POLITECNICO
MILANO 1863

Travlendar+ project

Design Document

RICCARDO FACCHINI

ANDREA GUGLIELMETTI

November 12, 2017

Deliverable specific information

Deliverable:	Design Document
Title:	Requirement Analysis and Verification Document
Authors:	Riccardo Facchini - Andrea Guglielmetti
Version:	1.0
Date:	November 12, 2017
Download page:	https://github.com/Riccardo95Facchini/FacchiniGuglielmetti.git
Copyright:	Copyright © 2017, Riccardo Facchini - Andrea Guglielmetti – All rights reserved

Contents

Copyright © 2017, Riccardo Facchini - Andrea Guglielmetti – All rights reserved

List of Figures

List of Tables

1 Introduction

1.1 Purpose

This document aims to detail the design of the software and of the architecture regarding the system of Travlendar+. To do so it will be taken a more detailed approach for the description of each component and the overall architecture of the system, by also pointing out the relations between each module and giving a description of such relationships.

1.2 Scope

Travlendar+ is a time/travel management web-based system, designed to help the users to keep track of their daily routine by scheduling for them the best way to move from one place to the other using all the information given by the users themselves and external data in order to deliver a tailored experience for everyone.
After the user enters all the needed data to register an event, the system will automatically alert him/her when it's time to leave and will give directions to reach each one of the means of travel as specified in the options, taking into account also factors like the weather and possible public transportation strikes.

1.3 Definitions, Acronyms, Abbreviation

- DD : Design Document
- RASD: Requirement Analysis and Specification Document
- API: Application Programming Interface
- DB: DataBase
- DBMS: DataBase Management System
- GPS: Global Position System

1.4 Revision History

1.5 Reference Documents

- Document of the assignment: Mandatory Project Assignments.pdf
- Requirements and Specification Document

1.6 Document Structure

2 Architectural Design

2.1 Overview

We need to design a system in which the user asks to the system to store an appointment and calculate the best path from a starting location to the appointment location.
Since this interaction between user and system can be summarize as:

1. User request a service to the system.
2. System responds to the user with the requested service.

Based on this, we decide to use a client-server architectural approach.

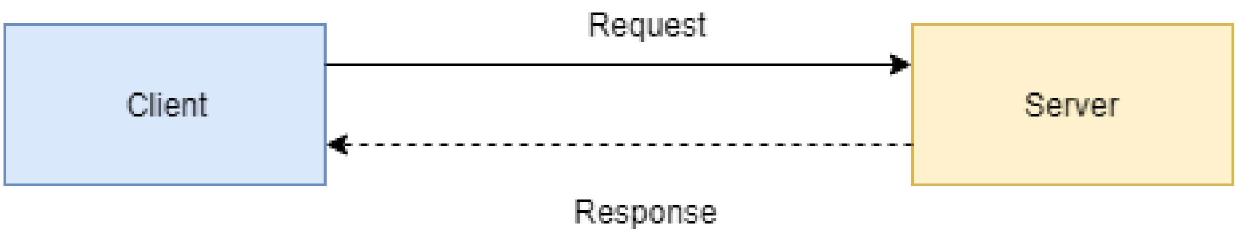

Figure 1: Client Server architecture

Furthermore, the system can be divided into three different subsystems: the presentation layer, the application layer and the data layer as we can see in Figure 2.

- The *Presentation Layer* provides the GUI of the system. This layer contains the mobile application and the web pages.
- The *Application Layer* contains the logic of the application,that receives the requests from the user, computes the best path to reach the appointment, checks the weather and the road conditions and executes the dynamic web pages of the web site.
- The *Data Layer* stores and maintains the data needed from the system to works properly, i.e. user's information and user's appointment information.

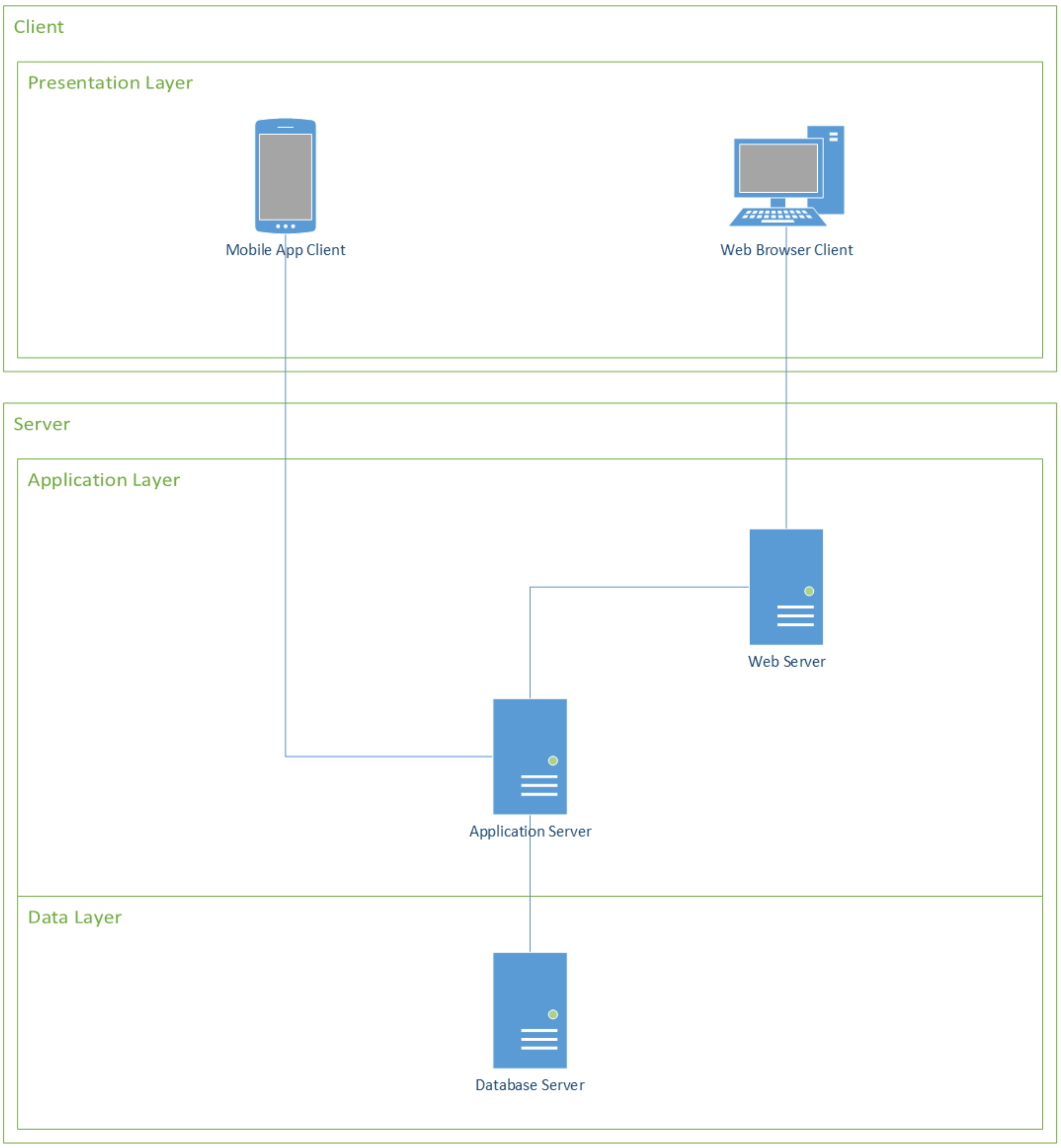

Figure 2: Overview of the system architecture

2.2 Component View

2.2.1 Overview

In Figure 3 is possible to see the high level components of the system and the interfaces used to connect one to another, where

- the *DBMS* provides the database and a way to retrieve data from it;
- the *Application Server* provides the main logic of the application;
- the *Web Server* provides the static pages and executes the dynamic pages of the web site.
- the *Mobile Application* is the mobile application used by a user with his/her smartphone.
- the *Web Application* is the application that runs on the user's browser.

2.2.2 Database View

The DBMS component provides a database and is DBMS for data storage and their management. It is possible to access the database only through the Application Server and an appropriate secure interface. For security and privacy reasons, data are encrypted inside the database. The Entity-Relationship diagram of the database is showed in Figure 4.

2.2.3 Application Server View

The *Application Server* contains the main logic of the application. It receives the user's request and interacts with the database to store and retrieves data. The *Application Server* as we can see in Figure 5 is composed of:

- **Authentication Manager**, it manages the request of a user to register or to login into the service. It can access to *Account Data Manager* to retrieves user's information from the database.
- **Profile Manager**, it manages the request of a user to update his/her profile. It can access the *Account Data Manager* in order to retrieves information in the database.
- **Account Data Manager**, it can access all the information about the user's account in the database.
- **Appointment Manager**, provides to the user the functionalities of creation / modification of appointments. It uses the *Path Calculator* to obtain the best path for the appointment and the *Appointment Data Manager* to stores and retrieves information.
- **Path Calculator**, it is responsible to compute the best path from the starting location defined by the user and the appointment location. To do so, it can access the *Additional Info Facade* to retrieves the user preferences, the weather and road informations. It needs also the *Google Maps API* to retrieves distance and time informations.
- **Weather Information Manager**, it manages weather information retrieving it from an external system via its API, showed in the diagram as *Weather API*.
- **Road Information Manager**, it manages road information retrieving it from an external system via its API, showed in the diagram as *Road API*.
- **Additional Info Facade**, it is a component that implements the *Facade Pattern*, in this way it is possible to reduce the coupling between the *Path Calculator* and the other interfaces from that needs to get information required.

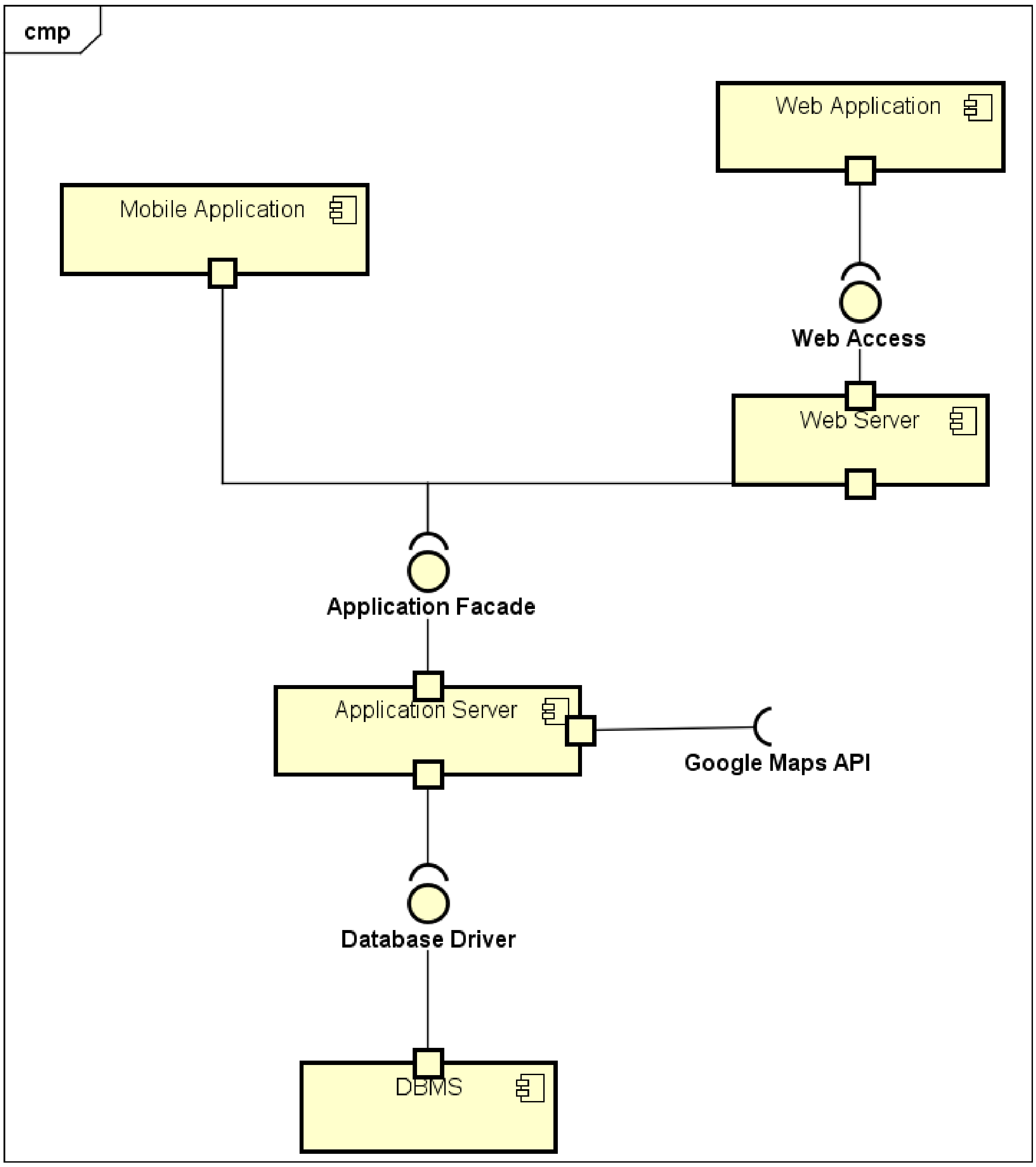

Figure 3: High level Component Diagram

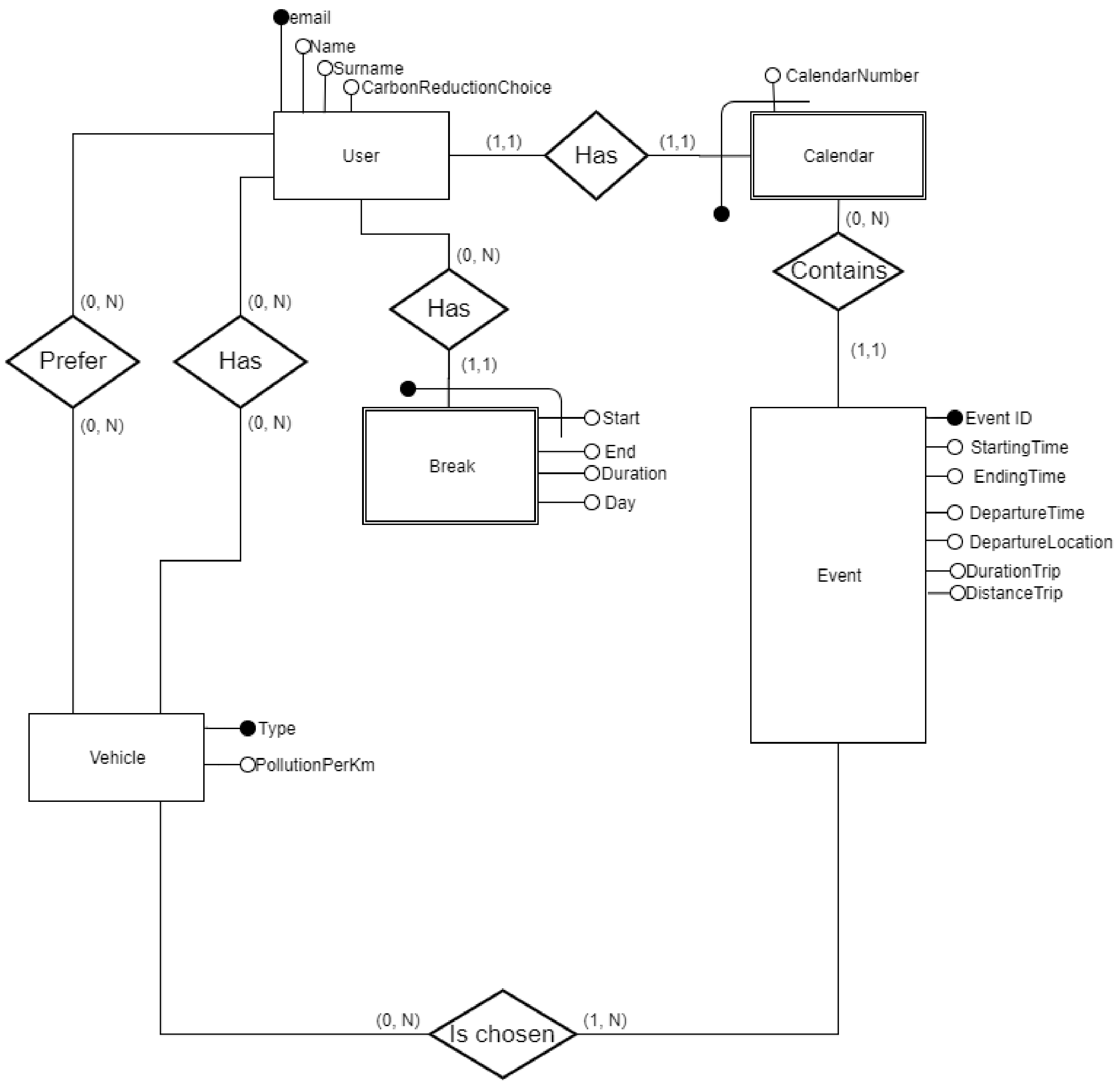

Figure 4: *Entity-Relationship Diagram* of the database

- **Travlendar+ Facade**, it is a component that implements the *Facade Pattern* and provides a common interface for both the *Mobile Application* and the *Web Server*.

2.2.4 Web Server

Since the presence of a *Web Application*, it is necessary a dedicated *Web Server* responsible to executes the web site's dynamic pages and provides the static pages to the user's browser. The *Web Server* interacts with the *Application Server* to get the proper information to fill up the pages. The *Web Server* also sends data from the user's browser to the *Application Server* to store inside the database.

2.2.5 Mobile Application

The *Mobile Application* is used by the user via its own smart device. The *Mobile Application* communicates directly the *application server* with a dedicated communication protocol. The component diagram of the *Mobile Application* is showed in Figure 6. The description of the components is the follow:

- **User View** is responsible of the graphical representation of the app and the interactions with the user.
- **GPS Manager** is responsible to interact with the GPS Module of the smart device.
- **DBMS**, is a physical view of the main database while storing only the current user's data. It is used by the *Controller* to notify the user when an event is about to start.
- **Controller**, is responsible to interact with the *Application Server* and link together the other components.

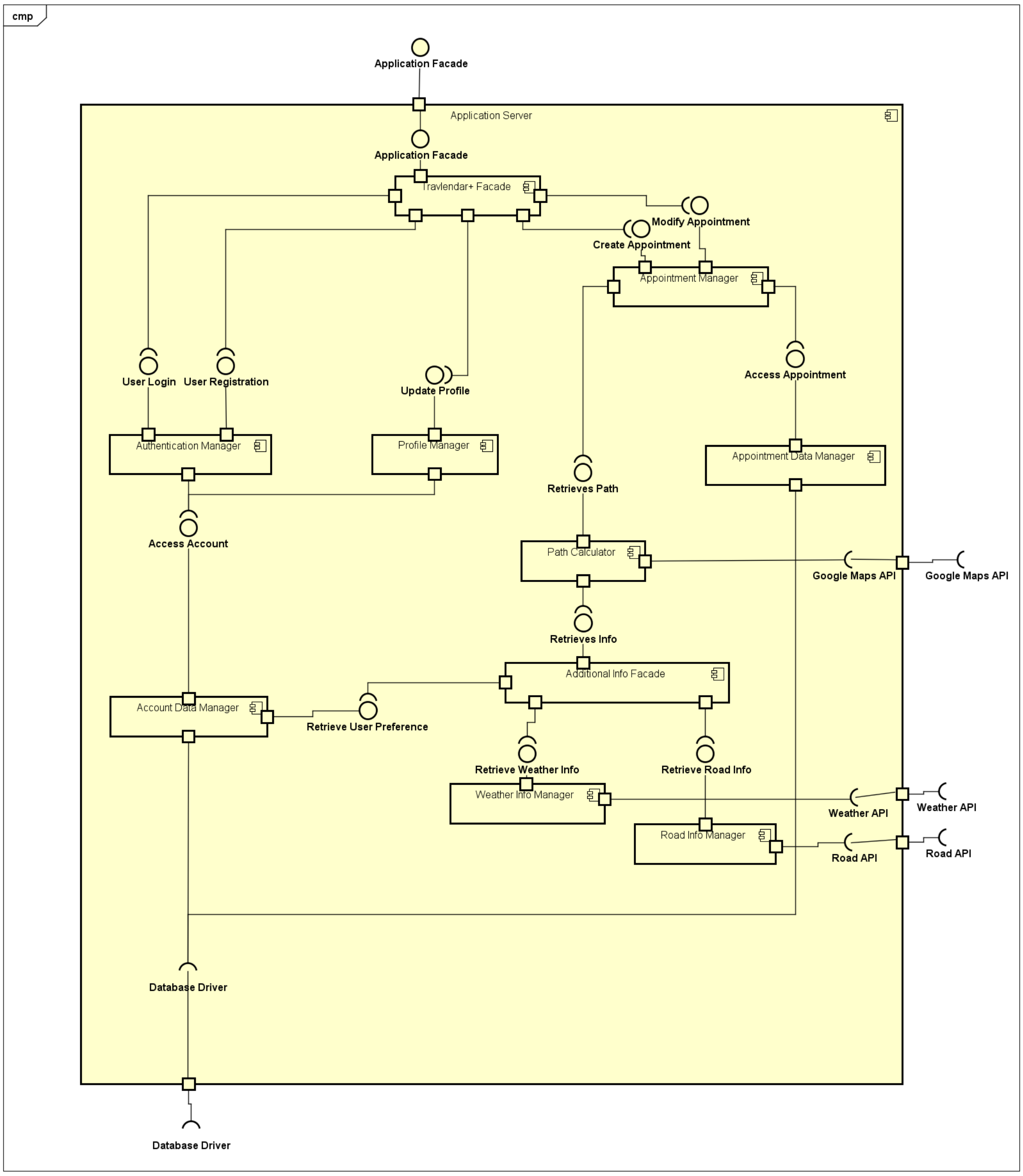

Figure 5: *Application Server* component diagram

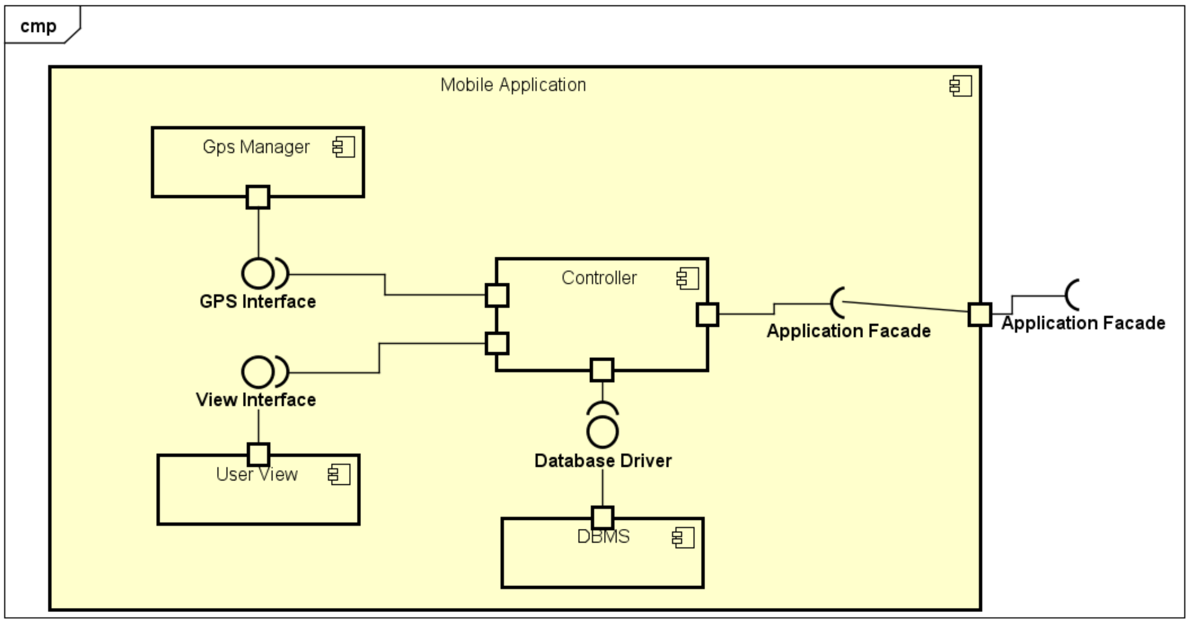

Figure 6: *Mobile Application* component diagram

2.3 Deployment View

The architecture chosen for Travlendar+ is a Four-Tier in Three-Layer one, where a high-level mapping layer to component is as follows:

- Presentation Layer: Mobile application/Web application and Web Server
- Logic Layer: Application server
- Data Layer: DBMS and Database

Note that the definition of layer is just a logic separation of the components that compose the system with the intent to better organise the code that needs to be developed, while a tier is a physical machine onto where the code is running.
To better understand the choice made a scheme is provided in Figure 7, where it's clear that there are three layers (the different coloured boxes) and four physical tiers (We consider the mobile and web application as a single tier, while obviously the code is written in the DBMS and not in the DB).

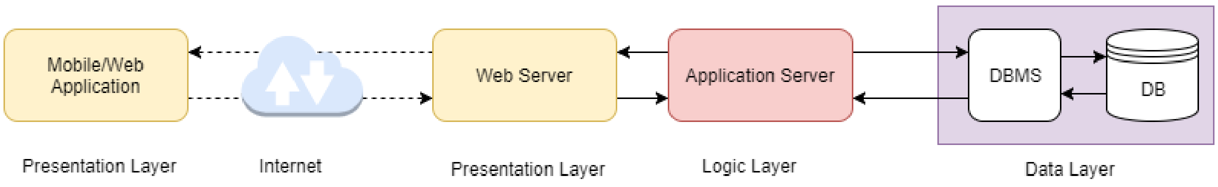

Figure 7: System's Tiers and Layers

Implementation choices
The technology chosen for the implementation on the system are mainly based on Java Enterprise Edition (JEE) since it offers a large number of tools and alternatives to develop multi-tier systems that need web based logic and storage and having at the same time the possibility of adding new functionalities in future, making the system more scalable.
Web Pages: The choice fell on JSP given the flexibility that a few snippets of Java code in a dynamic web page can provide.
Application Logic: EJB was the selected technology given that the system is developed mainly using JEE.
Application Server: GlassFish 5.0 has been chosen over other alternatives since it's an open source application server fully supported by Oracle. **Web Server:** GlassFish 5.0 was chosen again for coherence with the application server.
DBMS: MySQL was selected given that is supported by Oracle and is well known, making the amount of documentation available quite large. It was paired with InnoDB because it's the currently most used alternative and allows us to use foreign keys.

A complete overview of the technologies chosen can be seen in the decision flow diagram in Figure 8

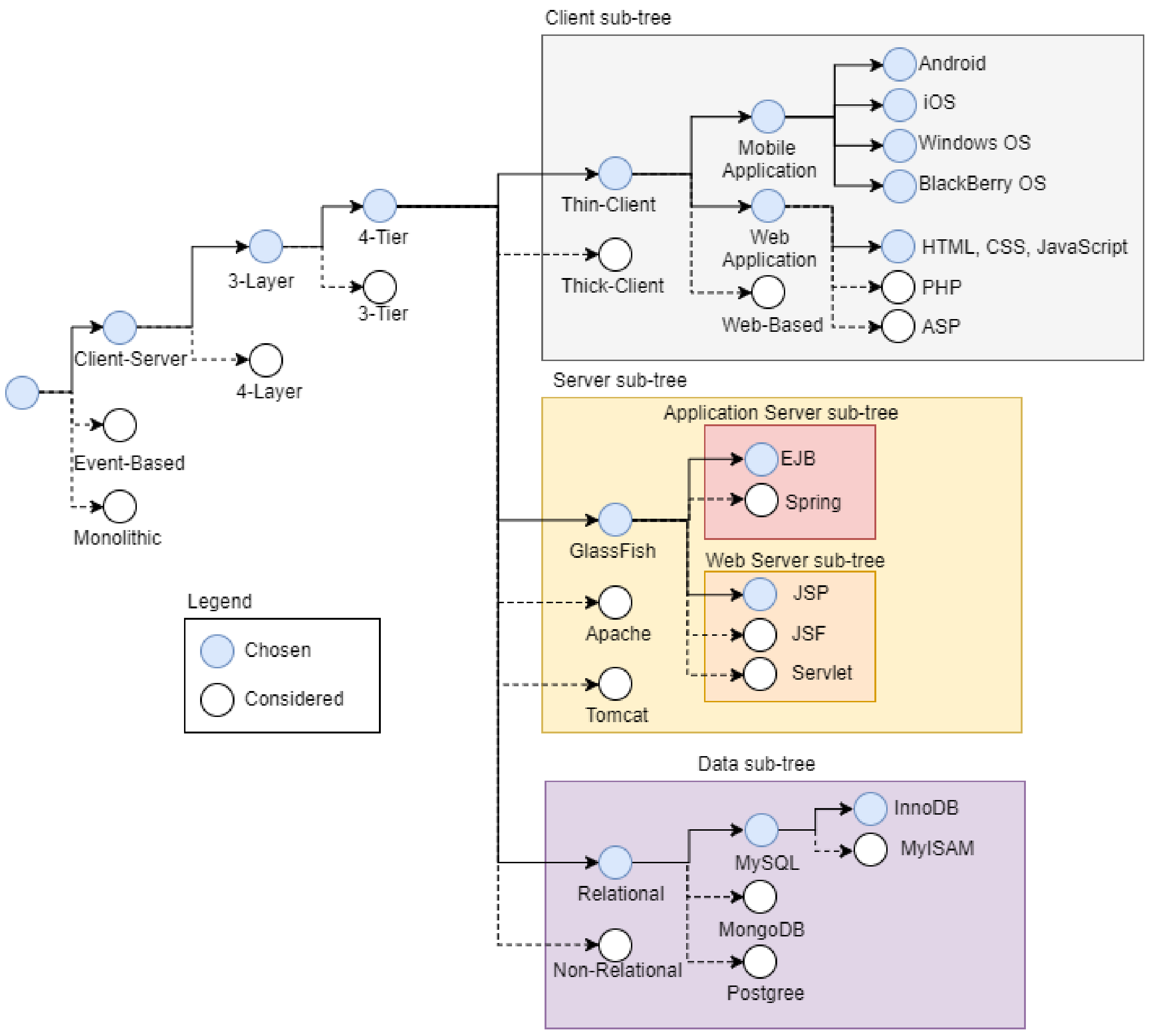

Figure 8: Decision Flow Diagram

2.3.1 Deployment Diagram

The deployment diagram can be seen in Figure 9, note that the components specified in Figure 6 and in Figure 5 can be seen in this representation of the mapping on concrete devices.

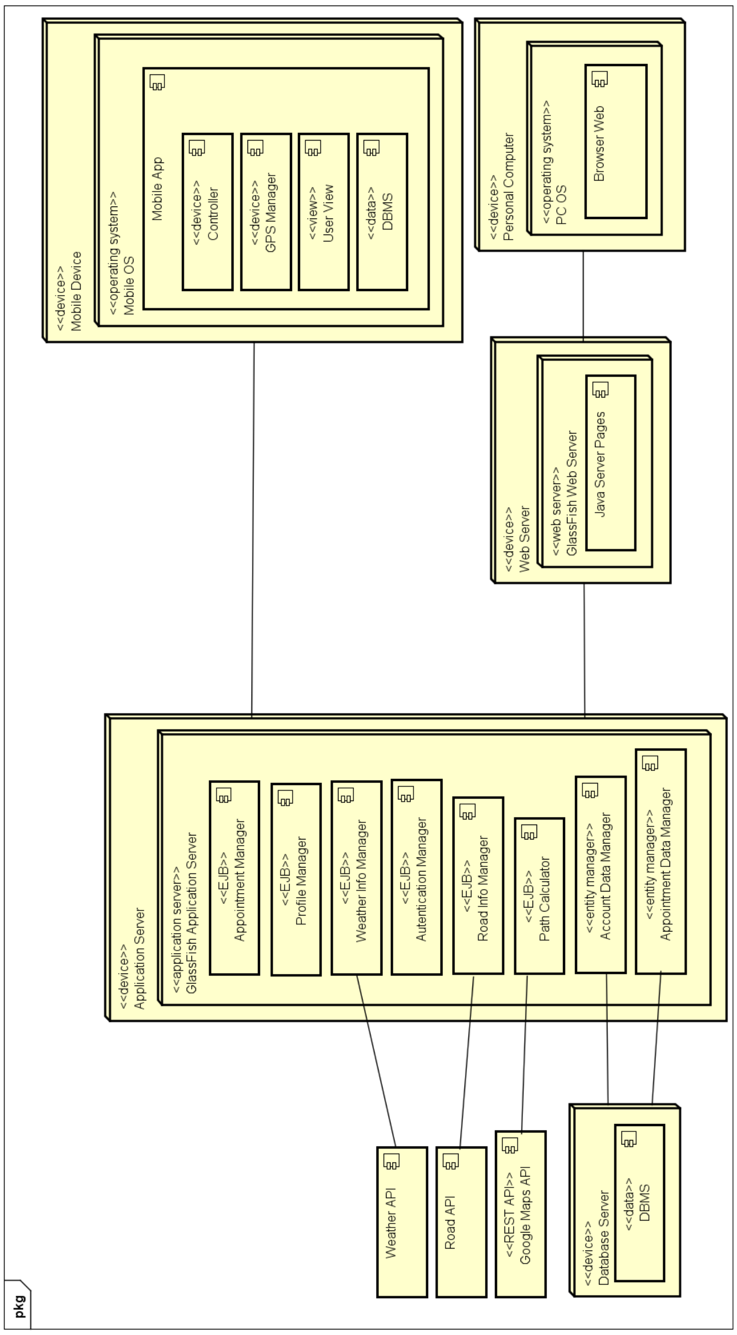
pkg
<<device>> Mobile Device
<<operating system>> Mobile OS
Mobile App
<<device>> Controller
<<device>> GPS Manager
<<view>> User View
<<data>> DBMS
<<device>> Personal Computer
<<operating system>> PC OS
Browser Web
<<device>> Web Server
<<web server>> GlassFish Web Server
Java Server Pages
<<device>> Application Server
<<application server>> GlassFish Application Server
<<EJB>> Appointment Manager
<<EJB>> Profile Manager
<<EJB>> Weather Info Manager
<<EJB>> Autentication Manager
<<EJB>> Road Info Manager
<<EJB>> Path Calculator
<<entity manager>> Account Data Manager
<<entity manager>> Appointment Data Manager
Weather API
Road API
<<REST API>> Google Maps API
<<device>> Database Server
<<data>> DBMS

Figure 9: Deployment Diagram

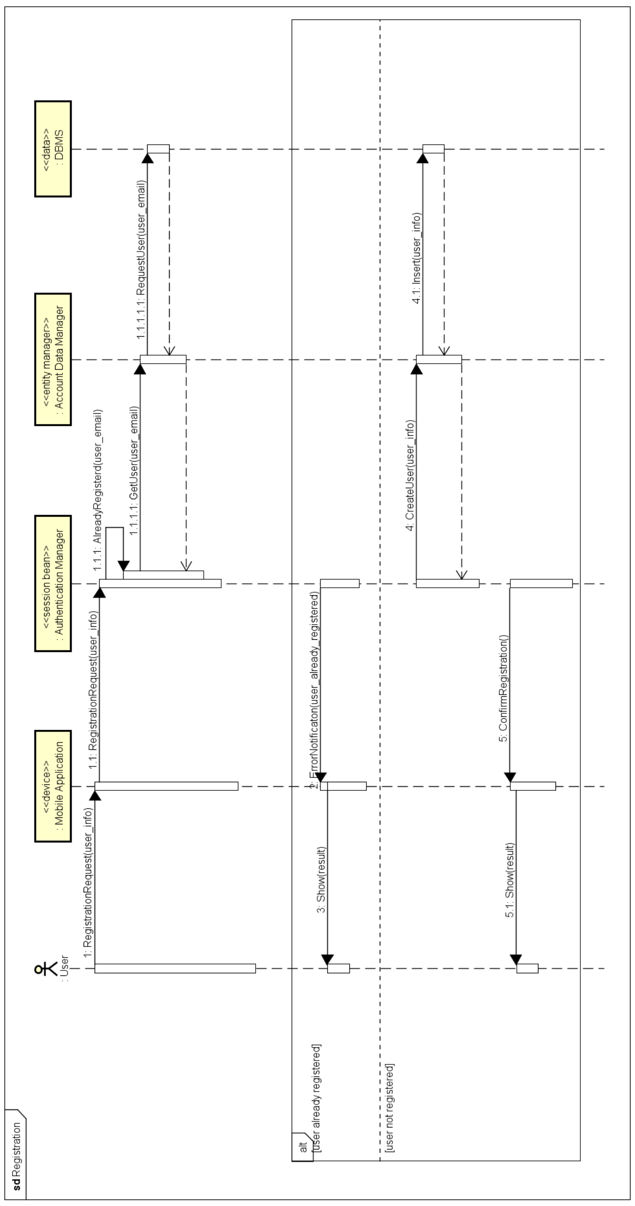
sd Registration
: User
<<device>> : Mobile Application
<<session bean>> : Authentication Manager
<<entity manager>> : Account Data Manager
<<data>> : DBMS
1: RegistrationRequest(user_info)
1.1: RegistrationRequest(user_info)
1.1.1: AlreadyRegisterd(user_email)
1.1.1.1: GetUser(user_email)
1.1.1.1.1: RequestUser(user_email)
alt
[user already registered]
2: ErrorNotificaton(user_already_registered)
3: Show(result)
[user not registered]
4: CreateUser(user_info)
4.1: Insert(user_info)
5: ConfirmRegistration()
5.1: Show(result)

Figure 10: Diagram of interaction between components in the *Registration* use case

2.4 Runtime View

In this section are represented the most important runtime views by using sequence diagrams that highlight the main interactions between the user and the components of the system for each analysed use case.
Note that each *Error Message* is just an abstraction of different errors with different codes for different situations, this was done in order to simplify the diagrams.

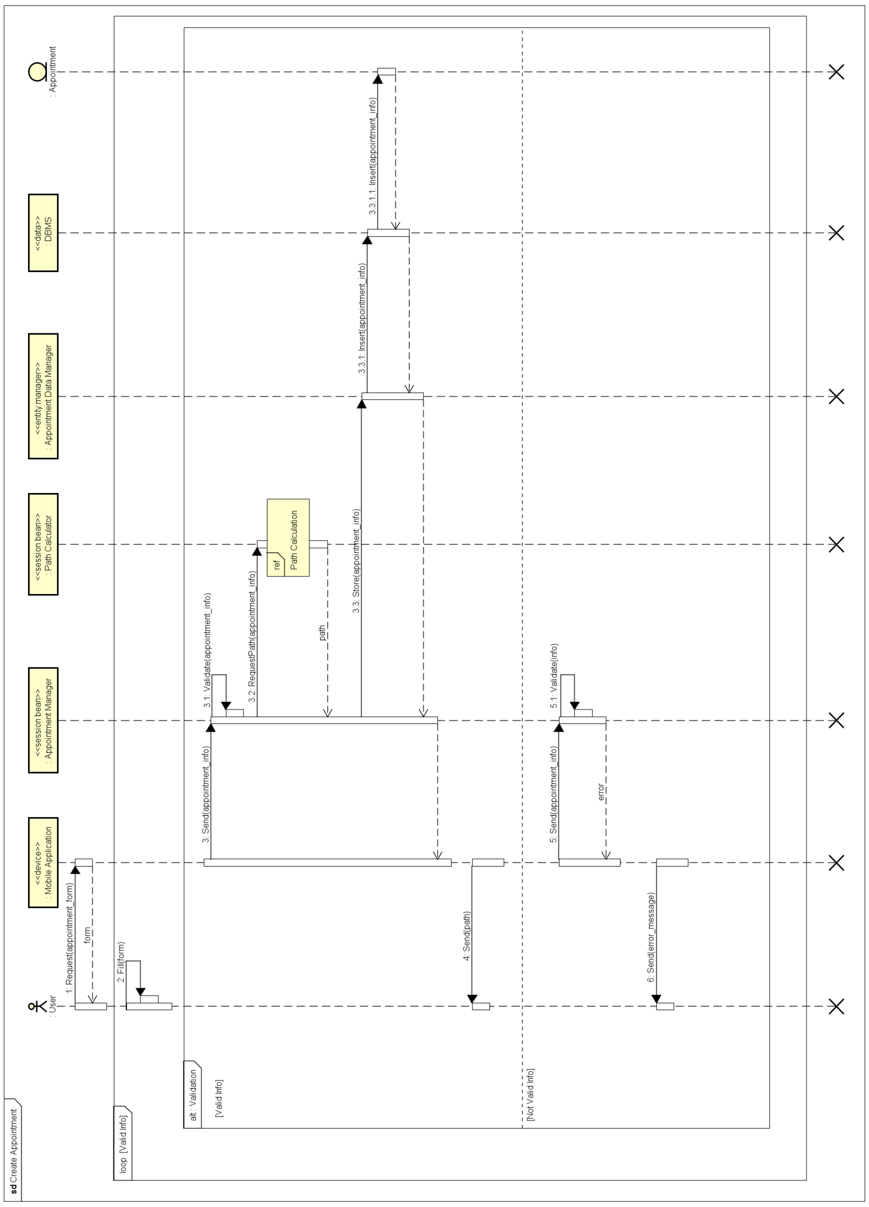
sd Create Appointment
: User
<<device>> Mobile Application
<<session bean>> Appointment Manager
<<session bean>> Path Calculator
<<entity manager>> Appointment Data Manager
<<data>> DBMS
: Appointment
1: Request(appointment_form)
form
loop [Valid Info]
2: Fill(form)
alt : Validation
[Valid Info]
3: Send(appointment_info)
3.1: Validate(appointment_info)
3.2: RequestPath(appointment_info)
ref
Path Calculation
path
3.3: Store(appointment_info)
3.3.1: Insert(appointment_info)
3.3.1.1: Insert(appointment_info)
4: Send(path)
[Not Valid Info]
5: Send(appointment_info)
5.1: Validate(info)
error
6: Send(error_message)

Figure 11: Diagram of interaction between components in the *Create Appointment* use case, the internal interaction of the path calculator is in Figure 14

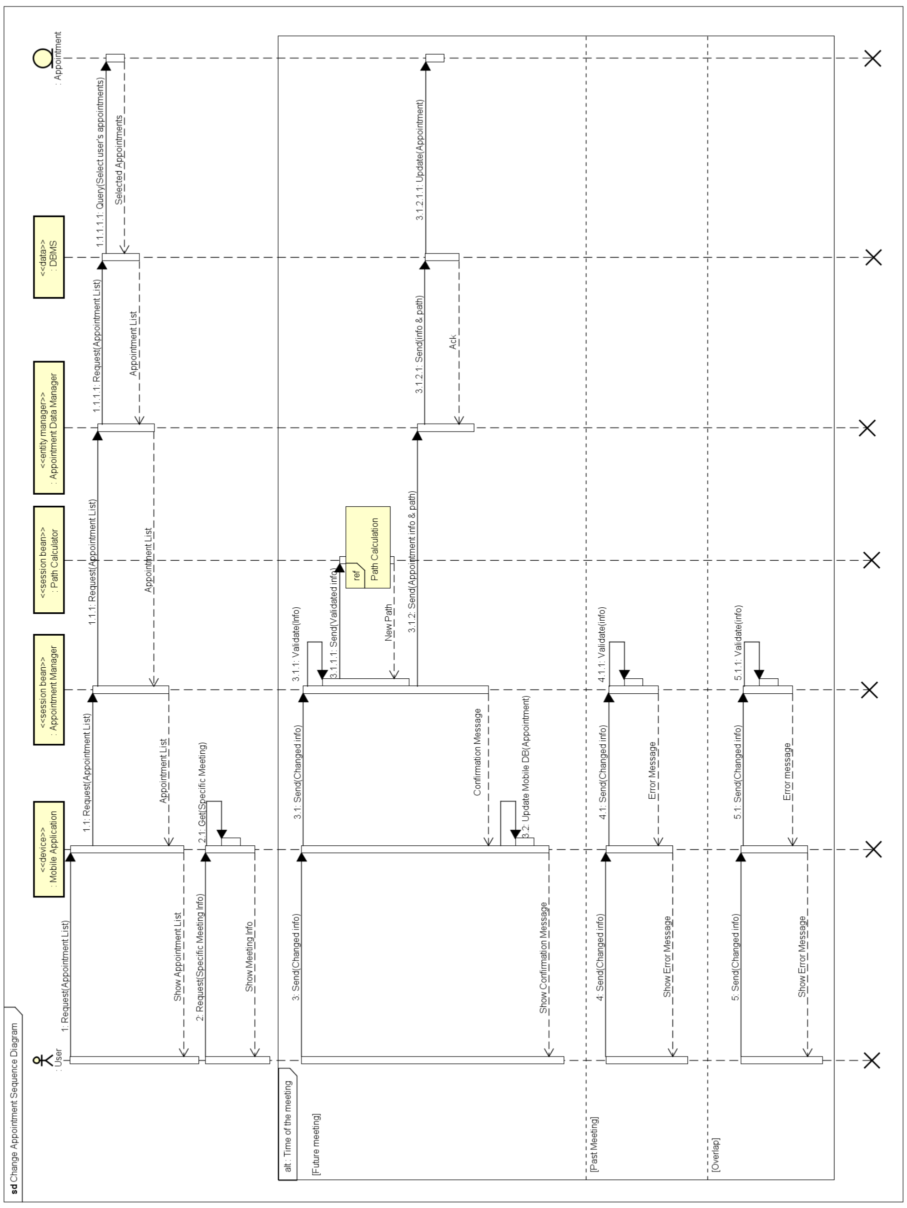
sd Change Appointment Sequence Diagram
: User
<<device>> : Mobile Application
<<session bean>> : Appointment Manager
<<session bean>> : Path Calculator
<<entity manager>> : Appointment Data Manager
<<data>> : DBMS
: Appointment
1: Request(Appointment List)
1.1: Request(Appointment List)
1.1.1: Request(Appointment List)
1.1.1.1: Request(Appointment List)
1.1.1.1.1: Query(Select user's appointments)
Selected Appointments
Appointment List
Appointment List
Appointment List
Show Appointment List
2: Request(Specific Meeting Info)
2.1: Get(Specific Meeting)
Show Meeting Info
alt : Time of the meeting
[Future meeting]
3: Send(Changed info)
3.1: Send(Changed info)
3.1.1: Validate(info)
3.1.1.1: Send(Validated info)
ref
Path Calculation
New Path
3.1.2: Send(Appointment info & path)
3.1.2.1: Send(info & path)
3.1.2.1.1: Update(Appointment)
Ack
Confirmation Message
3.2: Update Mobile DB(Appointment)
Show Confirmation Message
[Past Meeting]
4: Send(Changed info)
4.1: Send(Changed info)
4.1.1: Validate(info)
Error Message
Show Error Message
[Overlap]
5: Send(Changed info)
5.1: Send(Changed info)
5.1.1: Validate(info)
Error message
Show Error Message

Figure 12: Diagram of the interaction between components in the *Change Appointment* use case, it should be noted that the internal interaction of the path calculator is better specified in Figure 14 in order to keep the diagram as clean as possible

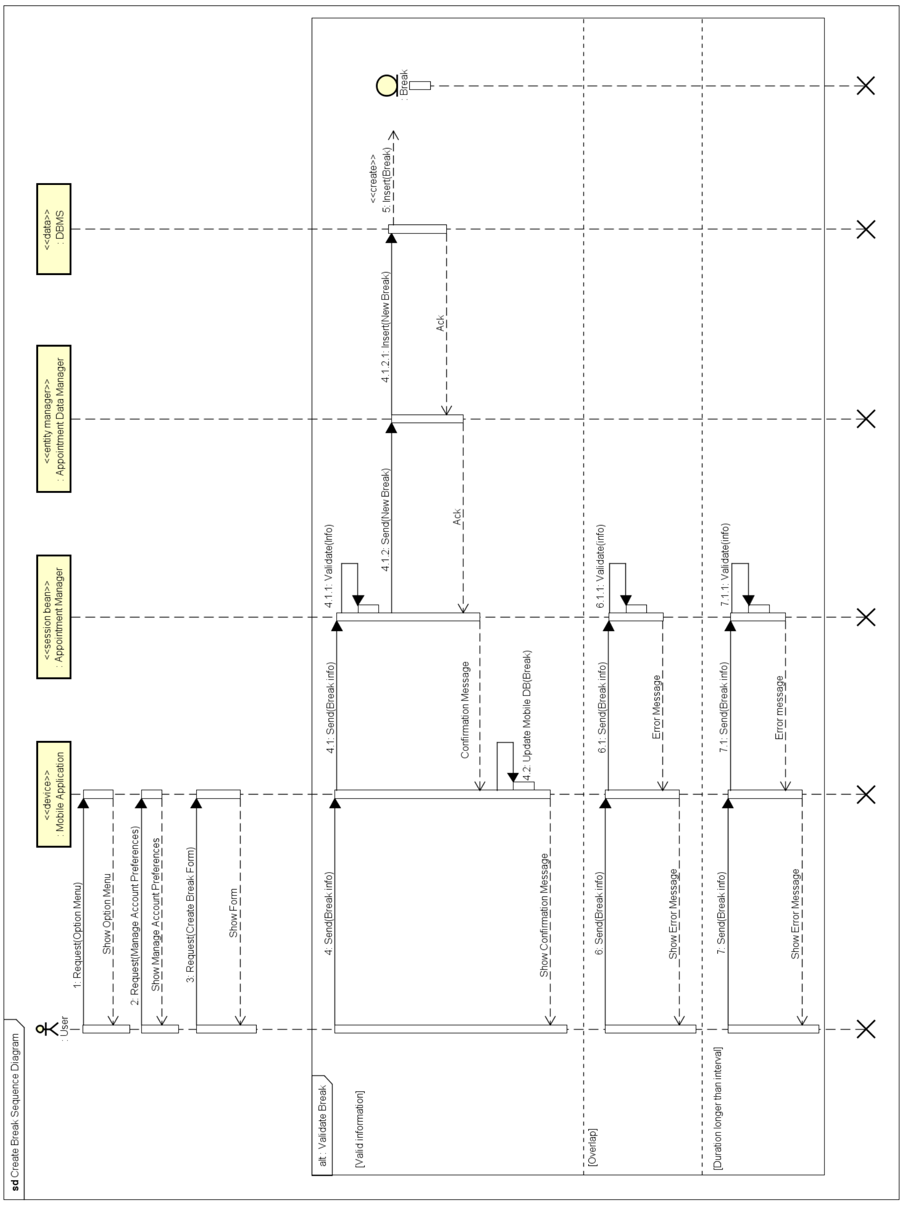

Figure 13: Diagram of the interaction between components in the *Create Break* use case, the *AppointmentManager* manages also breaks

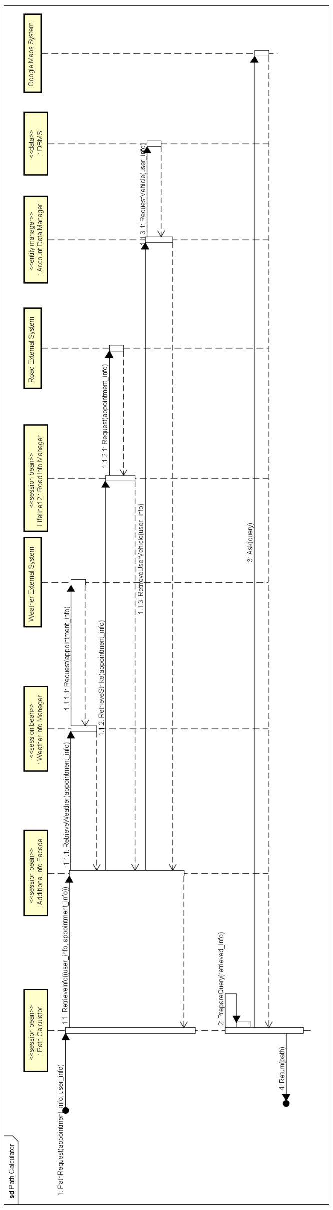
sd Path Calculator
<<session bean>> : Path Calculator
<<session bean>> : Additional Info Facade
<<session bean>> : Weather Info Manager
Weather External System
<<session bean>> Lifeline12 : Road Info Manager
Road External System
<<entity manager>> : Account Data Manager
<<data>> : DBMS
Google Maps System
1: PathRequest(appointment_info, user_info)
1.1: RetrieveInfo((user_info, appointment_info))
1.1.1: RetrieveWeather(appointment_info)
1.1.1.1: Request(appointment_info)
1.1.2: RetrieveStrike(appointment_info)
1.1.2.1: Request(appointment_info)
1.1.3: RetrieveUserVehicle(user_info)
1.1.3.1: RequestVehicle(user_info)
2: PrepareQuery(retrieved_info)
3: Ask(query)
4: Return(path)

Figure 14: Diagram of the interaction between components during a calculation of a path.
The *Path Calculator* asks for information to the *Additional Info Facade*, and, based on this information, prepare the request to send to *Google Maps*

2.5 Communication Interfaces

2.5.1 Database Driver

This interface is used to allows the interaction between the *Application Server* and the *DBMS*.
There are two different components that interacts with this interface:

- Account Data Manager, for managing user's information.
- Appointment Data Manager, for managing appointments' information.

2.5.2 Application Facade

This interface provides a common point of access for both the *Mobile Application* component and the *Web Server* component.
It is provided form the *Travlendar+ Facade*.

2.5.3 Google Maps API

This interface can provide:

- a map of the path from the departure location to the appointment location.
- the ETA with a specific travel means
- the path directions for each travel means.

The *Path Calculator* component builds the path from the information retrieved from this API.

2.5.4 Weather API

This interface provides a way to obtain the forecast of the appointment date. The *Path Calculator* component relies on this informations to take decision about the best path.

2.5.5 Road API

This interface provides a way to obtain information about possible strikes on the appointment date. The *Path Calculator* component relies on this informations to take decision about the best path.

2.6 Selected architectural styles and patterns

In this section we will discuss which architectural styles and were chosen and why.

2.6.1 Client-Server

The client-server architecture has been used multiple times, we note them all in the following list.

1. The **mobile application** is the client with regard to the **application server** (which is of course the server) that handles the requests, this ensures that the mobile application needs the least amount of logic programmed, we will elaborate more on this topic in Figure 15.
2. The **web browser** installed onto the user's personal computer acts as a client, while the server is the **web server** that receives its requests.
3. The **web server** is also a client when looking at its relationship with the application server, that has to handle the requests that are sent to the **web server** from the **web browser**.
4. The **application server** acts like a client too, this appends when it is performing query requests to the **DBMS** that is the server in the interaction.

2.6.2 Multi Tier

Already discussed at the start of subsection 2.3, a multi tier architecture allows the system to be much more **scalable** since each physical upgrade can be done without needing an intervention on the entire system.
It should be noted that it also benefits **fault tolerance** given that the mobile application can function without the need of the web server and it's **less expensive** to have redundant smaller section of the system rather than duplicating a mainframe.

2.6.3 Thin Client

A thin client architecture allows us to develop client side applications and interfaces that do not rely on the computational power of the user's hardware, this means that the system will be accessible even to users that do not own powerful last generation devices , this way the logic of the system is handled entirely by the application server, while the client side applications need only to display the information received from the server side, an abstract representation of the thin client concept can be seen in Figure 15.

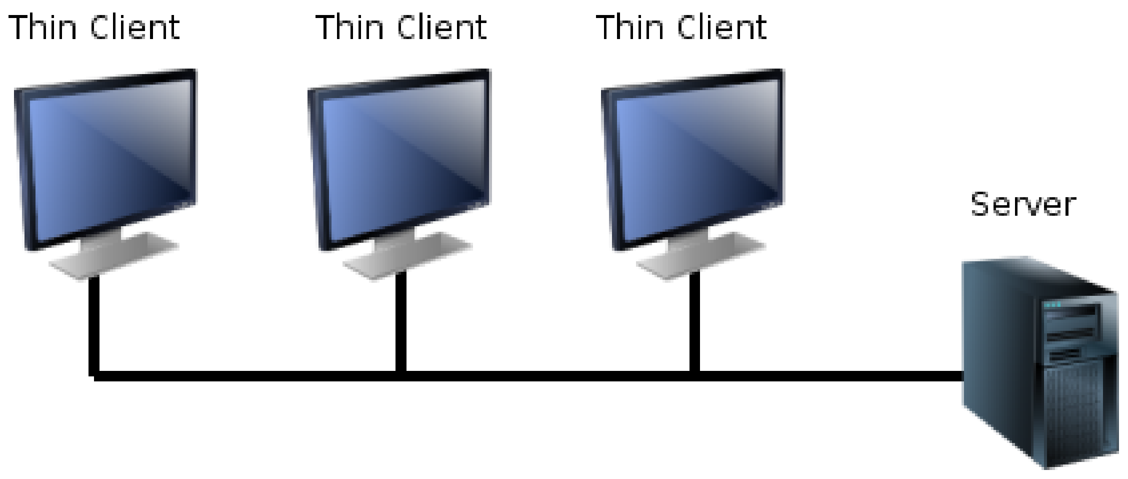

Figure 15: A series of thin clients connected to a server that provides the computational power.

2.7 Other design decision

2.7.1 Google Maps

Given that for a project of this size is not feasible to develop and maintain an entire navigation system, we will rely on *Google Maps* given the extreme flexibility that its APIs offer.
This choice translates in a path calculator that will act more like a *builder for queries*, as it will be discussed more in detail when talking about the algorithms used.

2.7.2 Cryptography

Given that the system has to memorize date, place and travel of the user for each event, privacy concerns can arise.
The system uses the **Advanced Encryption Standard** (AES) used as a standard by the U.S.A. government in the stronger 256 bits cipher to protect the user's most sensible information (like email and password) and the quicker 128 bits one for the trips data.
Using the method just described the system can still process with relative ease the tasks it is asked to do and even if a brute-force attack could manage to extract information about a trip, the identity of the user would still be almost impossible to decipher before it's obsolete.
Periodic re-ciphering of information is also scheduled when the traffic flow in the system is low.

3 Algorithm Design

Introduction

Once the system has the couple time-place for both the departure and the arrival, it must take into account all the preferences of the user, weather conditions and public transportation strikes in order to formulate the best possible query that will be forwarded and handled by Google Maps.

4 User Interface Design

4.1 UserInterfaces

5 Requirements Traceability

5.1 requirements traceability

6 Implementation, Integration and Test Plan

6.1 Introduction

In this section is present the documentation regarding the implementation order of the different components of the system and their integration with one another, also dealing with the test panning for when the code of the system will be written.

6.2 Entry Criteria

What follows is an illustration of the requisites that need to be fulfilled before the Integration phase can start.

RASD & DD:
The RASD and the previous sections of the DD must be completed and delivered before starting to consider the integration or the implementation.

Unit Testing:
Before starting the integration between different components each class and method must be keenly tested using JUnit testing, this is done to ensure that every sub-system is fully functioning on its own, or, in the case that the testing highlights bugs or incorrect behaviour in parts of the code, it allows the team to correct them at an earlier stage, resulting in a lower cost and more time efficient debugging.
The JUnit testing should cover no less than the 90% of the code to be considered satisfactory and each test must be run again at each addition of code and between integrations.

Documentation:
Each method and class must be fully documented using JavaDoc in order to assure that the code is easily understandable, making it easier to extend and maintain even by different people that may end up working on the system in the future.
Names of classes, methods and variables must be chosen with the intent of communicating the reason of their existence and not be confusing or too similar to one another, also they should follow the standard Java naming conventions.

6.3 Elements to be Integrated

As already stated before, we chose a Four-Tier architecture, so the integration plan will be based heavily on this decision, with the following subsystem to be integrated (this of course means that each one of this subsystem will have to be completely integrated with regard to its internal subsystem too):

Tier 1 *Database*: This tier is composed of the physical database and the DBMS that handles the requests of the higher systems via query to the database itself.
It should be noted that the integration of this tier is almost exclusively about the DBMS given that the database should be acquired from dedicated companies as an external system in order to avoid the toll of managing the redundancy and the expansion of the storage modules.

Tier 2 *Application Logic*: It includes all the components and subsystems that handle the logic of the system, it should be noted that each individual component should be tested by itself before integrating it with the others and after each integration new tests should be ran to confirm the correct functioning of the integrated subsystems.

Tier 3 *Web*: It handles all the interactions between the client's *web browser* and the *web server*, this means managing the interface that will be ultimately be displayed to the user.

Tier 4 *Client*: Composed by both the *mobile application* and the *web browser* it's the less logic heavy tier of the system, each client can be seen as a mere presentation system and the integration should focus on making sure that the communication doesn't imply high waiting time and that the graphical interface behaves as it should regarding the received data.

6.4 Integration Testing Strategy

As a testing strategy it has been chosen a **bottom up** approach, the thinking behind this choice can be found by noting that the system is composed by many small components that can be tested individually. This results in a minimal amount of stubs needed during the testing phase (but of course on the other hand there is a need for higher-level modules like drivers).
We also decided to integrate elements of a **critical-module-first** approach in order to give precedence to the core elements (like those containing the application logic) with the intention of conducting a more extensive testing on them and find system-breaking bugs as early as possible given that as it's commonly known it's much easier and cheaper to fix a defective software in the earlier stages of development.
Furthermore we note once again that the database is a commercially available solution, while the database management system is an already existing solution that is compatible with the DB, making them ready to use from the start with only the need of configuring the DBMS to communicate with the application server, but it doesn't require any programming in the sense of coding.

7 Appendix

8 Software Used

1. Texmaker as an editor for LaTeX.
2. Astah for component, deployment and sequence diagram.
3. Draw.io for ER diagrams and other generics diagrams.
4. Git & GitKraken.
5. Proto.io for mockups.

9 Hours of Work

The hours listed are comprehensive of individual and group work time.

1. Riccardo Facchini:
2. Andrea Guglielmetti: