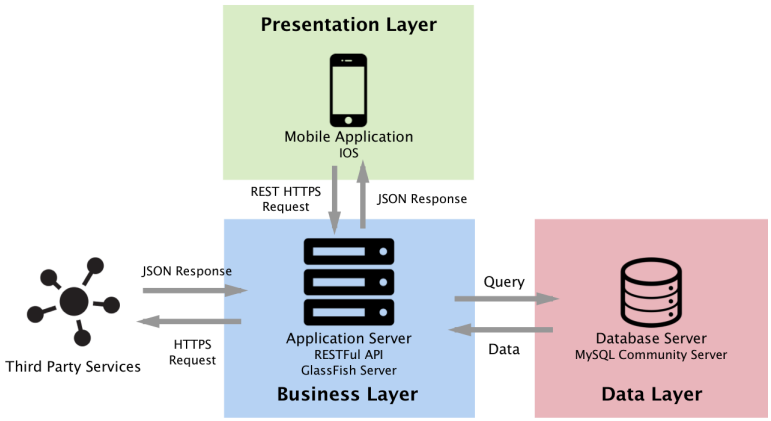

Figure 1: High Level View of the system's architecture

The **Presentation Layer** is the most external layer of the system, and is responsible for handling all GUI communication and logic. This layer does not handle data or process business rules, but it forwards all the requests to the layers bellow and translates the system operations results of these requests to something that users can understand. It is the only layer of the system that users can access directly and interact with.

The **Business Layer** implements all core functionality of the system. It's in this layer that the application logic and business rules are implemented, in particular, all the operations related to a user account and the appointment creation and management are performed by components of this layer. This layer interacts with the APIs exposed by the Data Layer in order to store and retrieve data. The business layer also depends on some third party systems and the external services they provide (specifically for the implementation of the appointment creation and management and travel mean functionality). These external services are directly invoked by some of the classes of the Business Layer using a public API provided by those.

At last, the **Data Layer** is the lowest layer of the architecture and includes the data persistence mechanisms responsible for data storage and management. It also provides an API to the Business Layer that exposes methods of managing the stored data without creating dependencies on the data storage mechanisms, promoting the encapsulation of the persistence mechanisms and avoiding data exposition.

Even though the **Third Party Services** don't belong to any particular layer, these services are illustrated in the figure above in order to highlight that the interaction with these will happen at the level of the Business Layer.

2.2 Component View

The main function of this section is to present a more detailed description of the components that must be developed as part of Travlendar+.

In the following diagram, we can see the component view of the system (the more complex components will be analyzed in further detail).

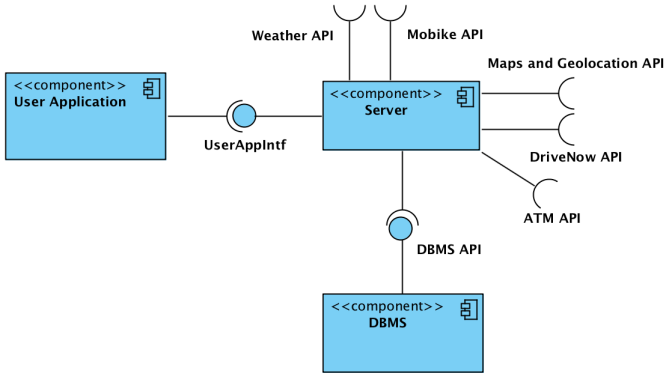

Figure 2: Component Diagram of the system

The **User Application** is the component responsible for providing the user interfaces that allow the user to interact with the system and all of its functionality. Since a thin-client architecture is being used to implement a client/server architecture, the bulk of the data processing occurs on the server. Given this, the User Application component will act as a simple "terminal" to send requests to the server, being therefore quite simple and not needed to go into further detail.

The **Server** is the main component of the system, responsible for the data processing in order to provide all of the system's functionality. Given its complexity, it will be analyzed in further detail later in this section.

The **Database Management System (DBMS)** is the component responsible for storing and retrieving data in a persistent and reliable way. Instead of implementing this component from scratch, a commercial solution shall be used (MySQL for example).

2.2.1 Server detailed analysis

Being the most complex component of the system, it is useful to analyze in detail the Server's constitution.

As mentioned previously, this component is primarily responsible for all operations related to system functionality. More specifically, this component is divided into six sub-components, each related to a different kind of functionality and operations:

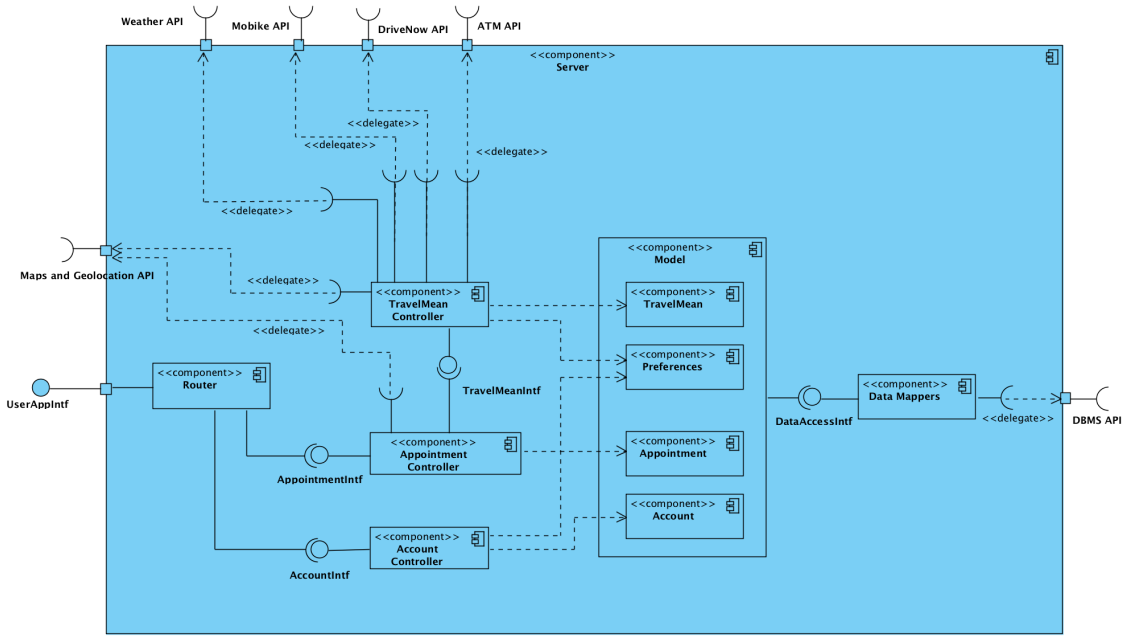

Figure 3: View of Server Component in detail

- The **Router** sub-component receives the requests from the User Application and forwards (routes) them to the corresponding controller.

- The **Travel Mean Controller** sub-component performs all the necessary operations to compute the best travel means to get to an appointment, given the schedule at that time. This module is the one with more external interfaces since it needs to gather data from the transport services and weather services in the city of Milan in order to perform its functions.

- The **Appointment Controller** sub-component performs all the necessary operations that allow the user to create, edit and delete an appointment. It is also responsible for all the needed procedures that allow the user to always have the best possible schedule: schedule and arrange the appointments according to the parameters the user defines for each appointment and based on the time the user needs to get from one appointment to another (data computed by the Travel Mean Controller, which explains the interface to this component). This, allied with the Travel Mean Controller, fulfills all the requirements related to the appointment creation and managing.

- The **Account Controller** sub-component implements all the methods for inserting or updating information about the users. More specifically, it allows the creation of new users (registration of visitors) and login of already existing

users. It also performs all the necessary operations to allow the user to edit all of his preferences related to the travel means.

- The **Model** sub-component (and all of its sub-components) manages the behavior and data of the application domain and responds to requests or instructions by the respective controller.
- The **Data Mappers** sub-component is a layer of software which separates the model from the database and acts as a "middle-man" for the interactions between these two.

2.2.2 Database detailed analysis

The following ER provides a graphical representation of the conceptual model of the database.

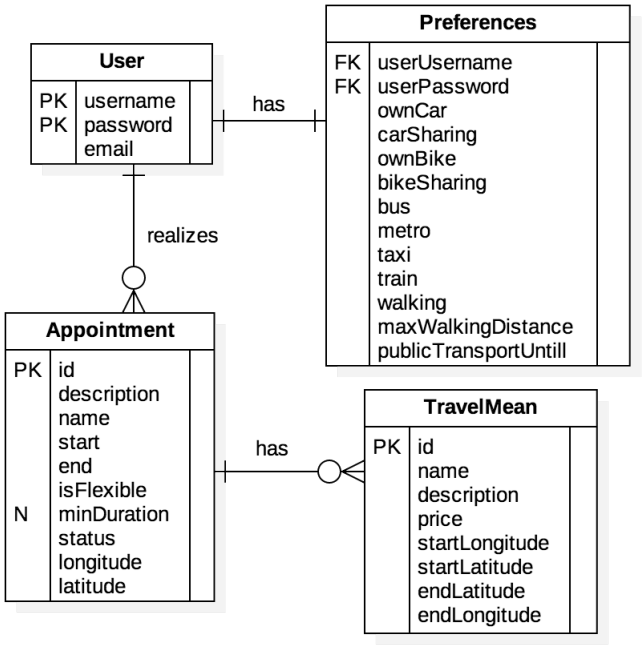

Figure 4: Database ER

2.3 Deployment View

The main purpose of this section is to show how the software components described above are going to be deployed in the system's hardware infrastructures. This char-

acteristic makes the developer's task easier since the mapping between software and hardware becomes explicit.

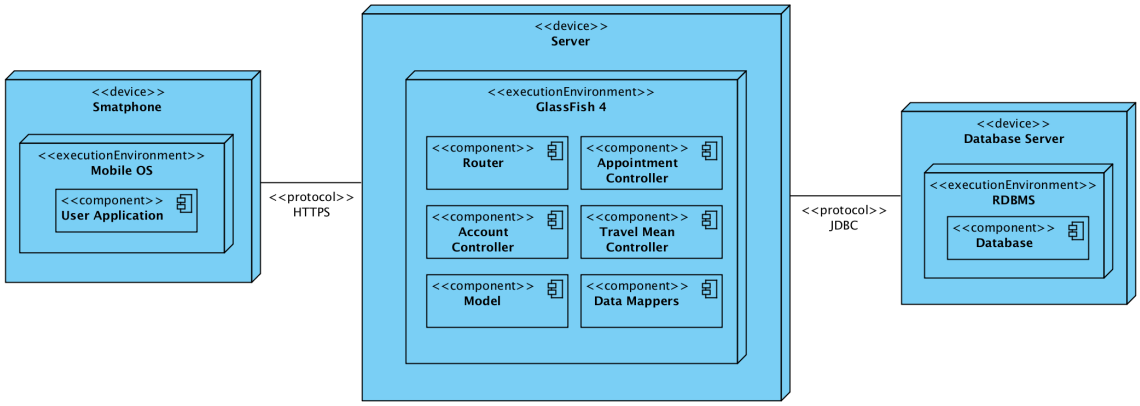

Figure 5: Deployment View of the system

2.4 Runtime View

The following diagrams are intended to describe the interactions between the main system components when performing a sample amount of functionality. These diagrams are still a high-level view of the system, since the function names or even the functions themselves may change during the development process.

2.4.1 User Login

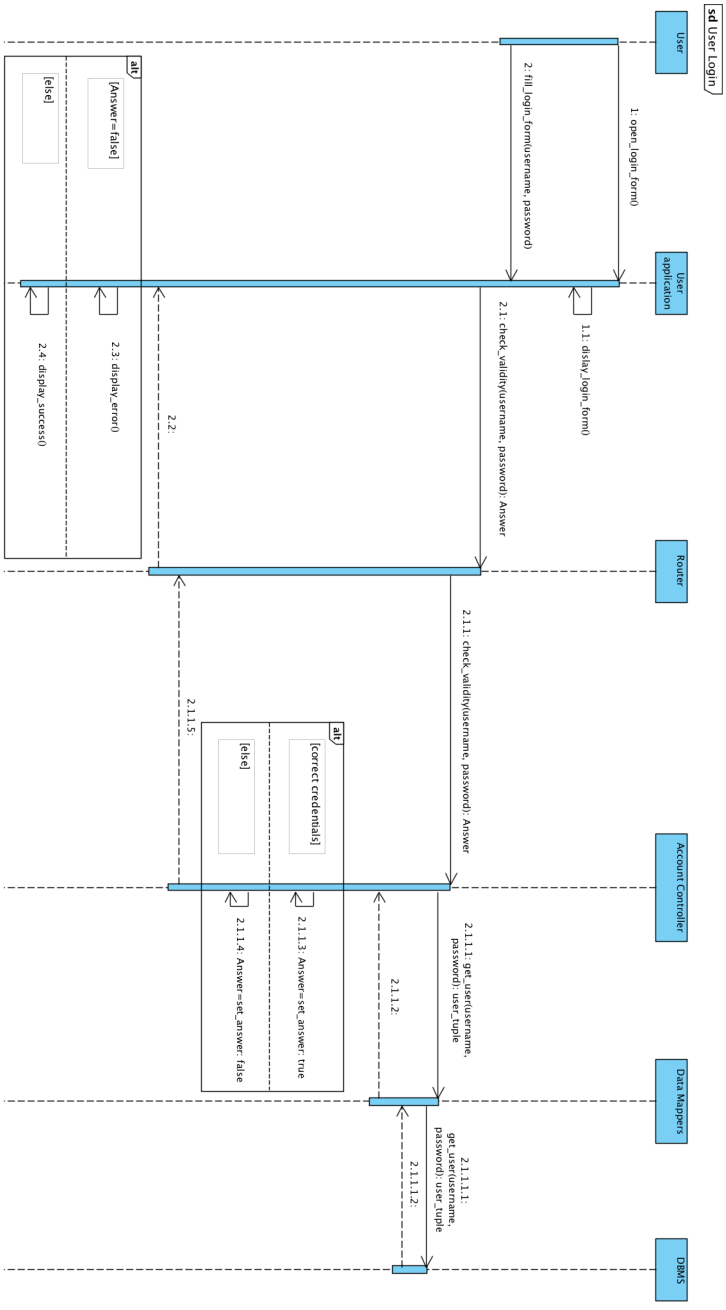

Figure 6: User Login

2.4.2 Create Appointment

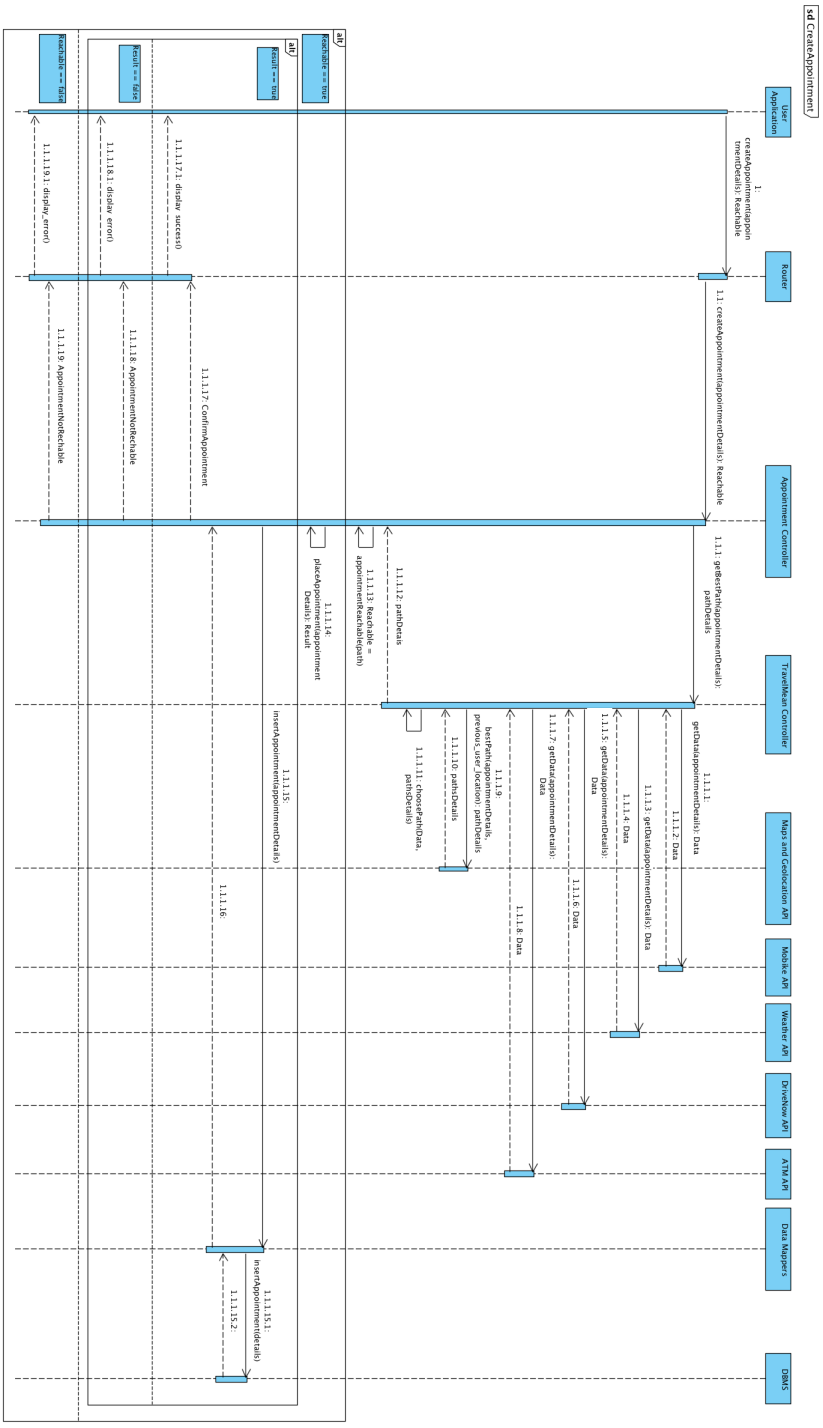

Figure 7: User Appointment Creation

2.4.3 Change Preference

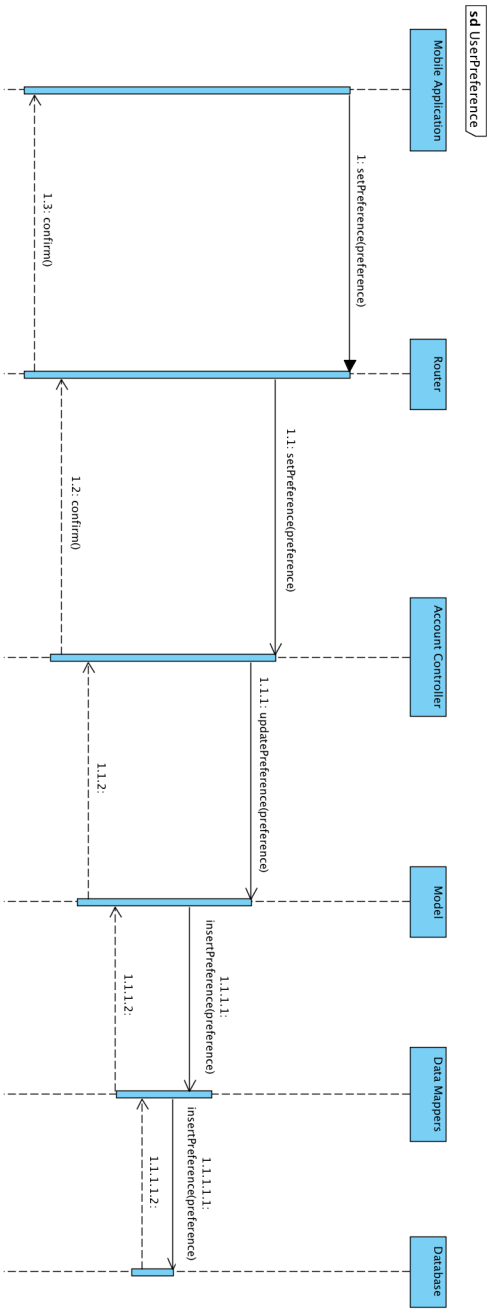

Figure 8: Change Preference

2.5 Component Interfaces

In this section the interfaces through which the different system's components interact will be described, and so will the main methods associated with each interface

- The **UserAppIntf** interface represents the way the User Application interacts with the Server, being this the only interface through which these two can communicate. This interface provides all of the methods provided by the **AppointmentIntf** and the **AccountIntf**.
- The **AppointmentIntf** interface represents the way components can interact with the Appointment Controller, responsible for managing all of the aspects associated with the appointment management. This interface provides the following methods:
  - createAppointment(string: name, string: description, date: from, date: to, float: longitude, float: latitude): receives the necessary data to create an Appointment and creates it.
  - createFlexibleAppointment(string: name, string: description, date: from, date: to, time: minDuration float: longitude, float: latitude): receives the necessary data to create a Flexible Appointment and creates it.
  - editAppointment(int: appointmentId, data): receives the data parameters related to an appointment that are supposed to be changed and the id of that appointment. The data parameters are the ones described above.
  - deleteAppointment(int: appointmentId): deletes the appointment that matches the appointment Id.
  - checkReachibility(appointmentDetails): checks if the appointment to be created/edited is reachable or not.
- The **AccountIntf** interface represents the way components can interact with the Account Controller, responsible for managing all of the aspects associated with the user account. This interface provides the following methods:
  - createAccount(string: username, string: email, string: password): registers a visitor into the system, creating an account for him.
  - loginUser(string: username, string: password): logs a user into the system.
  - recoverPassword(string: email): sends a new password to the user email, deleting the previous one.
  - setPreference(string: newPreference): changes a user preference to its new value.

- The **TravelMeanIntf** interface represents the way components can interact with the Travel Mean Controller, responsible for managing all of the aspects associated with the travel means used to get to the appointments. This interface provides the following methods:
  - getBestPath(appointmentDetails): receives the data parameters related to the appointment in question and returns the best travel mean to get to that appointment.
- The **DataAccessIntf** interface represents the way components can interact with the Data Mappers Component. It provides all of the needed methods and mechanisms to abstract the interaction between the database and the system.
- The **Maps and Geolocation API** interface represents the way components can interact with the Google Maps and Geolocation service. It provides all the needed methods to get information about the appointment location and the best routes and travel means to get to that location.
- The **Weather API** interface represents the way components can interact with the Weather Service. It provides all the needed methods to get information about the weather forecast in a specific place and time.
- The **Mobike API** interface represents the way components can interact with the Mobike service. It provides all the needed methods to get information about the bicycle's location and cost.
- The **DriveNow API** interface represents the way components can interact with the DriveNow service. It provides all the needed methods to get information about the cars location, status, and cost.
- The **ATM API** interface represents the way components can interact with the ATM service. It provides all the needed methods to get information about the travel mean schedule.
- The **DBMS API** interface represents the way components can interact with the Database. This interface provides the usual methods that are needed to interact with a database, like update, insert, delete and select queries.

2.6 Selected architectural styles and patterns

2.6.1 Client/Server Architecture

This architectural style keeps all the logic of the program, which is computationally heavy, on the server side, while the end-users only need to offer a simple amount of presentation functionality. This allows the system to be centralized, improving:

- Performance: system's files are only accessed by the server, so they're all in the same place, becoming easier to manage and faster to access.
- Security: facilitates the implementation of security layers and protocols between the client and the server, setting up access rights to reach the server for example.
- Scalability: it's easy to change the logic server-side without any implication on the client side. Also true the other way around: It's easy to change the client implementation without changing the logic server-side, making it easily portable to different platforms (like a desktop application, for example).

2.6.2 Service Oriented Architecture

This architectural pattern allows the system to be designed as a collection of services that communicate with each other. This way, the external APIs that the system interacts with can be interpreted as a set of these services, which makes them easily implementable (in this case our system is the service consumer, and the APIs are the service providers). It is also possible to see our system as the service provider, first providing an initial set of services and eventually, with the addition of some additional components, increasing the number of services provided.

2.6.3 Layered Architecture

This architectural pattern promotes a great level of separation of concerns between the various components of the system, given that every component operates at a single logical layer of abstraction. This design choice also simplifies the implementation and testing phase of the system: since the layers are isolated from each other, it is possible to test a specific layer even if the underlying layers aren't implemented, using mocks or stubs to simulate the functionality of those.

2.6.4 3-Tier Physical Architecture

This architectural style allows the various components of the system to be deployed on different devices, given the decoupling that exists between the modules. It also allows for any of the three tiers to be upgraded or replaced independently in response to changes in requirements or technology, improving the system's scalability. The system will be composed of 3 tiers: Presentation tier, Business tier (the business logic of the system) and Data tier.

2.6.5 Model-View-Controller

This architectural pattern divides a given application into 3 interconnected parts, separating the internal representation of information from the way this information is presented to the user. This is one of the most used patterns to promote decoupling between the major components of a system, allowing for efficient code reuse and parallel development.

2.6.6 Remote Facade

This architectural pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. Given that the communication between the different components of our system will happen over the network, the existence of a Facade with coarse-grain methods that map to various fine-grained ones also improves the communication's efficiency (fewer packets have to be sent over the network). With all this, it promotes decoupling between the server and the user application, since all the remote calls are made through one interface and then mapped inside the server (by the router component) to the respective controller.

2.6.7 Data Mapper

This architectural pattern is a data source architectural pattern. It consists of a layer of software that separates the domain layer objects from the database. Its responsible for transferring data between the two while keeping them isolated from each other and from the mapper itself.

2.6.8 REST

A REST architectural style with JSON data model format is also being used to allow the communication between the User Application and the Server, being this based on HTTP requests (POST, GET, PUT and DELETE to name the most important ones).

2.6.9 Other design decisions

- **Application Server:** GlassFish 4.1 was chosen since it provides containers for both the EJB and the JSP pages, and is a full Java EE Application server.
- **DMBS:** MySQL was chosen as DBMS since it's free and one of the most popular DBMS, with a big community and lots of documentation.

3 Algorithm Design

In this section, it is presented an overall description of the most relevant algorithms for the Travlendar+ system, explained either using pseudo-code notation or refering to the relevant documentation about it.

3.1 Appointment Placement Algorithm

The first and second algorithms check if it is possible to insert a new Fixed Appointment and a new Flexible Appointment, respectively, in the user's schedule and, if possible, inserts it.

3.1.1 Fixed Appointment Placement Algorithm

If this new appointment is a fixed one, there are three possible situations:

- The time interval of the appointment is free - the new appointment is successfully inserted in the system
- The time interval of the appointment isn't free, but the system manages to reschedule an adjacent flexible appointment, leaving enough space for this appointment to be scheduled - the new appointment is successfully inserted in the user's schedule.
- There is no way of inserting that appointment into the user's schedule - the system returns an error message.

```
begin
    N ← NewFixedAppointment;
    if N_timeInterval = free then
        insert(N);
        return True;
    if FlexibleAppointmentsIn(N_end − N_start) = 1 then
        if FlexibleAppointment before N_start and
         PreviousFixedAppointment_end − N_start >
         FlexibleAppointment_minDuration +
         TimeToGetToFlexibleAppointment + TimeToGetToN and
         possibleInterval(FlexibleAppointment, PreviousAppointment_end, N_start)
         then
            relocate(FlexibleAppointment);
            insert(N);
            return True;
        else if FlexibleAppointment after N_start and
         NextFixedAppointment_start − N_End >
         FlexibleAppointment_minDuration +
         TimeToGetToFlexibleAppointment +
         TimeToGetToNextFixedAppointment and
         possibleInterval(FlexibleAppointment, N_end, NextAppointment_start)
         then
            relocate(FlexibleAppointment);
            insert(N);
            return True;
    else if FlexibleAppointmentsIn(N_end − N_start) = 2 then
        if FlexibleAppointment1 before N_start and
         PreviousFixedAppointment_end − N_start >
         FlexibleAppointment1_minDuration +
         TimeToGetToFlexibleAppointment + TimeToGetToN and
         possibleInterval(FlexibleAppointment1, PreviousFixedAppointment_end, N_start)
         and FlexibleAppointment2_start after N_end and
         NextFixeAppointment_start − N_end >
         FlexibleAppointment_minDuration +
         TimeToGetToFlexibleAppointment +
         TimeToGetToNextAppointment and
         possibleInterval(FlexibleAppointment, N_end, NextAppointment_start)
         then
            relocate(FlexibleAppointment1);
            relocate(FlexibleAppointment2);
            insert(N);
            return True;
    return False
end
```

Algorithm 1: Fixed Appointment Placement

3.1.2 Flexible Appointment Placement Algorithm

If this new appointment is a flexible one, there are two possible situations:

- There is a free space in the user's schedule that fits the given appointment time interval and is at least as big as the flexible appointment's minimum duration plus the time to get there - the new appointment is successfully inserted in the system.
- There is no way of inserting that appointment into the user's schedule - the system returns an error message.

begin
  $N \leftarrow NewFlexibleAppointment$;
  if $N_{timeInterval} = free$ **then**
    $insert(N)$;
    $returnTrue$;
  if $FixedAppointmentsIn(N_{end} - N_{start})$ **then**
    if $AppointmentAfter_{start} - AppointmentBefore_{end} >$
     $N_{minDuration} + TimeToGetToN + TimeToGetToAppointmentAfter$
     then
      $insert(N)$;
      $return\ True$;
  $returnFalse$;
end

Algorithm 2: Flexible Appointment Placement

3.2 Reachability Checker Algorithm

This third algorithm checks if an appointment is reachable or not. Given a schedule state and a new appointment to be inserted, it computes the time interval between the previous appointment end time and the starting time of the appointment to be inserted and, if this interval is smaller than the required time to get from the previous location to the new appointment's location, the system returns that the appointment is reachable. Otherwise, it is unreachable.

Bellow the pseudo code for this algorithm is presented, in order to better understand it:

```
begin
    L2 ← NewAppointmentLocation;
    if PreviousAppointment then
        L1 ← PreviousAppointmentLocation;
        T1 ← PreviousAppointment_endTime;
    else
        L1 ← UserLocation;
        T1 ← 0
    end
    T2 ← NewAppointment_startTime;
    ΔT ← T2 − T1;
    T3 ← TimeToGetFromOldAppointmentToNew;
    if ΔT ≥ T3 then
        return True;
    else
        return False
    end
end
```

Algorithm 3: Reachability Checker

3.3 Best Path Computation Algorithm

To compute the best path to get to an appointment, the system can either use the result returned by the Google Maps and Geolocation API (if all the user settings are supported) or compute it itself (if the user has the bike and car sharing settings enabled, since these are not supported by the Google Maps and Geolocation API).

If this is the case, the system will get the required information from that API and compute the path itself, using the A* Algorithm[1]. After it, the system will choose the best one between these two paths (one returned by Google Maps and Geolocation API and the other computed with A*) and return the best one to the user.

4 User Interface Design

4.1 Mockups

One of the most important things to take into consideration when designing an information system is the way its functionality is going to be provided to the user. Since the user interacts with our system through a GUI, saying this is the same as saying that this user interface has to be as simple and intuitive to use as possible.

[1]Documentation can be found in the References section

Another thing that was taken into consideration while designing the interfaces was the adaptability of it to the different smartphone screen resolutions and form factors available on the market. To satisfy this requirement, the interface was drawn in a minimalist way, without demanding animations and small details that would make it impossible to use on smaller screens with low resolution.

The mockups of the UI of our system can be found in **section 3.1.1 of the RASD** and were designed with all this in mind: to be as simple and intuitive to use as possible while providing to the user all of the system's functionality.

4.2 UX Diagram

The following UX Diagram provides additional information about the way the user interacts with the system through the mobile application, and have the purpose of complementing the UI mockups provided in the RASD document with a more "dynamic" view over the interface.

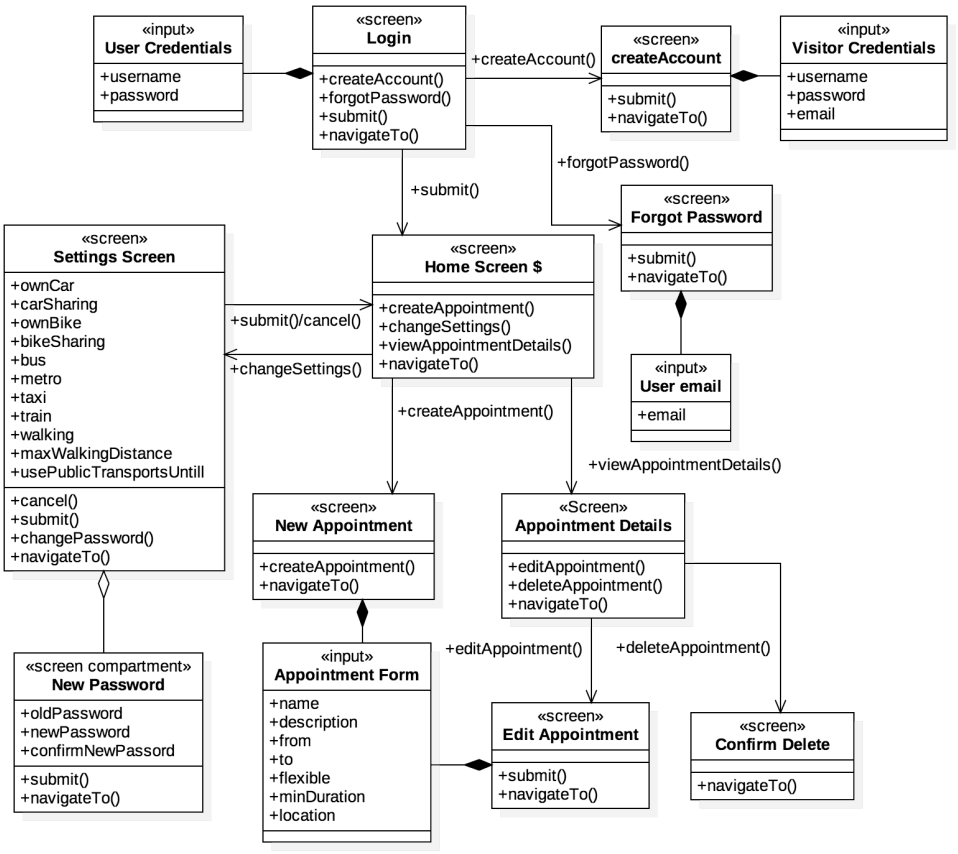

Figure 9: UX Diagram

5 Requirements Traceability

5.1 Functional Requirements

All of the design choices presented in this document were made with the goals and requirements defined in the RASD in mind, and aim to fulfill them in the best possible way. The following list provides an accurate mapping between the system's goals and requirements defined on the RASD and the system components, defined in the DD, that aim to fulfill those goals and requirements.
N.B.: All the requirement intervals include the first and the last on the interval.

- [**G1**] Allow only registered users to access the most relevant functionality of the system (R1-R4).
  - The **Account Controller**, which provides the visitors the ability to register as users of the system.
- [**G2**] Allow a logged user to create an appointment (R5-R10).
  - The **Appointment Controller**, which provides all of the business logic related to the appointments management, more specifically the ability to create a new appointment.
  - The **Travel Mean Controller**, which communicates with the third party services the system uses and implements all of the needed logic to compute for the best travel means to get to an appointment.
- [**G3**] Allow a logged user to edit an appointment (R6-R7 and R11-R13).
  - The **Appointment Controller**, which provides all of the business logic related to the appointments management, more specifically the ability to edit a previously created appointment.
  - The **Travel Mean Controller**, which communicates with the third party services the system uses and implements all of the needed logic to compute for the best travel means to get to an appointment.
- [**G4**] Allow a logged user to delete an appointment (R14).
  - The **Appointment Controller**, which provides all of the business logic related to an appointments management, specifically to delete an appointment from the user's schedule.
- [**G5**] Allow a logged user to have his schedule planned in the best possible way (R6 and R15-R19).

  - The **Appointment Controller**, which provides all of the back end logic related to the appointments management, more specifically to make any change in the calendar related to insertion or deletion of appointments.
  - The **Travel Mean Controller**, which communicates with the third party services the system uses and implements all of the needed logic to compute for the best travel means to get to an appointment.

- [**G6**] Allow a logged user to define preferences related to travel means (R20-R24).
  - The **Account Controller**, which provides the visitors the ability to manage its account preferences and change them according to his personal preferences.

For all of the requirements the **Router Component** takes part in the interaction, but only maps the User Application requests to the respective controller, not being relevant to refer to in every goal. So does the **Model Component**, the **Data Mappers Component** and the **Database Component**, since our application has to access the database (either to read or to write) every time it executes one of the main system operations/functionality.

5.2 Non-Functional Requirements

The non-functional requirements were no exception and were also taken into consideration when making the Design Choices for the system. In particular, this analysis will be focused on the **Performance, Reliability, Availability, Security**, and **Portability** requirements.

The usage of a 3 tier infrastructure is the key to achieve most of these requirements. This infrastructure allows us not only to achieve high levels of **Performance** (components with different hardware needs can be deployed in more powerful machines) but also satisfying levels of **Reliability** and **Availability** as, thanks to the low coupling between the different components, it allows them to be distributed over the network, possibly replicated and in different server farms (highly increasing fault tolerance).

Security is achieved thanks to software and hardware steps. At the software level, all interface methods check if the call parameters are valid before executing its main logic and if the caller is authorized to invoke them. The database APIs will also be protected against SQL Injection and all messages between different components of the system and external APIs will be encrypted and have unique identifiers, encrypting this way all of the critical data and preventing man-in-the-middle attacks (more specifically replay attacks).

At last, **Portability** is achieved by offering a standard web service mechanism based on JSON through the API's. This allows the system to be invoked from any platform with internet access, not restricting the system to a specific operating system, programming language or hardware brand.

6 Implementation, Integration and Test Plan

Implementation, Integration, and Testing are 3 different activities that must be done together in order for a project to be developed in a correct way. It will be presented in the paragraphs below a plan for the implementation, integration, and testing of the different components that, together, form our system.

The system is divided into layers, and as in a traditional layered architecture, the topmost layers depend on the lowest ones. This being said, the system must be implemented using 1 out of 3 different approaches:

- Bottom-Up Approach
- Top-Down Approach
- Mix between Bottom-Up and Top-Down, which meet somewhere in the middle

The **second and third** approaches imply that a component will be implemented and tested without the need for the components that this depends on to be implemented and tested already. Although this might look like a good idea, it carries a more complex development process, since mocks would have to be built in order to test the behavior of these components after being implemented. Taking this into consideration, the **approach chosen to develop our system was a bottom-up approach.** This way, the components which do not depend on others will be the first ones to be developed and extensively tested, followed by the ones that only depend on those, until we reach the highest layer of the system, with the components that have no other component depending on them. Thanks to this a component can be tested and integrated as soon as it's finished (or even close to being finished), allowing us to complete the development and testing phase almost in parallel. We must also keep in mind that in order to test the integration of two components, the main features of both of them should have already been developed and the related unit tests should have been performed.

6.1 DBMS

The first component to be implemented is the DBMS. It shall be implemented and then fully tested before finishing the implementation of the layer above since it has

to be configured and operative in order to allow to test all of the components which need access to the database.

6.2 Server

The next set of components to be implemented and tested are the ones that belong to the Business Layer, which encapsulates the business logic of the system. This layer is composed of the Server component and all of its sub-components.

First, we should start with the **Data Mappers** sub-component, which is the Server component that mediates the interaction between the server and the database, and only depends on the DBMS component.

After this, and following a critical-module-first approach, we should develop the **Appointment Controller and Travel Mean Controller** and the **Model components associated with this ones**. Both of this controllers require the interaction with the external services to be fully functional.

The last sub-components of the server to be developed will be the **Account Controller and Router Controller** and the **Model components associated with this ones**.

After being fully implemented, the Server component must be integrated with the DBMS component, as it needs the DBMS to work properly.

Before integration testing of the two integrated components, the used external services shall be configured and fully operative. After this, the Server can be integrated with the DBMS and shall be extensively tested, as it is the most critical and important part of the system.

6.3 User Application

The final part of the system to be implemented is the User Application, the topmost layer of the architecture. Since we opted for a thin-client this shall be one of the easiest and fastest steps of the development process. When finished, it shall be integrated with the rest of the components of the system in order to test the system as a whole.

7 Effort Spent

DATE	TASK	HOURS
08/11/2017	Scope, Purpose and Document Structure	1
09/11/2017	Overview and Architectural Styles and Patterns	3
10/11/2017	High Level Components description and Database Component View	4
11/11/2017	Component View and Architectural Styles and Patterns	3
12/11/2017	Component View and Architectural Styles and Patterns	1,5
13/11/2017	Component View and Deployment View	4
14/11/2017	Component View, Architectural Styles and Patterns and Component Interface	3
15/11/2017	User Interface (UX Diagram)	2
17/11/2017	Component diagram and Architectural Patterns	1,5
18/11/2017	Architectural Patterns and Component View	2
19/11/2017	Component Interfaces and Requirements Traceability	4
20/11/2017	Runtime View and Requirements Traceability	3
21/11/2017	Runtime View and Algorithm Design	4
22/11/2017	Algorithm Design	2
23/11/2017	Algorithm Design and Runtime View	1
24/11/2017	Implementation, Integration and Test Plan	2,5
25/11/2017	Final DD details	1,5
20/12/2017	Overall Improvements: creation of version 1.1	1
TOTAL	44	

DATE	TASK	HOURS
08/11/2017	Purpose and Document Structure	1
09/11/2017	High Level Component description	1
11/11/2017	Component View	2
14/11/2017	Runtime View	3
15/11/2017	User Interface (UX Diagram)	2
16/11/2017	Architectural Patterns	1
18/11/2017	Runtime View	3
19/11/2017	Algorithm Design	2
20/11/2017	Requirements Traceability	1
22/11/2017	Implementation, Integration and Test Plan	2
24/11/2017	DD Review	1,5
TOTAL	19,5	

8 Used Tools

- ShareLateX
- StarUML 2.8.0
- Sketch 47.1
- Visual Paradigm 14.2

9 References

References

[1] https://www.safaribooksonline.com/library/view/software-architecture-patterns/9781491971437/ch01.html

[2] https://martinfowler.com/eaaCatalog/index.html

[3] Fowler, Martin. *Patterns of Enterprise Application Architecture, 1st Edition*

[4] https://en.wikipedia.org/wiki/A*_search_algorithm

[5] http://web.mit.edu/eranki/www/tutorials/search/