TRAVLENDAR+

POLITECNICO
MILANO 1863

DESIGN DOCUMENT

26/11/2017 - v1.0

MATTEO COLOMBO   ALESSANDRO PEREGO   ANDREA TROIANIELLO

Deliverable:	DD
Title:	Design Document
Authors:	Matteo Colombo, Alessandro Perego, Andrea Troianiello
Version:	1.0
Date:	26-November-2017
Download page:	GitHub - ColomboPeregoTroianiello repository

Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose

The purpose of this document is to describe in details the concepts expressed in the RASD file, and to specify how the system will be implemented and which logical and physical components will be used.

1.2 Scope

Travlendar+ is a calendar-based application whose purpose is to help people to easily schedule working or personal meetings around Milan. Given what was written in the RASD, the main issues on which we will focus one are the following:

- Provide the services,
- Allow the user to access the system and all its functions,
- Provide a reliable system that can help the users in their daily life.

To solve these issue, the development should focus on:

- Extensibility
- Accessibility
- Reliability

For these reasons the system will be developed on a multi-tier architecture that will have to dialogue with almost standalone clients. In fact, thanks to the fact that most of the system logic is implemented on the client, the application will be more reliable and more efficient.

1.3 Definitions, Acronyms and Abbreviations

1.3.1 Definitions

- Schedule: set of meetings of the same day.
- Calendar: set of the schedules and from this you can select a specific schedule.
- Meeting: event in the schedule of the user that has a location and a start and end hour.
- Break: event that has a custom time range in which it can be schedule, with a minummum duration of 5 minutes.
- Lunch: it's a particular break with no customizable range time and minimum duration of 30 minutes.
- Travel pass: represents all type of passes (daily, weekly, monthly and yearly pass) for public transportations.
- Level: pollution level of a travel travel mean.
- Event type: the type of a meeting (e.g. family, personal, work, etc).

1.3.2 Acronyms

- RASD: Requirement Analysis and Specification Document.
- DD: Design Document
- IEEE: Institute of Electrical and Electronic Engineers.
- API: Application Programming Interface.
- JEE: Java Enterprise Edition.
- REST: REpresentational State Transfer.
- JAX-RS: Java API for RESTful Web Services.
- JPA: Java Persistent API.
- SQL: Structured Query Language.
- DBMS: DataBase Management System.
- JSON: JavaScript Object Notation.
- O/R: Object/Relational.
- UX: User eXperience.
- HTTPS: HyperText Transfer Protocol over Secure socket layer.

1.4 Reference Documents

- Specification Document: "Mandatory Project Assignments.pdf".
- Requirement Analysis and Specification Document: "RASD.pdf".
- IEEE 1016-2009 - IEEE Standard for Information Technology–Systems Design–Software Design Descriptions.
- UML guide site.

1.5 Document Structure

This document is structured in four parts:

- Part 1: the first part focuses on the design and it is composed by three parts: Architectural, Algorithm and User Interface Design.
  In the first part the document focuses on the physical and logical components, on how they are aggregated and on their functions.
  In the algorithm part, the document analyzes four of the main algorithms of the system.
  In the User Interface part, the document gives a more precise description of the UI components and explains how they are connected together.

- Part 2: in the second part the document focuses on the requirements of the system and specifies how they have been implemented and how they are connected to the components of the system.
- Part 3: in the third part the document focuses on the implementation, integration and test plan, explaining how the development team should proceed to test and implement the all the components.
- Part 4: in the fourth part the document focuses on the efforts spent to the writing of this document.

2 Architectural Design

2.1 Overview: High-level components and their interaction

This chapter describes the system components both at the physical and logical level. The main high level components of the system are the following:

- **Database:** The database server which is responsible for the data storage and retrieval. It doesn't implement any logic as it is used only for data storing purposes. This layer must guarantee that the ACID properties are respected.
- **Application server:** The application server contains all the logic on the server side of the system. This layer implements RESTful APIs and is used for registrations, login, backup and restore purposes.
- **Mobile Application:** The application consists in the client side of the application. It is installed on the users' devices and implements most of the logic of the system. For the account/backup purposes it communicates directly with the application server, while for all the other functions is standalone.

The components are structured in a three layer application, shown in the following figure.

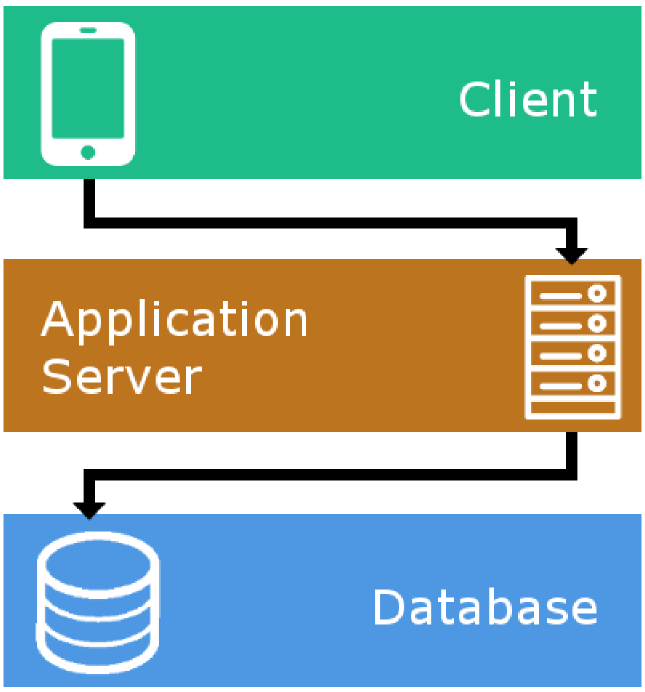


Figure 1: High Level Structure

2.2 Component View

2.2.1 High-Level Component View

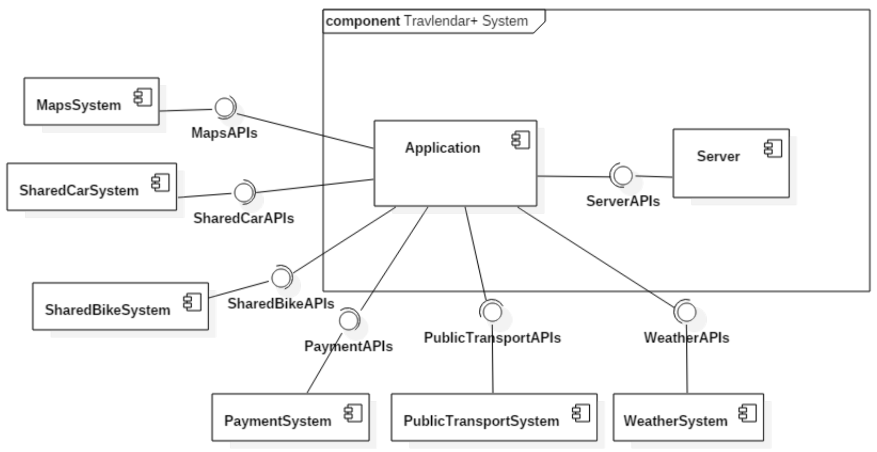


Figure 2: High Level Component View

Component to be developed:

- **Application:** It is the core of the system, it manages all information provided by the others services and performs the majority of the functions. It provides the client access to the entire system.
- **Server:** This component has an account manager and backup roles. It receives the data from the application and provides them when necessary (e.g. during the login operation).

Component to be integrated in the system:

- **MapsSystem:** It is the provider of the maps and all necessary information for the computation of the journey.
- **SharedCarSystem, SharedBikeSystem:** Those component provide all information (availability, location and costs) respectively shared cars and shared bikes.
- **PublicTransportSystem:** It provides all information about the public transport of the city.
- **WeatherSystem:** It provides the weather forecasts in the city.
- **PaymentSystem:** This component provides the payment service.

2.2.2 Application System

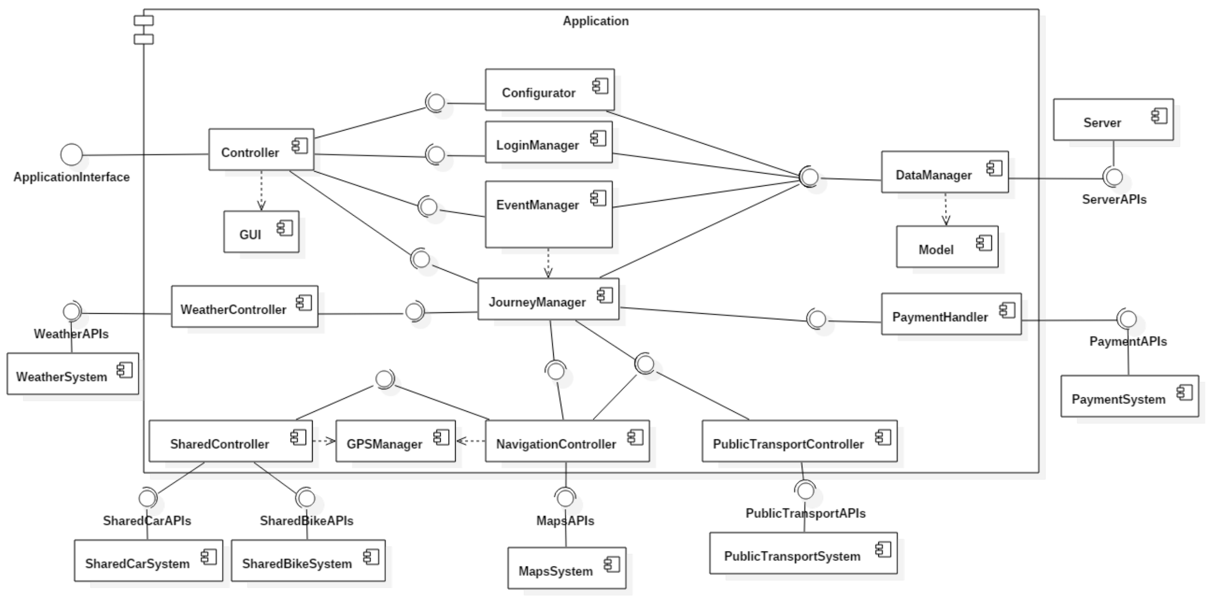


Figure 3: Application Component Diagram

- **NavigationController:** The component manages the user's navigation in real time and provides navigation utilities using the Maps APIs and GPS.
- **GPSManager:** The component that handles and gives the GPS information.
- **SharedController:** The component manages information of all shared systems.
- **PaymentHandler:** The component that handles the payment operations to buy a ticket for a public transport. It ensures that the user is able to successfully complete the payment.
- **PublicTransportController:** The component manages the availability information and the timetable of all public transport.
- **DataManager:** The component that implements and provides through an appropriate interface the methods for accessing the data of our system and it takes care to send the data to the server. This intermediate component between the entities of the the model and the other components will facilitate extendibility.
- **Model:** It represent how the data are structured (specified in RASD Class Diagram) in the application and ready to be stored by the server on the database.
- **LoginManager:** The component that handles the login operation.
- **Configurator:** The component that offers the configuration functionalities to customize a set of parameters of the user account.
- **Event Manager:** The component that handles the operations to create and manage an event.

- **Journey Manager:** It manages the user journey.
- **WeatherController:** This component manages the weather information.
- **Controller:** The component that handles the update of the GUI and the retrieval of the user input through the interfaces.
- **GUI:** Implementation of the presentation layer of the application.

2.2.3 Server System

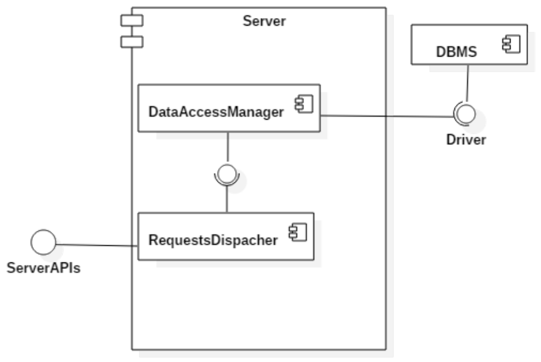


Figure 4: Server Component Diagram

- **RequestDispatcher:** It handles the requests from the application.
- **DataAccessManager:** The component that manages access to the database using a specific driver.
- **DBMS:** The system that will take care of the management of the data, integrated in our system using a specific driver.

2.3 Deployment View

This diagram purpose is to show the hardware components of our system and where the code is going to run.

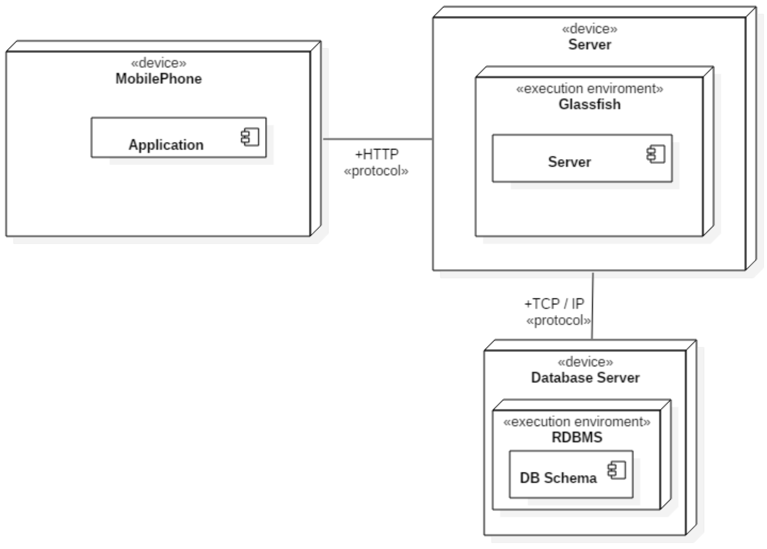


Figure 5: Deployment Diagram

2.4 Runtime View

2.4.1 Visitor Registration

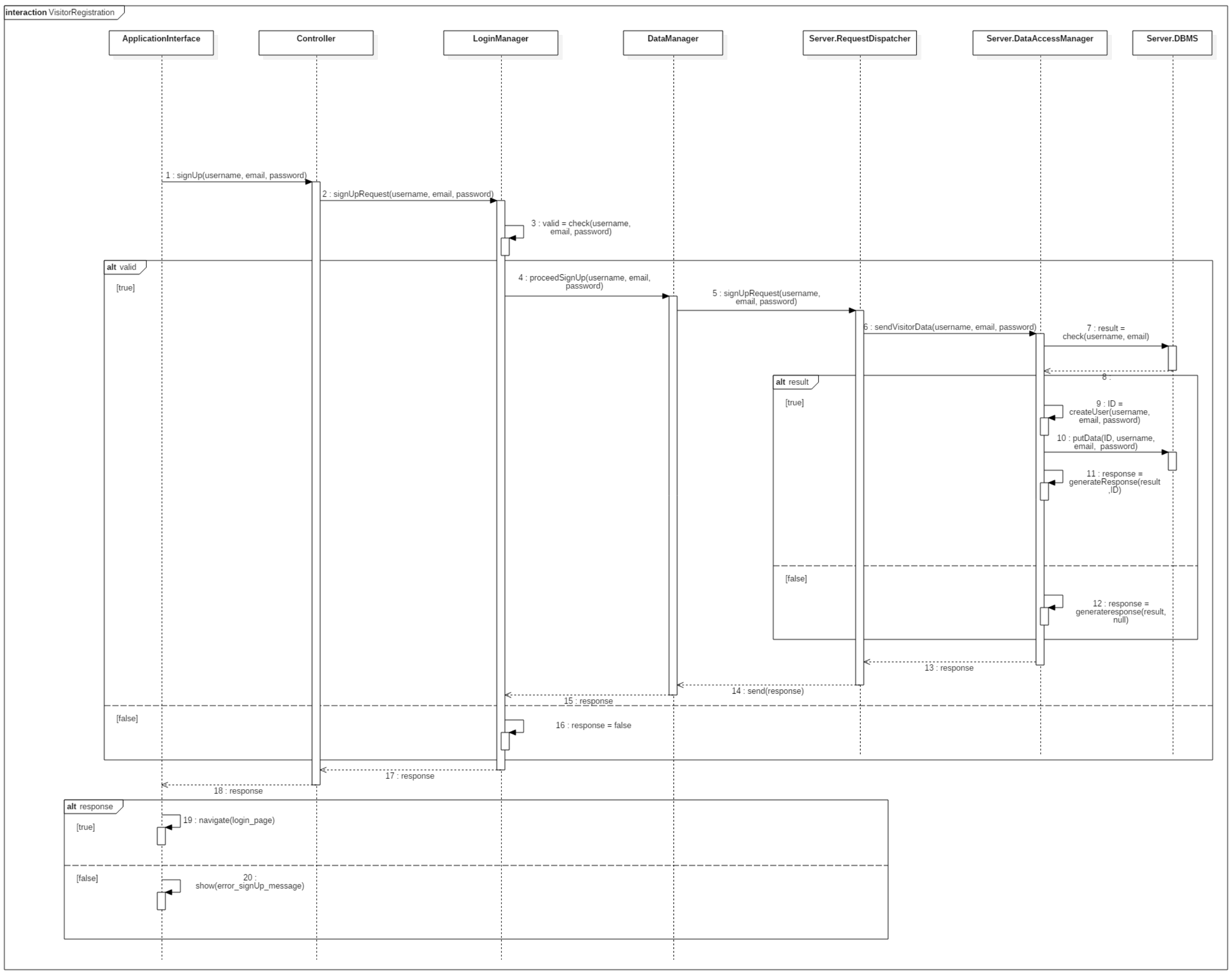

interaction VisitorRegistration
ApplicationInterface
Controller
LoginManager
DataManager
Server.RequestDispatcher
Server.DataAccessManager
Server.DBMS
1 : signUp(username, email, password)
2 : signUpRequest(username, email, password)
3 : valid = check(username, email, password)
alt valid
[true]
4 : proceedSignUp(username, email, password)
5 : signUpRequest(username, email, password)
6 : sendVisitorData(username, email, password)
7 : result = check(username, email)
8 :
alt result
[true]
9 : ID = createUser(username, email, password)
10 : putData(ID, username, email, password)
11 : response = generateResponse(result ,ID)
[false]
12 : response = generateresponse(result, null)
13 : response
14 : send(response)
15 : response
[false]
16 : response = false
17 : response
18 : response
alt response
[true]
19 : navigate(login_page)
[false]
20 : show(error_signUp_message)

Figure 6: Visitor Registration

In this runtime diagram you can see the registration process on the application. The informations are sent by the ApplicationInterface to the Controller that forwards them to the specific manager, in this case the LoginManager. The LoginManager checks if the username and the email are typed correctly or not, if not it returns a negative response. Otherwise the LoginManager proceeds with sign up process by sending the data to the DataManager which calls the server. The server manages the requests with the RequestDispatcher and sends the data to the DataAccessManager that calls the DBMS to check the availability of the username and password. If they are available, the DataAccessManager creates an ID and associates it to the user. The system returns a response the the ApplicationInterface that shows a confirmation or error dialog respectively in case of positive or negative response.

2.4.2 User Login

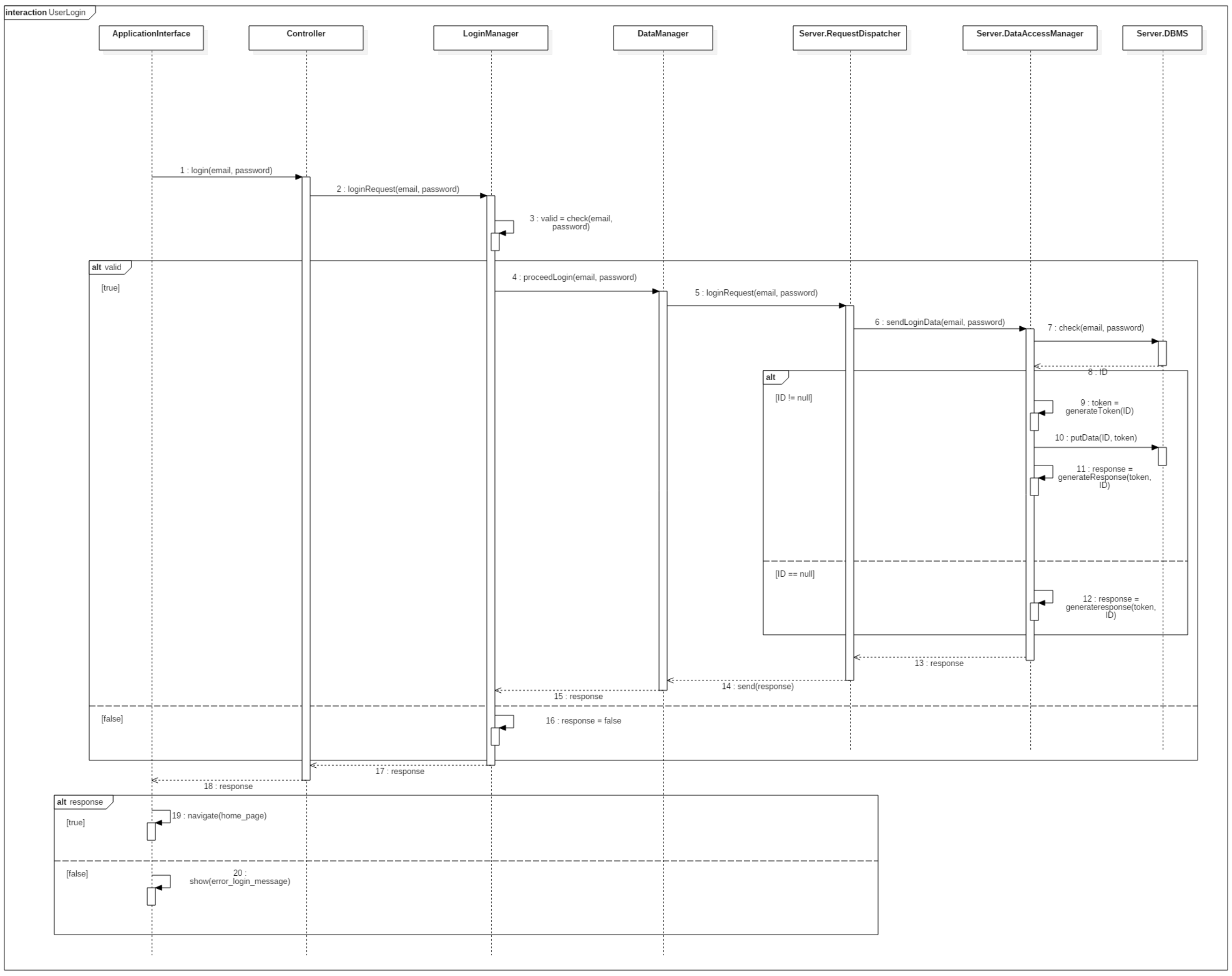

interaction UserLogin
ApplicationInterface
Controller
LoginManager
DataManager
Server.RequestDispatcher
Server.DataAccessManager
Server.DBMS
1 : login(email, password)
2 : loginRequest(email, password)
3 : valid = check(email, password)
alt valid
[true]
4 : proceedLogin(email, password)
5 : loginRequest(email, password)
6 : sendLoginData(email, password)
7 : check(email, password)
8 : ID
alt
[ID != null]
9 : token = generateToken(ID)
10 : putData(ID, token)
11 : response = generateResponse(token, ID)
[ID == null]
12 : response = generateresponse(token, ID)
13 : response
14 : send(response)
15 : response
[false]
16 : response = false
17 : response
18 : response
alt response
[true]
19 : navigate(home_page)
[false]
20 : show(error_login_message)

Figure 7: User Login

In this runtime diagram you can see the login process for a user that is already registered in the application. The login information are sent by the ApplicationInterface to the Controller that calls the LoginManager. The LoginManager checks if the parameters are typed correctly. If the parameters are not valid the LoginManager generates a negative response, if it is valid he proceeds with the login process and passes the data to the DataManager that makes a request to the server. The RequestDispatcher on the server manages the requests and queries the DBMS through the DataAccessManager to check the data. If data are already present in the DBMS, the DataAccessManager generates a token and stores it in the DBMS, then it generates a positive response; if data are not present the DataAccessManager generates a negative response. In case of positive or negative response the ApplicationInterface redirects the user to the home page or to an error message.

2.4.3 Setup Preferences

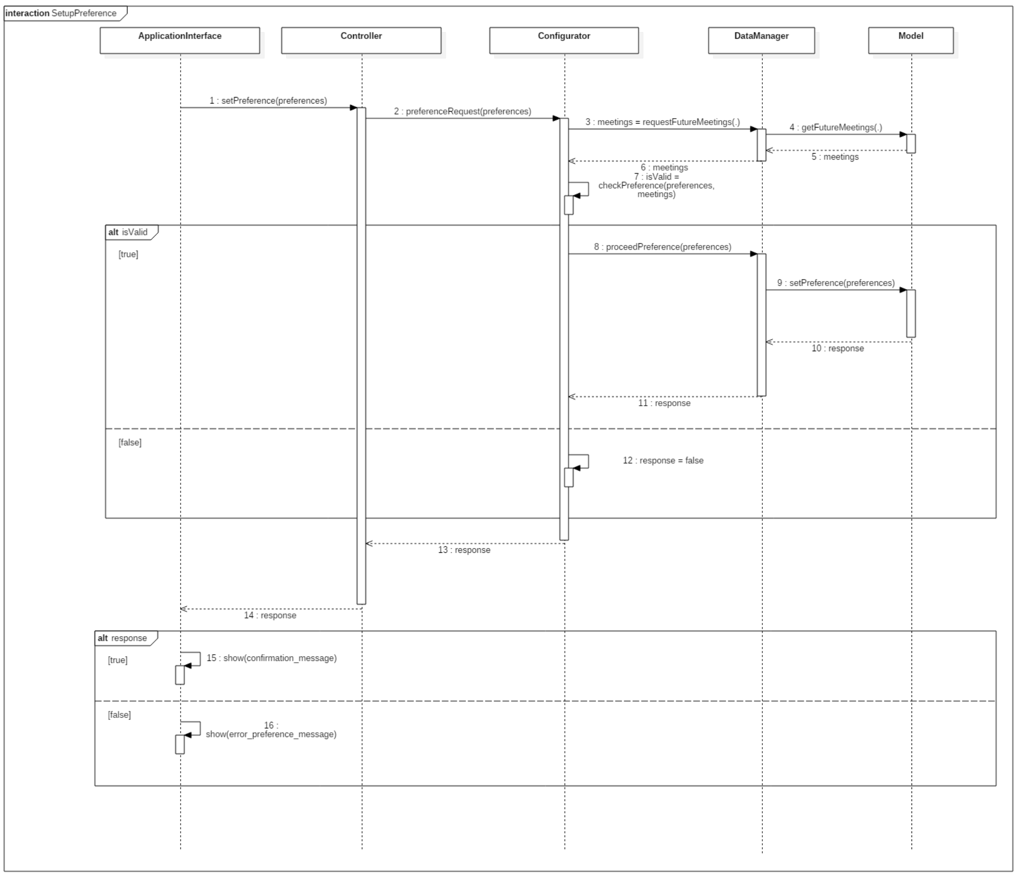

interaction SetupPreference
ApplicationInterface
Controller
Configurator
DataManager
Model
1 : setPreference(preferences)
2 : preferenceRequest(preferences)
3 : meetings = requestFutureMeetings(.)
4 : getFutureMeetings(.)
5 : meetings
6 : meetings
7 : isValid = checkPreference(preferences, meetings)
alt isValid
[true]
8 : proceedPreference(preferences)
9 : setPreference(preferences)
10 : response
11 : response
[false]
12 : response = false
13 : response
14 : response
alt response
[true]
15 : show(confirmation_message)
[false]
16 : show(error_preference_message)

Figure 8: Setup Preferences

This diagram shows how the user can set his preferences. The ApplicationInterface sends all the preferences set by the user to the Controller, which makes a request to the Configurator. The Configurator is the component that manages the preferences, it requests to the DataManager the future meetings that are already present in the Model and checks the correctness of the data. If the preferences are valid, they are passed to the DataManager that puts them in the Model; if the data are not valid it generates a negative response. Depending on the answer, the ApplicationInterface shows a confirmation or error message.

2.4.4 Create Meeting

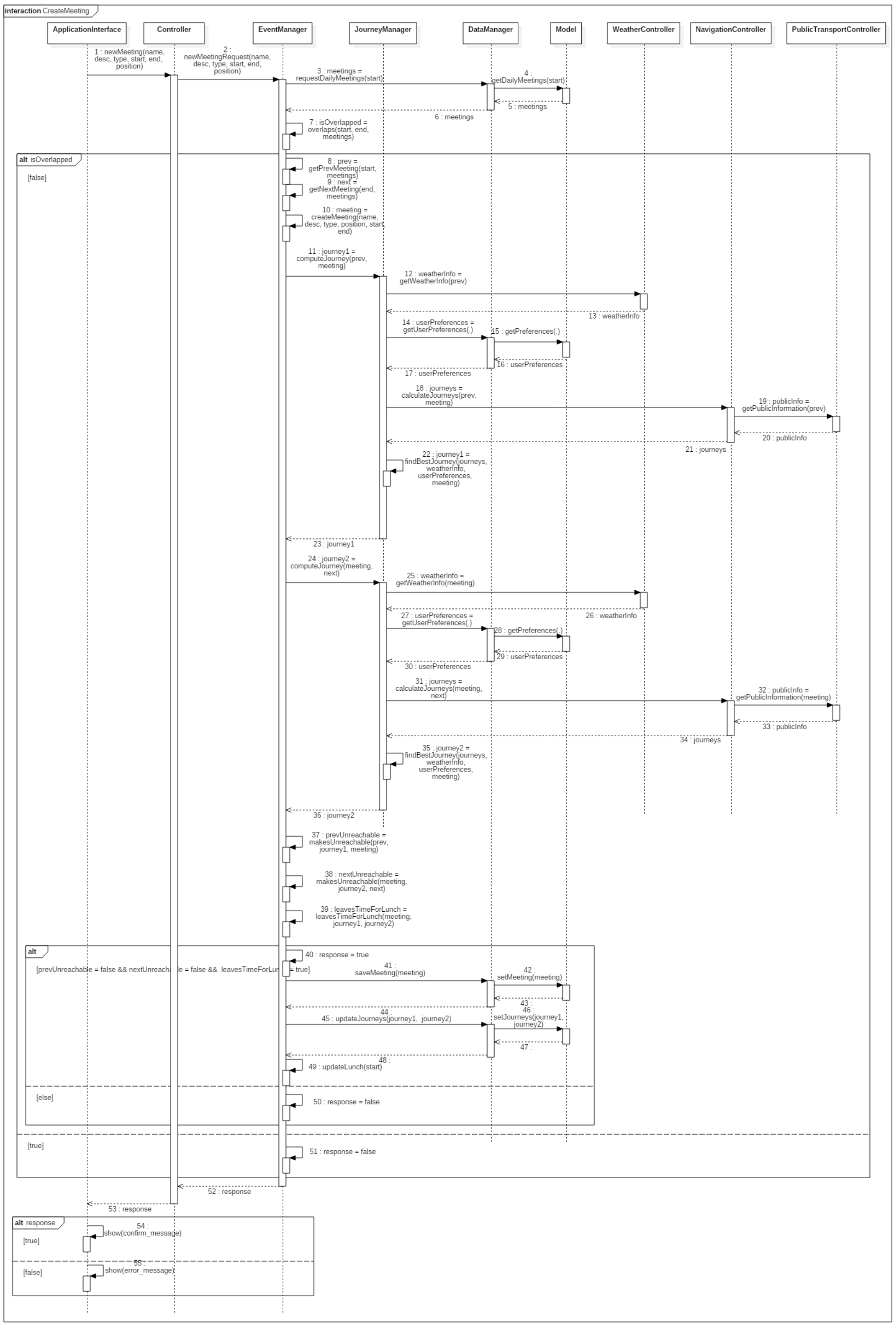

interaction CreateMeeting
ApplicationInterface
Controller
EventManager
JourneyManager
DataManager
Model
WeatherController
NavigationController
PublicTransportController
1 : newMeeting(name, desc, type, start, end, position)
2 : newMeetingRequest(name, desc, type, start, end, position)
3 : meetings = requestDailyMeetings(start)
4 : getDailyMeetings(start)
5 : meetings
6 : meetings
7 : isOverlapped = overlaps(start, end, meetings)
alt isOverlapped
[false]
8 : prev = getPrevMeeting(start, meetings)
9 : next = getNextMeeting(end, meetings)
10 : meeting = createMeeting(name, desc, type, position, start, end)
11 : journey1 = computeJourney(prev, meeting)
12 : weatherInfo = getWeatherInfo(prev)
13 : weatherInfo
14 : userPreferences = getUserPreferences(.)
15 : getPreferences(.)
16 : userPreferences
17 : userPreferences
18 : journeys = calculateJourneys(prev, meeting)
19 : publicInfo = getPublicInformation(prev)
20 : publicInfo
21 : journeys
22 : journey1 = findBestJourney(journeys, weatherInfo, userPreferences, meeting)
23 : journey1
24 : journey2 = computeJourney(meeting, next)
25 : weatherInfo = getWeatherInfo(meeting)
26 : weatherInfo
27 : userPreferences = getUserPreferences(.)
28 : getPreferences(.)
29 : userPreferences
30 : userPreferences
31 : journeys = calculateJourneys(meeting, next)
32 : publicInfo = getPublicInformation(meeting)
33 : publicInfo
34 : journeys
35 : journey2 = findBestJourney(journeys, weatherInfo, userPreferences, meeting)
36 : journey2
37 : prevUnreachable = makesUnreachable(prev, journey1, meeting)
38 : nextUnreachable = makesUnreachable(meeting, journey2, next)
39 : leavesTimeForLunch = leavesTimeForLunch(meeting, journey1, journey2)
alt
[prevUnreachable = false && nextUnreachable = false && leavesTimeForLunch = true]
40 : response = true
41 : saveMeeting(meeting)
42 : setMeeting(meeting)
43 :
44 :
45 : updateJourneys(journey1, journey2)
46 : setJourneys(journey1, journey2)
47 :
48 :
49 : updateLunch(start)
[else]
50 : response = false
[true]
51 : response = false
52 : response
53 : response
alt response
[true]
54 : show(confirm_message)
[false]
55 : show(error_message)

Figure 9: Create Meeting

This diagram shows the process to create a new meeting. The ApplicationInterface retrieves all information about the meeting that the user wants to create and sends them to the ViewController that calls the EventManager component.The EventManager requests all the meetings of the day of the new one to the DataManager and checks if there is an overlap with other events. If this is the case the event can't be created and an error message is shown. Otherwise the event is valid and the EventManager takes the information of the previous and the next events and creates an object meeting for the new event. To find the best journey the JourneyManager calls different components to retrieve the information that it needs: the WheaterController for weather information, the DataManager and the Model for the user's preferences and the NavigationController, which calls the PublicTransportController for the information about hours and availability of public transportations and then calculates all the possible journeys. After that the JourneyManager computes the best journey within all the available ones, taking in consideration the user's preferences and the weather information. Eventually, a control is performed to check if the events would be reachable and if the travels leave enough time for lunch. Finally if all the controls succeed the EventManager saves the meeting into the Model, updates the relative journeys, the lunch time and returns a response to the ApplicationInterface that shows a confirmation or error message.

2.4.5 Travel to the Meeting

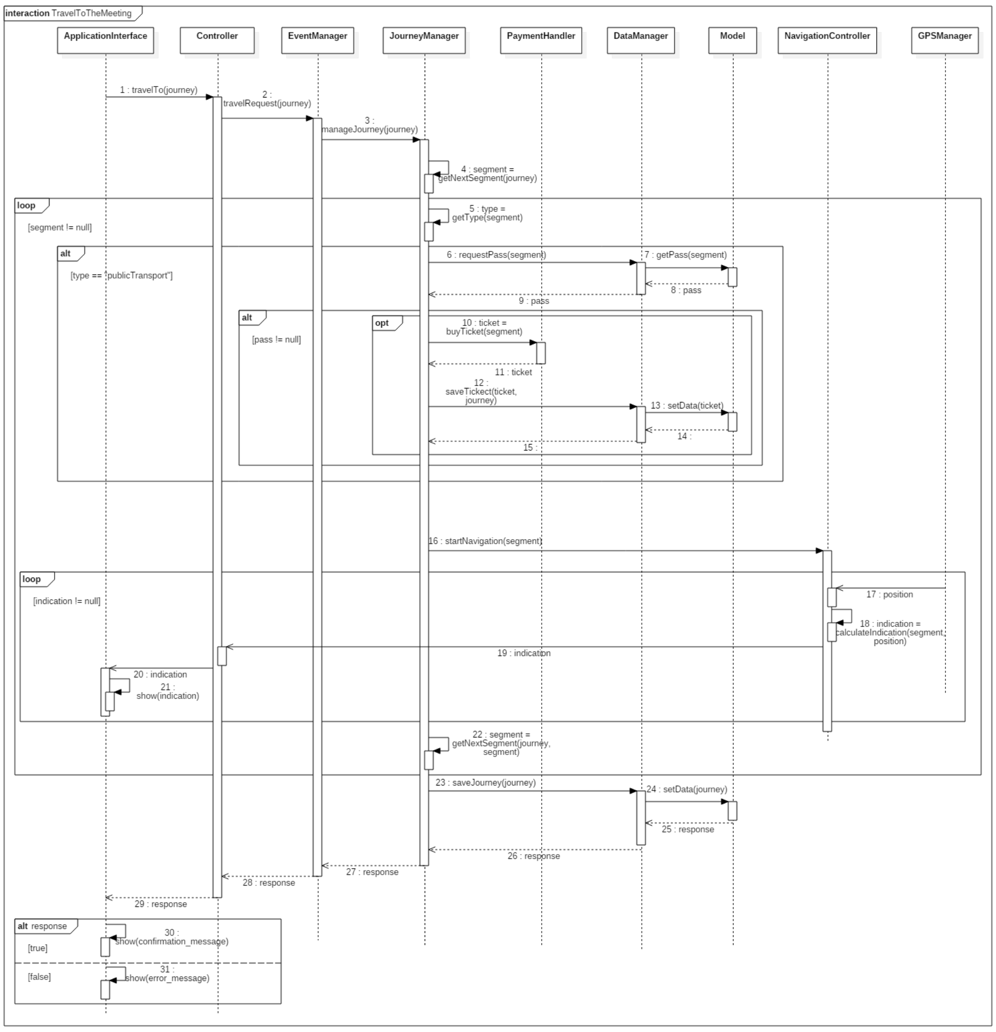
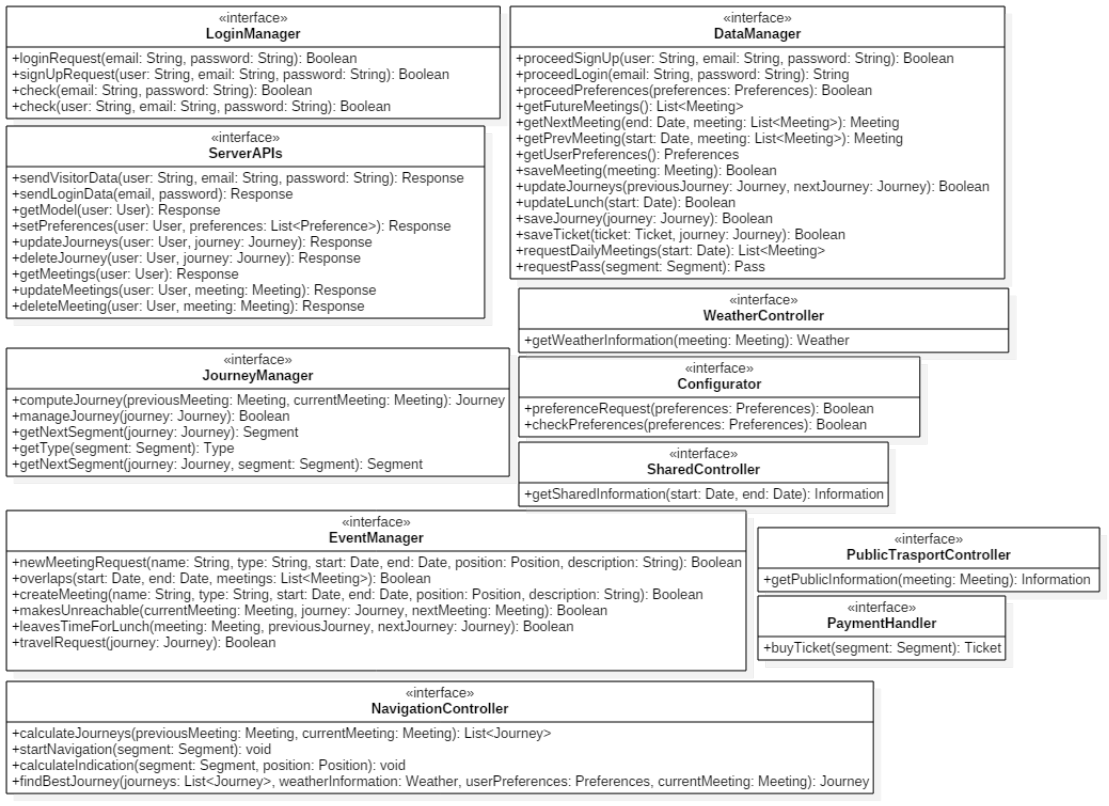


Figure 10: Travel to the Meeting

This diagram shows the flow of method calls and events that are performed when the user is traveling to a meeting. The user starts the navigation in the ApplicationInterface which, through the ViewController and the EventManager, passes the relative journey to the JourneyManager. For each segment that composes the journey, the JourneyManager reads the travel mean, and in case it is a public transport mean, it requests to the DataManager if there is a relative ticket. If there isn't, the user can opt to buy one. In case a new ticket is bought, the application saves it and its information in the model. Then, when the users starts the navigation the Navigation-Controller begins showing directions. Every time the user position changes, the GPS Manager sends an asynchronous call to the Navigation Controller. From the new position, the NavigationController calculates the directions and sends them to the ApplicationInterface which shows them to the user. The navigation ends when the user location is equal to the final destination of the segment. When the navigation ended and the user arrives at the meeting location, the JourneyManager updates the journey information in the Model and saves it as completed; then it send a response to the ApplicationInterface that shows a confirmation or error message to the user.

2.5 Component Interfaces

The server communicates with the DBMS via the JPA over standard network protocols. Thus, the DB and the server layers can be deployed on different tiers, as well on the same one.
The low-level technicalities about the specific dialect of SQL for the selected DBMS are abstracted by the JPA, which also deals with the O/R mapping.
The mobile app shall communicate with the server using the back-end programmatic interface presented in the component view and implemented as a RESTful interface over the HTTPS protocol. The RESTful interface is implemented in the application server using JAX-RS and uses JSON as the data representation language.



```
«interface»
LoginManager
+loginRequest(email: String, password: String): Boolean
+signUpRequest(user: String, email: String, password: String): Boolean
+check(email: String, password: String): Boolean
+check(user: String, email: String, password: String): Boolean

«interface»
DataManager
+proceedSignUp(user: String, email: String, password: String): Boolean
+proceedLogin(email: String, password: String): String
+proceedPreferences(preferences: Preferences): Boolean
+getFutureMeetings(): List<Meeting>
+getNextMeeting(end: Date, meeting: List<Meeting>): Meeting
+getPrevMeeting(start: Date, meeting: List<Meeting>): Meeting
+getUserPreferences(): Preferences
+saveMeeting(meeting: Meeting): Boolean
+updateJourneys(previousJourney: Journey, nextJourney: Journey): Boolean
+updateLunch(start: Date): Boolean
+saveJourney(journey: Journey): Boolean
+saveTicket(ticket: Ticket, journey: Journey): Boolean
+requestDailyMeetings(start: Date): List<Meeting>
+requestPass(segment: Segment): Pass

«interface»
ServerAPIs
+sendVisitorData(user: String, email: String, password: String): Response
+sendLoginData(email, password): Response
+getModel(user: User): Response
+setPreferences(user: User, preferences: List<Preference>): Response
+updateJourneys(user: User, journey: Journey): Response
+deleteJourney(user: User, journey: Journey): Response
+getMeetings(user: User): Response
+updateMeetings(user: User, meeting: Meeting): Response
+deleteMeeting(user: User, meeting: Meeting): Response

«interface»
WeatherController
+getWeatherInformation(meeting: Meeting): Weather

«interface»
JourneyManager
+computeJourney(previousMeeting: Meeting, currentMeeting: Meeting): Journey
+manageJourney(journey: Journey): Boolean
+getNextSegment(journey: Journey): Segment
+getType(segment: Segment): Type
+getNextSegment(journey: Journey, segment: Segment): Segment

«interface»
Configurator
+preferenceRequest(preferences: Preferences): Boolean
+checkPreferences(preferences: Preferences): Boolean

«interface»
SharedController
+getSharedInformation(start: Date, end: Date): Information

«interface»
EventManager
+newMeetingRequest(name: String, type: String, start: Date, end: Date, position: Position, description: String): Boolean
+overlaps(start: Date, end: Date, meetings: List<Meeting>): Boolean
+createMeeting(name: String, type: String, start: Date, end: Date, position: Position, description: String): Boolean
+makesUnreachable(currentMeeting: Meeting, journey: Journey, nextMeeting: Meeting): Boolean
+leavesTimeForLunch(meeting: Meeting, previousJourney, nextJourney: Journey): Boolean
+travelRequest(journey: Journey): Boolean

«interface»
PublicTrasportController
+getPublicInformation(meeting: Meeting): Information

«interface»
PaymentHandler
+buyTicket(segment: Segment): Ticket

«interface»
NavigationController
+calculateJourneys(previousMeeting: Meeting, currentMeeting: Meeting): List<Journey>
+startNavigation(segment: Segment): void
+calculateIndication(segment: Segment, position: Position): void
+findBestJourney(journeys: List<Journey>, weatherInformation: Weather, userPreferences: Preferences, currentMeeting: Meeting): Journey
```

Figure 11: Component Interfaces

This diagram shows the most relevant parts of the interfaces that our system is going to use for communication between the various components.

2.6 Selected Architectural styles and patterns

The following architectural styles and patterns have been used:

- **Observer pattern:** This pattern is used between NavigationController and Controller. It allows the NavigationController to update the user's view with new directions in a transparent way when the position changes.
- **Model-Control-View:** It is used for the main components of the system. It's a really good choice of design that allows to keep clear the role of every component and that makes the system easier to deploy and maintain.
- **Client-Server:** This pattern is a good practice for a distributed system. It is used between the application server (client) queries the DB (server) and the application (client) communicates with the application server.
- **Service-oriented Architecture:** It is used by the system for the communication between the server and the user's device (RESTful). The SOA allows to think at a higher level of abstraction, by looking at the component interfaces and not at their specific implementation. SOA style also improves modularity: by making service description, discovery and binding explicit, it is easier to build new plugins and test single modules independently.
- **Fat Client:** The fat client paradigm is implemented because the interaction between user's device and the server hasn't a central role in the system behavior . Having a fat client in our case is an advantage because most application logic is on the user's device, which has sufficient computing power and is able to manage concurrency issue efficiently.

3 Algorithm Design

This chapter presents some of algorithms of the application; their code is written in a Java-like pseudocode.

3.1 Is Overlapped

This algorithm is used during the process of creation of a meeting and checks wether the new meeting that the user wants to create would be overlapped with another already scheduled. In case of overlaps, the method returns true.

```
boolean isOverlapped(Meeting meeting, List meetings){
    for(i=0; i< meetings.length; i++){
        if(meetings[i].start >= meeting.start && meetings[i].start <=
    meeting.end){
            return true;
        }else if(meetings[i].end >= meeting.start && meetings[i].end<=
    meeting.end){
            return true;
        }
    }
    return false;
}
```

3.2 Create Meeting

This algorithm is used during the process of creation of a meeting and checks if, given an already scheduled meeting and given the computed journey to reach the second one, the second meeting is reachable or not. In case the second meeting is unreachable, the method returns true.

```
boolean makesUnreachable(Meeting prev,Journey journey,Meeting next){
    if(prev.end + journey.duration > next.start){
        return true;
    }
    return false;
}
```

3.3 Check User Preferences

This algorithm is used to check if the new setting selected by the user would create a conflict with an already planned (future) meeting. If any of the preferences would create a conflict, the method returns false.

```
boolean checkPreferences(List preferences, List meetings){
    today= getDate().day;
    for each pref in preferences{
        if(generatesConflicts(pref, today, meetings)){
            return false;
        }
    }
    return true;
}
```

3.4 Find Best Journey

This algorithm is used to filter the journeys list to find the best one for the user. The function takes in consideration all the constraints depending on the journey length, on the weather and on the meeting. Then, all the journeys are ordered according to a criterion (footprint, duration), eventually the best journey that also respect the "prefered travel mean for the meeting type" preference is selected and returned.

```
Journey findBestJourney(List journeys, Weather weatherInfo, Preferences userPreferences, Meeting meeting){
     List validJourneys;

     for(i=0; i < journeys.size(); i++){
          boolean valid = true;
          Journey j = journeys.get(i);
          Segment segments = j.getSegments();
          for(h=0; h< segments.length && valid; h++){
               valid = (segments[h].getTravelMean()).checkIfConstraintsAreRespected(segments[h],weatherInfo, meeting);
          }
          if(valid){
               validJourneys.add(j);
          }
     }

     if(userPreferences.getFootprintPref()){
          validJourneys.orderByFootprint("asc");
     }else{
          validJourneys.orderByDuration("asc")
     }

     if(meeting.getType().hasPreference()){
          TravelMean preferedTravelMean= meeting.getType().getPreferedTravelMean();
          int index = validJourneys.getIndexJourneyWPreferedTM(preferedTravelMean);
          if(index >0){
               TravelMean.moveToFirst(index);
          }
     }

     return validJourneys.get(0);
}
```

4 User Interface Design

Examples of mockups were already provided in the RASD file. Hence in this section it is provided the UX diagram of the application.

4.1 UX Diagram

The UX diagram describes in detail all the the pages of the application, with the exception of the TravelMeanNavPage and the Settings page that to keep the clarity and cleaness of the diagram are not expanded and their input form are not shown.

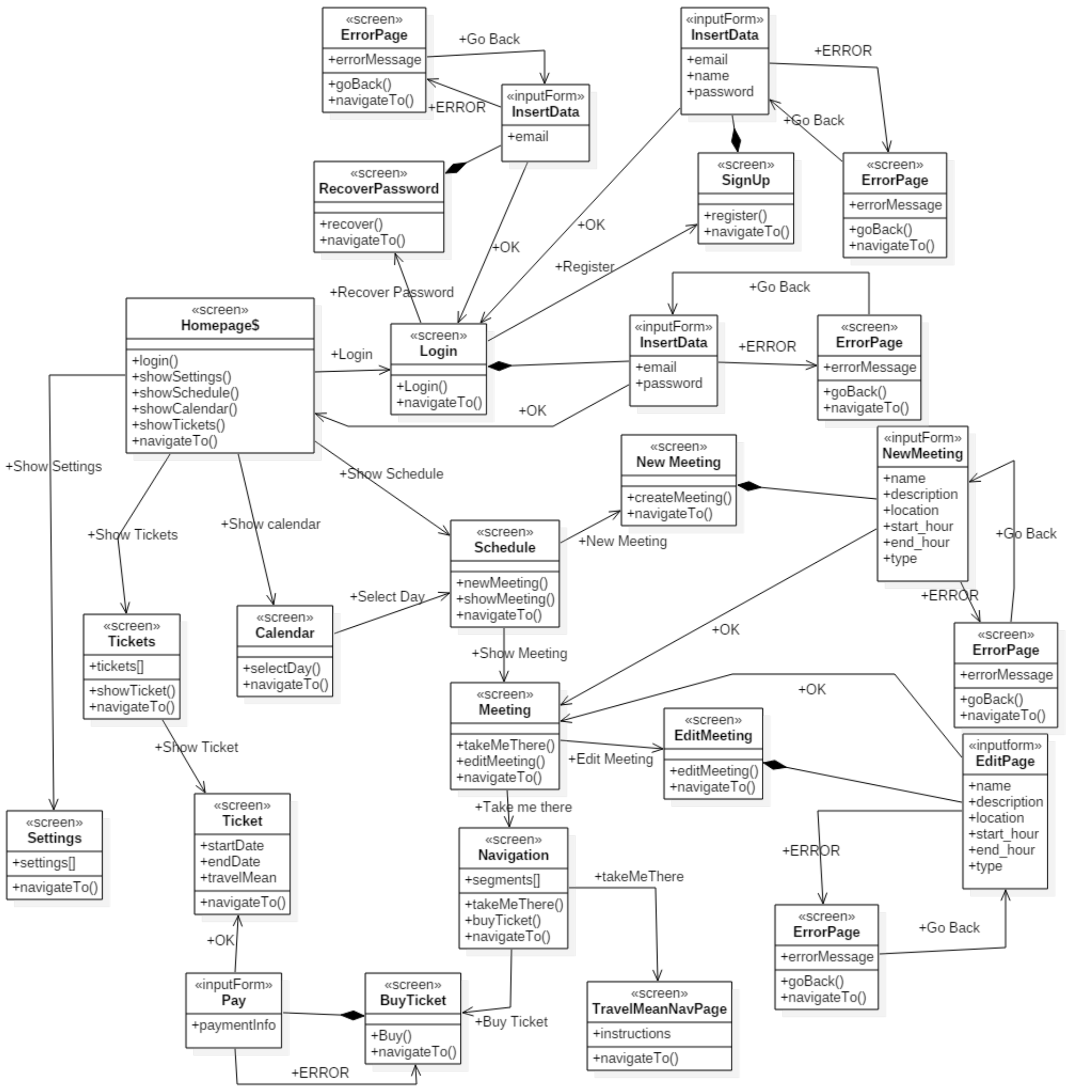

«screen» ErrorPage +errorMessage +goBack() +navigateTo()
+Go Back
+ERROR
«inputForm» InsertData +email
«screen» RecoverPassword +recover() +navigateTo()
+OK
«inputForm» InsertData +email +name +password
+ERROR
+Go Back
«screen» SignUp +register() +navigateTo()
«screen» ErrorPage +errorMessage +goBack() +navigateTo()
+OK
+Register
+Recover Password
+Go Back
«screen» Homepage$ +login() +showSettings() +showSchedule() +showCalendar() +showTickets() +navigateTo()
+Login
«screen» Login +Login() +navigateTo()
«inputForm» InsertData +email +password
+ERROR
«screen» ErrorPage +errorMessage +goBack() +navigateTo()
+OK
+Show Settings
+Show Schedule
«screen» New Meeting +createMeeting() +navigateTo()
«inputForm» NewMeeting +name +description +location +start_hour +end_hour +type
+Go Back
+Show Tickets
+Show calendar
«screen» Schedule +newMeeting() +showMeeting() +navigateTo()
+New Meeting
+Select Day
+ERROR
«screen» Tickets +tickets[] +showTicket() +navigateTo()
«screen» Calendar +selectDay() +navigateTo()
+OK
«screen» ErrorPage +errorMessage +goBack() +navigateTo()
+Show Meeting
+OK
«screen» Meeting +takeMeThere() +editMeeting() +navigateTo()
«screen» EditMeeting +editMeeting() +navigateTo()
+Edit Meeting
«inputform» EditPage +name +description +location +start_hour +end_hour +type
+Show Ticket
+Take me there
«screen» Settings +settings[] +navigateTo()
«screen» Ticket +startDate +endDate +travelMean +navigateTo()
«screen» Navigation +segments[] +takeMeThere() +buyTicket() +navigateTo()
+takeMeThere
+ERROR
«screen» ErrorPage +errorMessage +goBack() +navigateTo()
+Go Back
+OK
«inputForm» Pay +paymentInfo
«screen» BuyTicket +Buy() +navigateTo()
+Buy Ticket
«screen» TravelMeanNavPage +instructions +navigateTo()
+ERROR

Figure 12: UX Diagram

5 Requirements Traceability

All the decisions in the DD have been taken following functional and nonfunctional requirements written in the RASD. In this chapter, each requirement is mapped to specific component.

- **LoginManager:**
  - [R.1.1] An account requires an username.
  - [R.1.2] An account requires an email address.
  - [R.1.3] An account requires a password.
  - [R.1.4] The email address must be unique.
  - [R.2.1] The user must be logged in the application.
- **EventManager:**
  - [R.2.2] A meeting requires a location, which is either coordinates or an address.
  - [R.2.3] A meeting requires a starting and ending hour.
  - [R.2.4] A meeting requires a name.
  - [R.2.5] A meeting requires a type.
  - [R.2.6] A meeting may have a description.
  - [R.3.1] A warning should be given at the creation of a meeting if it is unreachable or if it is causes conflicts.
  - [R.3.2] A warning should be given in case of exceptional events that make one or more meetings unreachable.
  - [R.3.3] A warning forbids the user to force the creation of a meeting.
  - [R.6.1] The minimum length of a lunch break is 30 minutes.
  - [R.6.2] The lunch break must be scheduled between 11.30 am and 2.30pm.
  - [R.7.1] The minimum length of a lunch break is 5 minutes.
- **Configurator:**
  - [R.4.2] Different type of meetings should be reached with different/specific travel means.
  - [R.4.3] Different travel means should be suggested depending on the hour of the day.
  - [R.4.4] Travel means can have constraints on the length of their journey section.
  - [R.5.1] The application must support different travel means and transportation systems.
  - [R.5.2] Users can select constraints on travel means.
  - [R.5.3] The application must show results consistent with the users preferences.
  - [R.5.4] Users should be able to specify the travel passes they own.
  - [R.5.5] Users must specify their home address.

  - [R.5.6] Users can disable travel means.
  - [R.5.7] Users can enable the option to minimize air pollution.
  - [R.7.2] Users can set as many breaks as they want.
  - [R.7.3] Users can select the time slot in which they want their break to be scheduled.

- **JourneyManager:**
  - [R.4.1] Different travel means should be suggested depending on the weather conditions.
  - [R.4.3] Different travel means should be suggested depending on the hour of the day.
  - [R.8.1] The application should notify the user when he has to get off from a transportation system.
  - [R.8.2] The applications should give directions to the users when they are driving, cycling or walking.

- **PaymentHandler:**
  - [R.8.3] Users can buy travel passes through the application.

- **SharedController:**
  - [R.8.4] The application must locate the nearest vehicle of the selected sharing system.

6 Implementation, Integration and Test Plan

The project will be developed with a bottom-up approach, in this way the system can be divided in groups of components that can be developed independently. Furthermore the testing phase will be done in parallel with the implementation. The development of the server and of the mobile applications will proceed in parallel. The most critical part for both the client and the server is the model, which is the first component that will be developed. Then, the development of the application will move on all the standalone components that do not need the connection to the server to work. In this way we will obtain a fully functional application, that could work without a server and without user's account and backup functions. Particular attention will be given to the testing of all the functions, in particular we will focus on the functions that require an internet connection and a GPS system. Navigation simulation will be performed to test the navigation system and its reliability. Intensive tests will be performed to check if the Event Manager complies with all the functional requirements of the application. Other tests will be performed to check if the non-functional requirements are satisfied. Instead, the development of the server, after the implementation of its model will move to the implementation of the connection part, so that it could be fully functional before the implementation of the account and backup functions in the clients. Tests will concentrate on the security and reliability aspect; the security of the system will be tested to check if there are vulnerabilities that could compromise the user data, while reliability will be tested together with durability. The reliability of the system will be tested to assure that the requirements are respected, whereas durability test will ensure that in case of problems the user data wouldn't be subject to losses or damages. Once the network part of the server is completed and once the server is fully functional, we will proceed with the development of the components of the mobile applications that require a server. These parts will be the last to be developed and, given the fact that most of these depend on the server, the effort in testing them won't be minimal respect to the one spent for the other components. When all the part will be completed and the system will be deployed, we will continue performing test and checks to guarantee the efficiency and reliability of the system and to avoid possible problems introduced by mobile systems updates.

7 Effort Spent

7.1 Time Spent

Member	Hours
Colombo Matteo	40
Perego Alessandro	40
Troianiello Andrea	40

Table 1: Time Spent

The commits on GitHub are not completely representative of the work done, a good part of it has been done in group.

7.2 Used Tools

1. TexMaker for the editing of the LaTeX document.
2. Star UML for the creation of UML diagrams (Component view, Deployment and Sequence diagrams), UX diagram and Component Interfaces.
3. Visual Studio Code for the code editing.
4. Gimp for image editing.