POLITECNICO
MILANO 1863

Design Document (DD)

Davide Rossetto 894029, Alessandro Tatti 883861

Delivery date: 2017 Nov 26

v0.9

Contents

Section 1

Introduction

1.1 Purpose

The structure of the following document is divided into two sections:

1. Design Document (main section)
2. Integration Test Plan Document (secondary section)

The Design Document (DD) is intended to provide a more detailed functional description of the Travlendar + system to - be by providing technical details and describing the main architectural components, their interfaces, and their interactions.

The relationships among the different modules are highlighted using UML standards and other useful diagrams that show the structure of the system.

The document has to guide the software development team to implement the project architecture, providing a stable reference and a unique view of all parts of the software, defining their operation.

The second part of the document is intended to provide guidelines to adequately carry out the planning of the integration test phase.

The document includes determining which necessary tools, drivers and data structures that will be useful during the test process.

1.2 Scope

The system aims to support a personal event management service, providing features for choosing the best path, means of transport, and insertion of the break. The system must also make sure there are no overlap among meetings and among meetings and breaks.

The system is structured with a four-layer architecture. The purpose of the document is to describe this architecture in detail.

The system described is suitable for different types of customers: different actors that interact with the system - to be by generating a client-server dualism, so the flow of requests and responses.

The architecture must be designed with the intention of being maintainable and extensible, to make future possible changes.

This document aims to guide the implementation phase so that cohesion and decoupling are increased as much as possible. In order to do this, individual components must not include too many independent functions and reduce interdependence.

This document will follow specific architectural styles and design templates used for future implementation, as well as common design paradigms that combine useful features of this concepts.

1.3 Definitions, Acronyms, Abbreviations

ACID: Atomicity, Consistency, Isolation and Durability. This is the set of properties of database transactions.

API: Application Programming Interface.

DD: Design Document.

DBMS: DataBase Management System.

E-R diagram: entity relationship diagram.

EJB: Enterprise JavaBean.

HTTPS: HyperText Transfer Protocol over Secure Socket Layer.

IE: Internet Explorer.

ITPD: Integration Test Plan Document.

JAX - RS: Java API that provides support in creating web services according to the REST pattern.

JEE: Java Enterprise Edition.

JPA: Java Persistence API.

JSON: JavaScript Object Notation

MVC: Model-View-Controller.

NGINX: Web Server used in the Web Server Implementation.

RASD: Requirements Analysis and Specification Document.

REST: REpresentational State Transfer. It is a way of providing interoperability between computer systems on the Internet.

RESTful: a REST compliant system.

Swift: language used for developing iOS apps.

UIKit: the framework provides the required infrastructure for your iOS.

UML: Unified Modeling Language.

UX: User Experience.

XML: eXtensible Markup Language

1.4 Revision history

v0.1 Construct basic document's structure, add *Overview.*

v0.2 Add *Purpose*, *Scope*, *Definitions, Acronyms, Abbreviations*, *Reference documents* and *Document structure.*

v0.3 Add *High level components*, *Appendix* and *Bibliography.*

v0.4 Add *Component view.*

v0.5 Add *Deployment view*, *Runtime view*, *Component interfaces*, *Architectural styles and patterns* and *Other design decisions.*

v0.6 Add *Requirements Traceability* and *UX diagrams.*

v0.7 Add *Algorithm Design.*

v0.8 Add *Implementation, Integration and Test Plan.*

v0.9 Syntactical review.

v1.0 First delivery.

1.5 Reference documents

This document follows the guidelines provided by ISO/IEC/IEEE 1016:2009 [3] related to system design and software design descriptions for complex software systems.

The indications given in this document are based on those given in the previous delivery for the project, the RASD document [1].

This document is strictly based on the RASD assignment [2] and the test plan example presented during the lessons for the project of Software Engineering II, course held by Elisabetta Di Nitto and Matteo Giovanni Rossi at the Politecnico di Milano, A.Y. 2017/2018.

1.6 Document structure

This document consists of six sections:

Section 1: Introduction. This section provides a general introduction and overview of the Design Document and the covered topics that were not previously taken into account by the RASD [1].

Section 2: Architectural Design. This section shows the main system components together with subcomponents and their relationship. This section is divided into different parts whose focus is mainly on design choices, interactions, architectural styles and patterns.

Section 3: Algorithm Design. This section focuses on the definition of the most relevant algorithms to be implemented by the system to be.

Section 4: User Interface Design. It provides an overview on how the user interface will look like.

Section 5: Requirements Traceability. This section explains how the requirements you have defined in the RASD [1] map to the design elements that you have defined in this document.

Section 6: Implementation, Integration and test plan. This section identifies the order in which the developer plan to implement the subcomponents of the system and the order in which he plans to integrate such subcomponents and test the integration.

At the end of the document are an Appendix and a Bibliography, providing additional information about the sections listed above.

Section 2

Architectural Design

2.1 Overview

This section gives a detailed overview of physical and logical infrastructures of the system-to-be and describes the main components and their interactions.

A top-down approach is adopted for the description of the architectural design of the system:

High level components: a description of the high-level components and their interactions.

Component view: a detailed vision of the components described in the previous section.

Deployment view: a set of guidelines on how to deploy the components on physical tiers.

Runtime view: a detailed vision of the dynamic behavior of the software.

Component interfaces: a description of the different interfaces.

Architectural styles and patterns: a set of Architectural styles, design patterns and paradigms used in the design phase.

Other design decisions: a list of all relevant decisions taken during the design process and not mentioned before.

2.2 High level components

The high-level components of the system are the following:

Database: the *Data Layer* of the system; all the data structures and entities concerning data storage. This level does not contain any application logic.

Application Server: the layer that contains all the *application logic* and *key algorithms* of the system.

Web Server: the Layer that provides all the *web pages* to the *web-based application.* This level does not contain any application logic.

Mobile Application: the *Presentation Layer* of the *mobile application*; This level does not contain any application logic.

Web Browser: the *Presentation Layer* of the *web-based application*; It just renders the pages obtained from the *Web server* and executes its scripts.

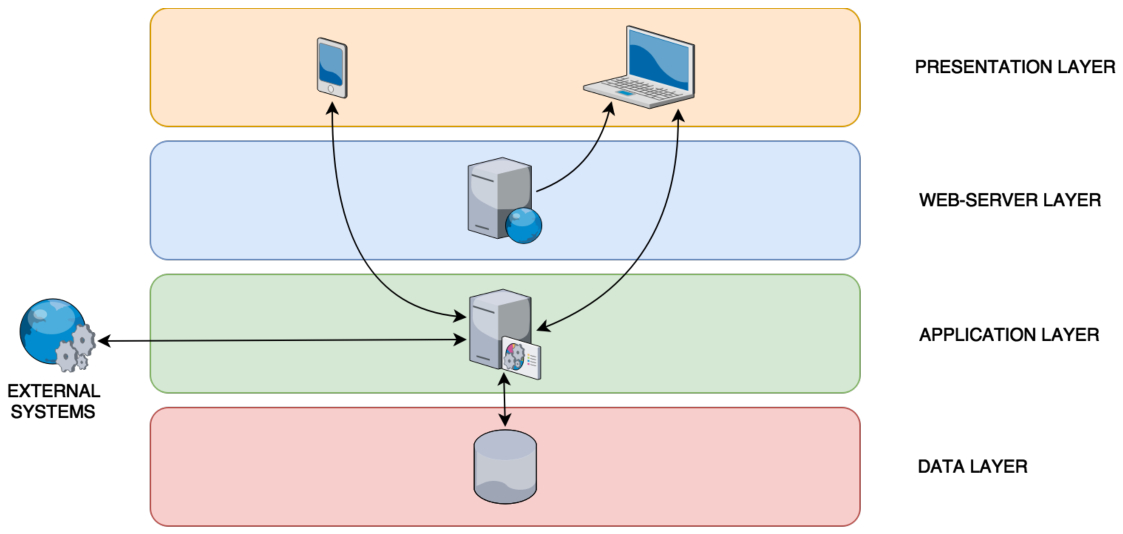

Figure 1: Layer structure of the system.

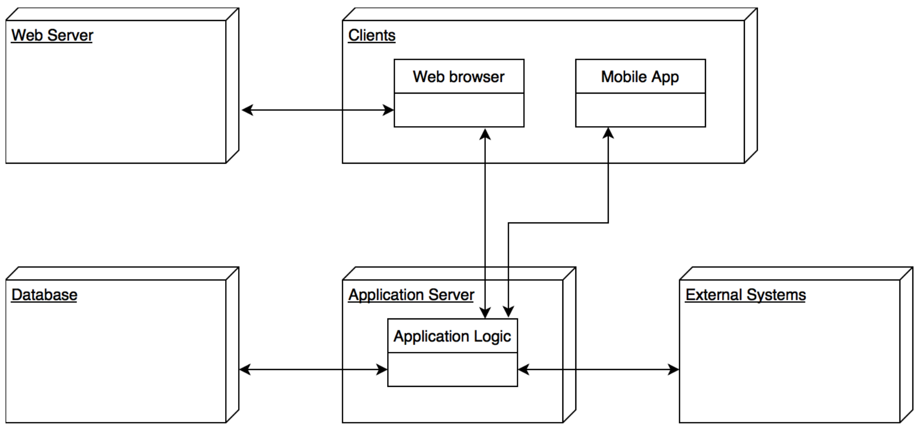

Figure 2: High-level components of the system.

2.3 Component view

This section shows in a more detailed way the components introduced in the overview, their role and their interactions.

2.3.1 Database

The database layer must include a DBMS component in order to manage the insertion, modification, deletion, and registration of data transactions within the storage memory.

The DBMS must guarantee the correct functioning of simultaneous transactions and ACID properties; the DBMS must be relational because the requirements of the application in terms of data storage do not require a more complex structure than the one provided by the relational data structure.

The data layer must be accessible only through Application Server via a dedicated interface. The Application Server must provide a persistence unit to handle the dynamic behavior of all persistent application data.

The database must be optimized during the implementation phase to ensure security by granting access to data according to the applicant's privilege level. Sensible data, for example passwords and personal information, must be encrypted correctly before being stored. Users must be granted access only on the provision of correct and valid credentials.

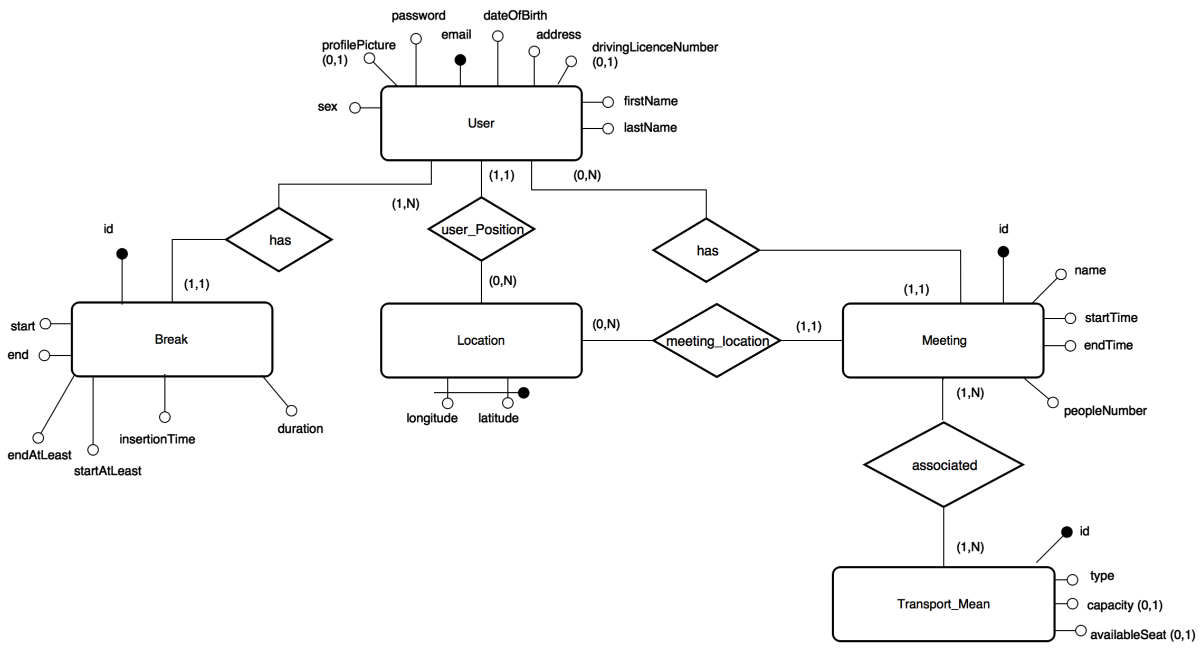

Figure 3: ER Diagram.

2.3.2 Application Server

This layer must handle business logic, data layer connections, and application access modes from different clients and external systems.

The main feature of Application Server is the specific business logic modules that describe business rules and workflows for each feature provided by the application itself.

The interface with the data layer must be managed by a dedicated persistence unit to ensure that only the Application Server can access the database.

Application Server must provide a means of interfacing with the Web Server and mobile clients through specific APIs in order to separate the different levels from their individual implementation. It must provide a way to communicate with external systems by adapting the application to existing external infrastructures.

The main business logic modules must include:

UserManager: this module must manage all the logics concerning to user account creation, login and management.

EventManager: this module must contain all the logic needed to create and modify meetings and breaks. It must guarantee the consistency of the model preventing from overlapping among events.

MapManager: this module must contain all the logic used to locate users and meetings.

TravelInformationProvider: this module must be able to interface with external agents such as *Travel Information Providers* (i.e., ATM, Trenord, Alitalia, etc.).

NotificationManager: this module must be able to notify the user when it is necessary to do so (e.g. an user tries to insert a meeting that overlaps with a previous one).

2.3.3 Web Server

The Web Server Layer is responsible to provide the web pages to the users that intend to access the application's services via web.

This layer does not contain any application logic; it only provides static web pages that, through propers scripts (Javascript), are able to access the REST APIs of the Application Server and make it accessible to end users.

The scripts must be able to consume all the API's endpoints for the functionalities intended to be accessible via web. The communication with the APIs must be through textual data files over HTTPS (e.g. XML or JSON), as previously specified in *Application Server* section.

2.3.4 Mobile Application Client

The Mobile Application Client must be designed to allow easy communication and implementation independent with the Application Server. The mobile application user interface should be designed following the guidelines provided by Android and iOS producers. The application must provide a software module that manages the GPS connection so that it can track location data by providing it to application servers.

2.3.5 Web Application Client

A *Web Application Client* is any modern browser (eg., IE, Firefox, Chrome, Safari) able to run Javascript to allow the pages to consume the REST APIs of the *Application Server*.

2.3.6 Implementation Choices

Database Implementation

The *Data Layer* must respect the following constraints:

- Relational DBMS;
- MySQL 5.7 as DBMS implementation;
- JPA as database interface in the *Application Server*.

Application Server Implementation

In this layer the central character is JEE7. This choice grants us several benefits such as security, reliability and availability.

More specifically:

- GlassFish Server as *Application Server* implementation;
- Enterprise JavaBeans (EJB)
- JavaPersistenceAPI (JPA)
- JAX-RS to implement proper RESTful APIs

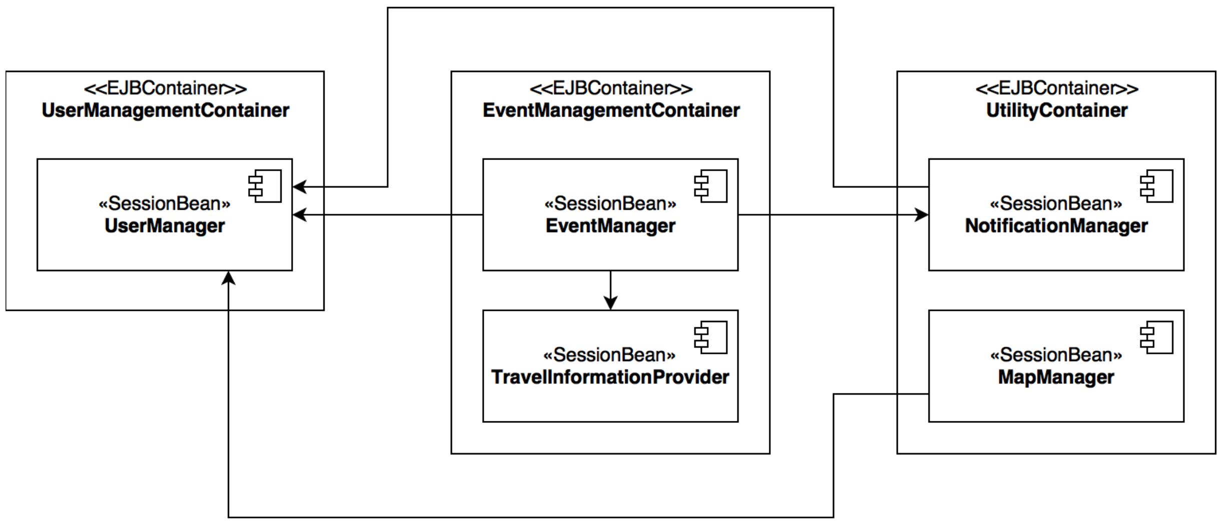

Figure 4: The components of the Application Server. An arrow going from component A to component B means that A uses interfacing methods provided by B.

Web Server Implementation

The main choice for this layer is to use NGINX 1.12 as web server.

NGINX is the most widely used web server implementations, is event-driven and will grant us reliability and efficiency handling HTTP(S) requests.

The *web server* will serve static files, i.e. html, css, javascript files, that will be interpreted client-side.

Web pages must be able to connect to the *application server*'s REST APIs and allow the user to access all the functionalities of the system through a web interface.

Mobile Application Implementation

There must be two *mobile application client* implementations to support both iOS and Android OSs:

iOS: This implementation must be written in Swift using UIKit.

Android: This implementation must be written in Java.

Both of them must:

- Communicate with the *application server* through the REST APIs over HTTPS.
- Follow the design guidelines provided by the devices' vendors.

2.4 Deployment view

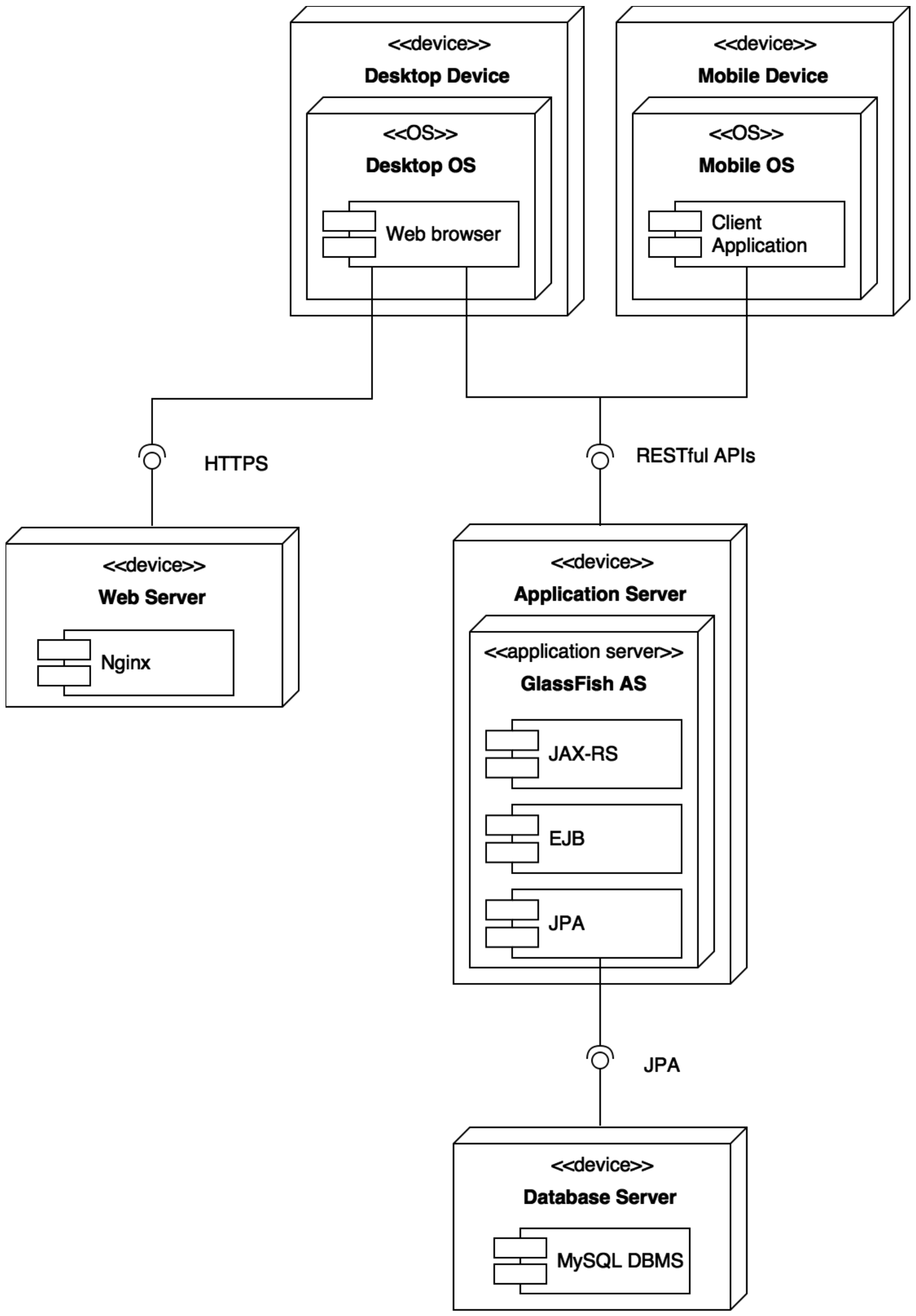

Figure 5: Deployment View Diagram.

2.5 Runtime view

This section describes the dynamic behavior of the system in the most relevant cases.

The sequence diagrams, shown below, highlight the runtime interactions between client, server, and database.

Interactions are expanded and analyzed in detail when there are internal interactions between subcomponents of the Application Server.

Direct interactions between Application Server and Database are not explicitly represented because they are abstract by the persistence unit of the Application Server.

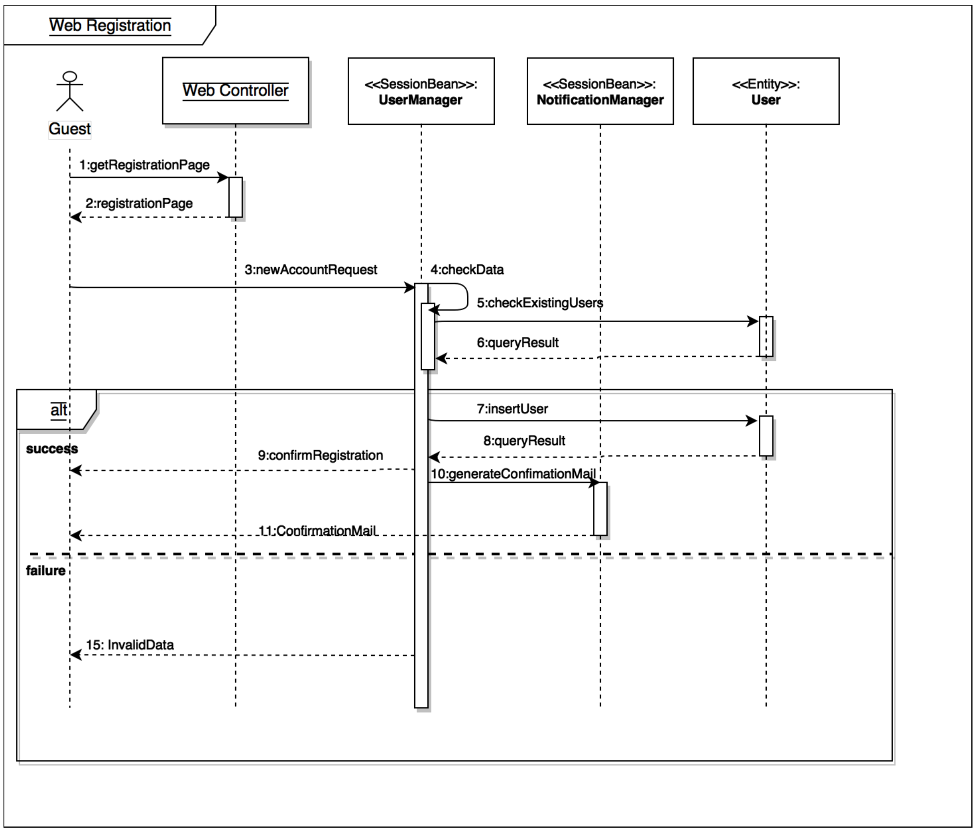

Figure 6: Registration *sequence diagram.*

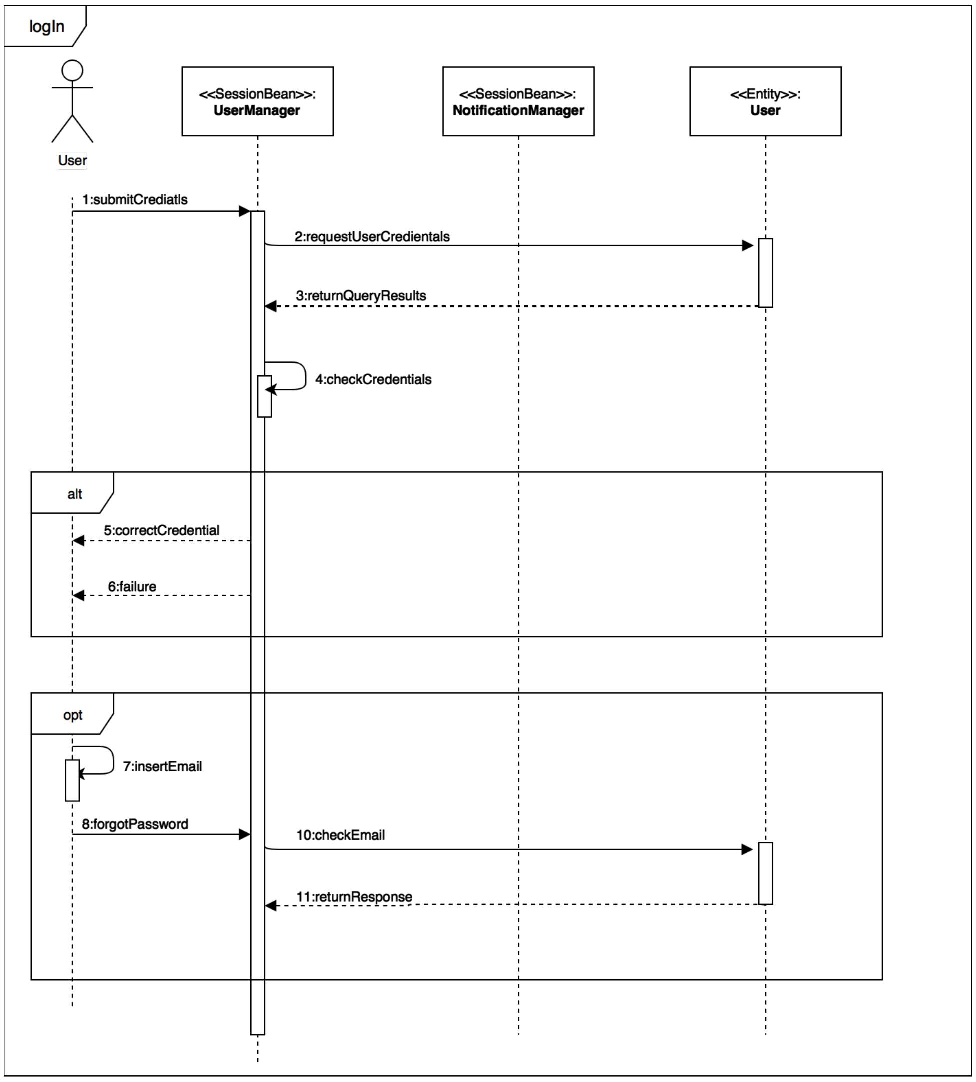

Figure 7: Login/Credentials recovery *sequence diagram.*

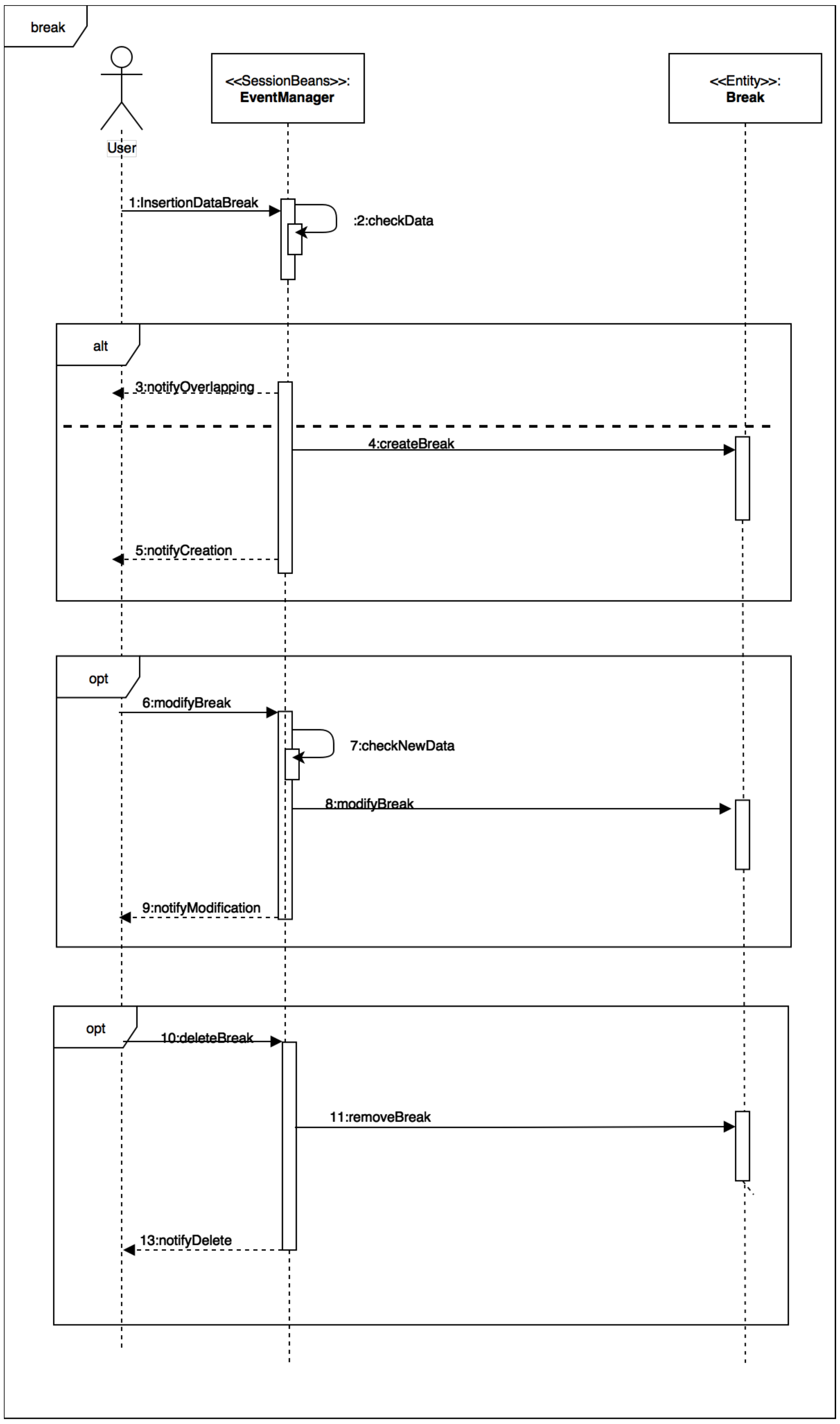

Figure 8: Meeting creation, modification and deletion *sequence diagram*.

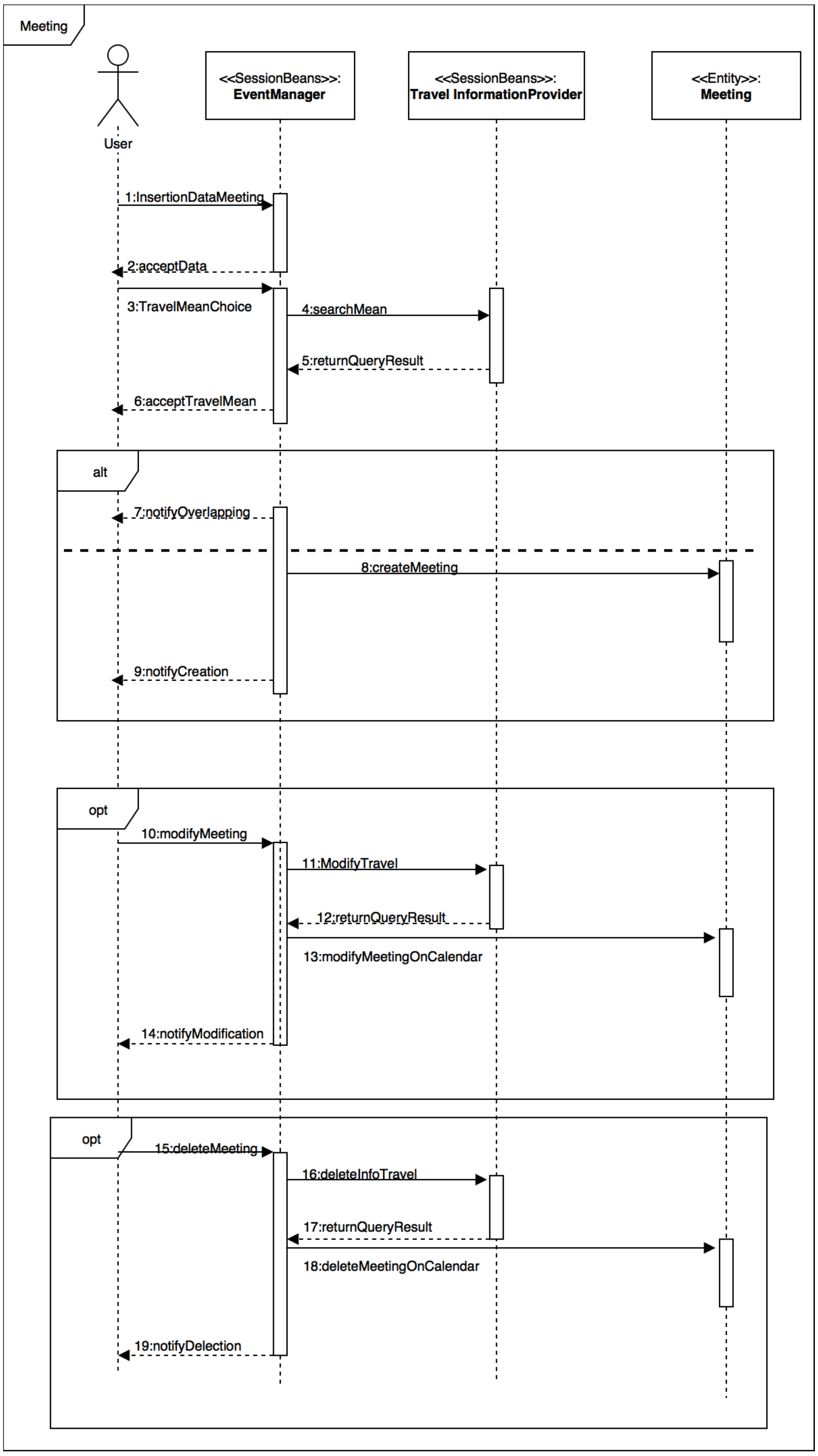

Figure 9: Break creation, modification and deletion *sequence diagram.*

2.6 Component interfaces

This section of the Design Document describes in detail the interfaces of the various components of the system. The various sections underline the relevant details about the interfaces required to use and interact with each component of the Application Server.

2.6.1 Database - Application Server

The Application Server is the only one that can access the Database directly through the Java Persistence API mapping between objects and actual relations.

2.6.2 Web Server - Web Browser

As described in the RASD document [1], interactions between clients browsers and Web Servers are based on the HTTPS protocol.

2.6.3 Application Server - Web Server and Clients

The communication between the Application Server and the clients must take place via the RESTful APIs provided by the Application Server and implemented using JAX-RS.

2.6.4 Application Server - External Systems

Application Server must connect with the following external system:

- One or more travel and transport operators that use the interface APIs to which the Application Server must adapt in order to deal with or search for travel solutions.

2.6.5 Internal interfaces for Application Server Components

UserManager

The *UserManager* interacts with the following entities:

- *user* entity, in order to create, update, and delete user information during the registration process and check credentials during login to the application.
- *location* entity, in order to always update user's position during any procedure.

The procedures implemented by the *UserManager* are:

submitLoginCredentials – This procedure is used to submit login credentials in order to access the application. The procedure takes the parameters as the email address and the password; if corrected, this function will have such a value for the user to perform other functions; if the credentials do not exist or are wrong, the return value is an error.

forgotPassword – this procedure is used when the user has forgotten the login credentials for the application. It returns a link to reset password.

registerUser – This procedure is used to request the creation of a new Travelendar + account; this takes as parameters all your personal information required for registration, in particular: first name, last name, sex, date of birth, address, email, password and driver's license number (optional); the return value for the procedure is an error when sending invalid data or a confirmation message in case of success.

deleteUser – This procedure is used to permanently delete a user account from the system; this does not require any parameter other than user authentication; the return value for this procedure is a confirmation message.

editProfile – This procedure is used whenever a user wishes to modify any information about his/her personal profile; this takes as a parameter a set of modified elements; the return value for this procedure is an error when entering invalid data, otherwise a confirmation message.

updateCurrentPosition – This procedure has the task of keeping the user's position updated. It takes the coordinates of the users place as parameters and sends it to the Map Manager to calculate the position; in case of failed sending returns an error message.

EventManager

The *EventManager* interacts with the following entities:

- *meeting* entity, in order to create, edit and delete meetings in the calendar.
- *location* entity, in order to set the location of a meeting.
- *brake* entity, in order to create, edit and delete brakes in the calendar.
- *transport_mean* entity, in order to calculate how to reach a meeting.

The procedures implemented by the *EventManager* are:

createMeeting – This procedure is used to submit meeting information on meeting creation. The procedure takes as parameters *name*, *startTime*, *endTime*, *location*, *peopleNumber.* On success the procedure returns the *id* of the meeting.

editMeeting – This procedure is used to submit meeting information while editing a meeting. The procedure takes as parameters *name*, *startTime*, *endTime*, *location*, *peopleNumber.* All the parameters are optionals and the procedure modifies only the ones that are not null.

deleteMeeting – This procedure is used to delete a meeting. The procedure take as parameter the meeting's *id.*

getMeeting – This procedure is used to retrieve detailed information of a meeting. The procedure takes as parameter the *id* of the meeting.

getMeetings – This procedure is used to retrieve a list of meetings according to the filtering fields. The procedure takes as parameters *after* (default = now), *before* (default = now + 7 days), *near*, *radius*, *limit* (default = 15). All the parameters are optionals.

createBreak – This procedure is used to submit break information on break creation. The procedure takes as parameters *name*, *minStartTime*, *maxEndTime*, *duration.* On success the procedure returns the *id* of the break

editBreak – This procedure is used to submit break information while editing a break. The procedure takes as parameters *name*, *minStartTime*, *maxEndTime*, *duration.* All of the parameters are optionals and the procedure modifies only the ones that are not null.

deleteBreak – This procedure is used to delete a break. The procedure take as parameter the *id* of the break.

getBreak – This procedure is used to retrieve detailed information of a break. The procedure takes as parameter the *id* of the break.

getBreaks – This procedure is used to retrieve a list of breaks according to the filtering fields. The procedure takes as parameters *after* (default = now), *before* (default = now + 7 days) and *limit* (default = 15). All the parameters are optionals.

MapManager

The *MapManager* interacts with the following entities:

- *location* entity, in order to keep users position updated and calculate the location of the meeting. It also calculates the path to reach the meeting.

The procedures implemented by the *MapManager* is:

computeLocation – This procedure is used to keep the user's position updated and the location of the meeting; it takes coordinates as parameters and sends it to the external system that will calculate the position.

TravelInformationProvider

The *TravelInformationProvider* interacts with the following entities:

- *user* entity.
- *location* entity.
- *travel_mean* entity.

The procedures implemented by the *TravelInformationProvider* is:

computeTravel – This procedure computes which are the most suitable travel means to reach a certain location. The procedure takes as parameters *fromLocation*, *toLocation*, *userPreferencies.* returns a list of suitable travel means according to user's preferences.

NotificationManager

The *NotificationManager* doesn't interact with any entity

The procedures implemented by the *NotificationManager* are:

sendEmail – The procedure is used during the application processes to generate an e-mail message; this takes as a parameter the information entered by the user and those processed by external systems; the returned value is not defined because the e-mail (confirmation or error) is immediately sent.

2.7 Architectural styles and patterns

The following architectural styles and patterns have been used:

Client and server

The client-server architecture is used at different levels in the Travlendar + system design:

- The mobile application is client with respect to the Application Server that receives and processes requests.
- The users browser, which is a system client, communicates with the Web Server, which has the task of providing the requested web page.
- The Application Server assumes client role when querying the database.

Multi - tiered architecture

The multi - tiered client - server architecture allows to physically separate presentation, application processing, and data management operations. Moreover, in case of a Web Server failure, the system is accessible from the mobile application, so the number of system levels will increase in the availability of the service.

Thin client

The thin client approach has been following during the design phase for the interaction among the users machines and the system itself. System logic is implemented by Application Server, which has sufficient computing power and can manage concurrency issues in an efficient way. Mobile application and user browser does not contain a decision logic, but only the presentation function is performed. This architectural choice allows users to enjoy the service through devices with limited computing power and easy updating software.

Model View Controller

The web and mobile applications follow the Model - View - Controller software design pattern. It divides the application into three interconnected parts. This is done to separate internal representations of information from the ways information is presented to, and accepted from, the user.

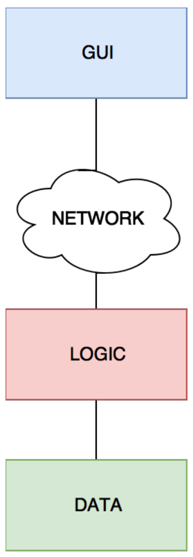

Figure 10: The remote presentation architectural style that inspired the design of web/mobile application.

2.8 Other design decisions

2.8.1 User's passwords storage

Is good practice not to store user's passwords in plain text to prevent various kinds of cyberattacks. For this reason, this kind of sensible data must be ashed and salted before storing it in the database, preventing an attacker to know its actual value.

2.8.2 Maps

As previously said, the application make use of maps to graphically show to the user the location of the events. For obvious reasons the system will make use of an external map service.

The appointed service provider will be *Google Maps*.

2.8.3 Travel Information Provider

To get informations about travel means schedules and routes we rely on an external services such as ATM APIs for the metropolitan area of Milan and Trenitalia for national travels, and so on and so forth.

Section 3

Algorithm Design

3.1 Most appropriate mean selection

As mentioned in the RASD document [1], the application are able to autonomously select the most appropriate travel mean to reach each meeting.

This decision is the output of the following algorithm:

1. Ask the *TravelInformationProvider* for a list of possible means to reach the meeting;
2. Filter the list based on the user preferences (keep a mean only if the user selected it in his/her travel preferences);
3. Check if the user wants to minimize his/her carbon footprint:
4. (a) If yes, select the less polluting travel mean.
   (b) If not, select the mean that takes the less time to reach the meeting.
5. Done.

Section 4

User Interface Design

4.1 UX diagrams

The purpose of the UX diagram is to show the different screens provided by the user interface of an application and their dynamic content. It highlights the interactions between the different screens and the presence of input forms and data required on each screen.

The diagram shown below describes both the Web and Mobile applications, and follows the user interface requirements that are provided in the RASD document [1] (Figure 4.1).

The Web and mobile applications implement the same functionality.

It is emphasized that possible user choices are represented as input modules because they are data to be sent to the system and involve updating the screen.

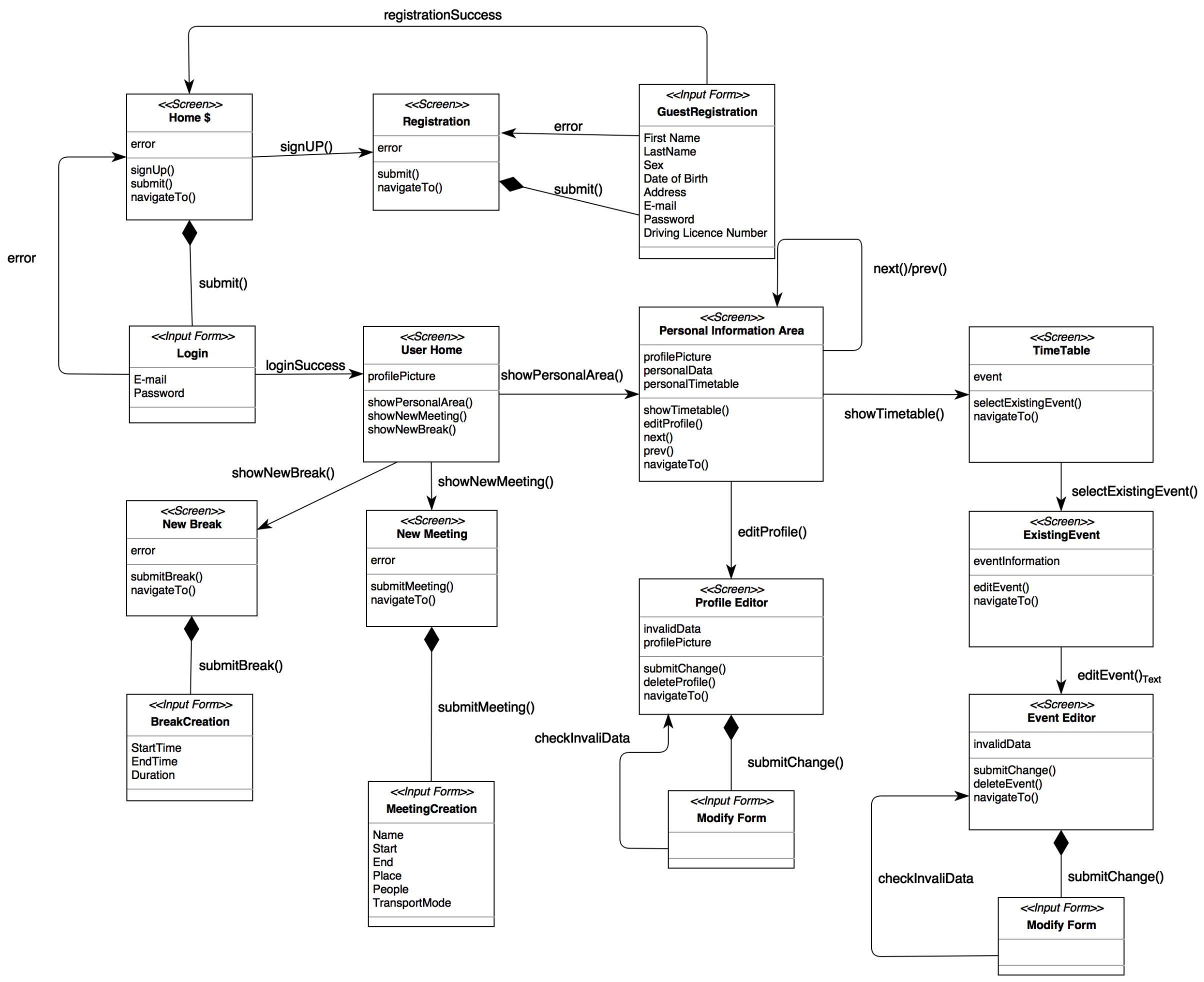

Figure 11: UX diagram of both Web and Mobile Application.

4.2 User interface

4.2.1 Web interface

The following mock-ups show how the interface of the Web Application should look like on the users browser.

4.2.2 Mobile interface

The following mock-ups show how the interface of the Mobile Application should look like on the users mobile devices.

4.2.3 Web interface

4.2.4 Mobile interface

Section 5

Requirements Traceability

5.1 Functional requirements

The following table (Table 5.1) is aimed to map the functional requirements described in the RASD document [1] to the components in the Design Document (only application logical components).

Goal	DD Components	RASD Requirements
1 2 3	UserManager	3.2.1 Register 3.2.2 Login 3.2.6 Manage profile information 3.2.7 Delete profile
4 5 6 10 11 12	EventManager	3.2.3 Create Meeting 3.2.4 Modify Meeting 3.2.5 Delete Meeting 3.2.10 Provide constraints 3.2.11 Minimize Carbon Footprint 3.2.12 Specify Break Timing
4 5	Map Manager	3.2.3 Create Meeting 3.2.4 Modify Meeting
4 5 6 8 9 10 11	TravelInformationProvider	3.2.3 Create Meeting 3.2.4 Modify Meeting 3.2.5 Delete Meeting 3.2.8 Activate Travel Mean(s) 3.2.9 Deactivate Travel Mean(s) 3.2.10 Provide constraints 3.2.11 Minimize Carbon Footprint
1 1 4 5 10 12	NotificationManager	3.2.1 Register 3.2.2 Login 3.2.3 Create Meeting (overlap) 3.2.4 Modify Meeting (overlap) 3.2.10 Provide constraints 3.2.12 Specify Break Timing

5.2 Non-functional requirements

The following table (Table 5.2) illustrates the sections of the Design Document that highlight the subjects related to the satisfaction of the non-functional requirements in the RASD document [1].

DD Section	**RASD Requirements**
4 User Interface Design	3.1.1 User Interfaces
2.6.2 Web Servers - Web Browsers	3.1.4 Communications Interfaces
2.4 Deployment View	3.4.1 Reliability
2.2 High level componets	3.4.2 Availability
2.8 Other Decisions 2.6.2 Web Servers - Web Browsers	3.4.3 Security
2.3.6 Implementation Choices	3.4.5 Portability

Section 6

Implementation, Integration and Test Plan

6.1 Elements to be integrated

The integration test phase for the Travlendar + system will be structured according to the architectural division in the levels described in the first part of this document.

The subsystems to integrate are as follows:

Database Tier – This includes all commercial database structures that will be used to store and manage system data, like the DBMS and the Database Engine; the component to be integrated is the DBMS.

Application Logic Tier – This includes all business logic for the application, data access components, and interface components to external systems and clients. All interactions must be tested and all subsystems that interact with this layer must be integrated individually.

Web Tier – This includes all components of the Web interface and communication with the application logic level and the browser client. Integration tests must be performed in both ways for this level.

Client Tier – Includes the Mobile application client, the Web browser client, and its internal components. Individual clients must behave correctly with respect to their internal structure and must be individually integrated with the level they interface with.

6.2 Integration Testing Strategy

For the integration testing process, we decide to follow a bottom-up approach.

This choice is natural since the integration testing can start from the smaller and the lowest level components, which have already been tested at the drive level, and do not depend on other components or not already developed components.

The bottom - up strategy will be mixed with a critical-module-first approach to avoid major component failures and threats to the correct implementation of the entire Travlendar + system.

The higher-level subsystems described in the previous paragraph are loosely coupled and sufficiently independent of each other because they correspond to different levels.

To this purpose, the second cited approach is used to establish the correct integration order and to obtain the complete system.

At this level of integration testing should also cover the communication functionalities with external systems.

Section A

Appendix

A.1 Software and tools used

The software and tools used during the drawing of this document are:

LaTex: used to build this document.

Google Drive / Google Documents: used to always update and share the documents.

draw.io: used to draw diagrams (https://www.draw.io).

Marvel: used to build the mockups (https://marvelapp.com).

GitHub: used as version control and to keep it shared between group members and teachers (https://www.github.com).

A.2 Hours of work

Most of the work on this document was made in the presence of both members of the group. The approximate number of hours worked by each member of the group is as follow (including hours spent in group work):

Davide Rossetto: about 23 hours
Alessandro Tatti: about 25 hours

Section B

Bibliography

[**1**] AA 2017/2018 Software Engineering 2 - Requirements Analysis and Specification Document - Davide Rossetto, Alessandro Tatti

[**2**] AA 2017/2018 Software Engineering 2 - Project goal, schedule and rules

[**3**] IEEE Standard 1016:2009 System design - Software design descriptions