Politecnico di Milano

Software Engineering 2

Design document

Version 1.1 - 27/12/2017

Pietro Melzi, Alessandro Pina, Matteo Salvadore

AA 2017-2018

Contents

Chapter 1

INTRODUCTION

1.1 Purpose

This document has the purpose to provide a deeper technical description of the system, already specified in the RASD document, by illustrating the main architectural components as well as their interfaces and their interactions.
We will share with all the interested parties a more detailed description of how Travlendar+ is designed and architected, with a particular emphasis on which design decisions the development team has made and the rationale behind them.
Here we will also include information about how the system will be actually implemented, integrated and tested.

1.2 Scope

Travlendar+ is a service based on a mobile application and a web application.
The system aims to help his users with a calendar-based application that support the users, computing travels between his events, helping them to organize their schedule and allowing them to arrange their trips.

For further details see the Scope section in the RASD document.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- *Arrange trip*: the system provides all the information available about a travel to the user. If the travel is related to one or more public travel means, the system helps the user to organize it. It is indicated if the user already holds a valid ticket for a certain travel or if he has to buy a required one. Purchase of a ticket is handled on the websites of the transport service providers. Trip and travel are synonymous.

- *Best path*: it is the preferred travel option proposed to reach the event location among all the feasible paths, according to parameters specified by the user. The best path can be chosen according to one of these features: length, cost or environmental sustainability. Best path is the path taken into account and showed in the daily schedule; the user can substitute it at any moment with an alternative feasible path.

- *Break event*: it is an optional event whose starting and ending time are flexible. The user can define this kind of event when he wants to reserve a certain amount of time in the

schedule and he doesn't need to specify a starting time. For instance, a user could be able to specify that lunch must be possible every day between 11:30- 2:30, and it must be at least half an hour long, but the specific timing is flexible. The app would then be sure to reserve, if possible, at least 30 minutes for lunch each day.

- *Constraint*: it is a rule defined on travel means by the user. When the system calculates feasible paths, each constraint must be taken into account: travel options not respectful of existing constraints are ignored. A constraint can be associated to a type of event.
  There are different types of constraints: min/max distance allowed for a travel mean, interval of hours in which it is possible to take a travel mean and possibility to deactivate a travel mean.

- *Current Unix Timestamp*: it is the number of seconds since Jan 01 1970. (UTC)

- *Customized type of event*: it is a set of rules related to a particular type of event that can be used several time by the user. The user can define a customized type of event in hs preferences, starting from the default type of event.

- *Default type of event*: it is a general set of rules defined usually the first time that the user exploits the system, it can be modified. Default type of event is related to each event that doesn't need a particular type of event. New types of event are defined starting from the constraints enclosed in the default one.

- *Feasible path*: a path that allows the user to reach a specified location before the starting time of the event to attend. It observes the constraints defined for that event.

- *Google Maps API*: a set of functions offered by Google Maps to decide the travel path between two locations.

- *Overlapping event*: an event (or a part of it) that happens in the same time slot of another event, added previously in the schedule. Because the schedule must be feasible, an event is considered as "overlapping" also if only its related travel overlaps an event present in the schedule.

- *Periodicity*: it is related to events that occur more than one time. It can be defined as weekly, monthly or in which days of the week the event happen.

- *Schedule*: a daily plan containing a set of events inserted by the user that allows him to travel and attend all the events. If an event overlaps other events, it cannot be inserted into the schedule.

- *Type of event*: a set of rules (constraints) that is defined by the user and can be associated to multiple events.

- *Transport service provider*: a public or private company that controls and supplies the transport with a travel mean.

- *Travel*: it is used to indicate the path that the user has to follows in order to reach a location. It can be composed by different travel components.

- *Travel component*: each single path that can be traveled with a travel mean. It has a starting time, a ending time, a departure location, an arrival location and a length. The union of one or more travel components creates a travel.

1.3.2 Acronyms

- *API*: Application Programming Interface;
- *ACID*: Atomicity, Consistency, Isolation and Durability;
- *DB*: Data Base;
- *DBMS*: Data Base Management System;
- *DD*: Design document;
- *GCM*: Google Cloud Messaging;
- *GTFS*: General Transit Feed Specification;
- *HTTP*: HyperText Transfer Protocol;
- *ICSEA*: Tenth International Conference on Software Engineering Advances;
- *Java EE*: Java Enterprise Edition;
- *RASD*: Requirement Analysis and Specification Document;
- *RESTful*: REpresentational State Transfer;
- *RSA*: Rivest-Shamir-Adleman algorithm;
- *UTC*: Coordinated Universal Time.

1.3.3 Abbreviations

- *GMaps*: Google Maps;
- *I&T*: Integration and Testing;
- *TSP*: Transport Service Provider.

1.4 Revision history

2017/11/26 - *Version 1.0* - First delivery of DD.
2017/12/27 - *Version 1.1* - Small fixes in RESTful external interfaces and in swap algorithm.

1.5 Reference Documents

- Specification Document: "Mandatory Project Assignments.pdf";
- Requirements Analysis and Specification Document, version 1.1.

1.6 Document Structure

This Design Document is composed by eight chapters:

1. The first chapter is an introduction to DD and contains also other information: definition of the used terms, revision history of the document and references;
2. The second chapter is dedicated to the architecture of the system. In this section is given a description of different components, providing information on their deployment, on the offered functionalities and on the interaction between them. The last section of the chapter explains techniques and design decisions used to realize several aspects of the system;
3. The third chapter includes a general description of the main algorithms to be used by the system-to-be;
4. The fourth chapter explains how the user can interact with the system: here are shown mockups and user experience diagrams;
5. The fifth chapter contains a table that explains the relations between a component of the system and the requirements that this component satisfies;
6. The sixth chapter refers to the realization phase of the product: it describes and establishes rules of precedence for implementation, integration and testing of the system;
7. The seventh chapter contains a report of the hours spent to write this document;
8. The eighth chapter contains references to external documents used in this document and to the software used in order to write this document.

Chapter 2

ARCHITECTURAL DESIGN

2.1 Overview

In this section we will present a general overview of Travlendar+, with a specific focus on the main logical components and their interactions.

The main high-level components of the system-to-be are:

Mobile App:

The Mobile App is a presentation layer that lets the users access the functionalities offered by the system-to-be on their smartphones and tablets. The mobile application offers also an internal logic that handles:

- the notifications reception, through the services offered by Google Cloud Messaging APIs;
- the map and path drawing functionalities offered through the interaction with Google Maps mobile APIs.

Web Browser:

The web browser is a second presentation layer that lets the users access the functionalities offered by Travlendar+ on their browsers. It relies on the connection with the web server in order to obtain dynamic web pages.

Web Server:

This layer provides dynamic web pages for the web-based application, it communicates directly with the application server to satisfy the client requests, using proper interfaces. This layer interacts also with an external system (Google Maps API) in order to display an embedded map containing the user travel's information.

Application Server:

This is the logic layer that implements all the core functionalities of the system. It receives and replies to client's requests, if required it sends notifications to the mobile applications, it interacts with external systems in order to satisfy the user's requests and it interacts with a DBMS in order to guarantee the information's persistence.
In particular, the application server will interact with these external systems:

- Google Maps APIs, in order to provide feasible paths according to the user preferences;

- Transport Service provider's systems, in order to offer functionalities that allow the user to arrange his trips, such as ticket buying, location of sharing vehicles and strikes information;
- Weather APIs, in order to retrieve weather information and to apply possible travel constraints related to the weather;
- Google Cloud Messaging APIs, in order to send notifications to the user's mobile apps.

Database Server:

The data layer, that supports all data storage and management operations. This layer ensures that ACID properties are satisfied. The database server will interact only with the application server, so that the data will never be exposed directly to the client layers.

External Systems:

Those systems are not internal components of our application, but the system-to-be will have to interact with them in order to guarantee all the system functionalities.
Most of their interactions with the system have already been described in the previous paragraphs, but here we will explain in detail the interaction with Transport Service providers: Travlendar+ will initially integrate some external transport services systems (public transport service providers and sharing vehicles providers) but will also offer proper APIs to allow other Transport Service providers to interact with Travlendar+ and therefore to be considered as suggested travel means to the users.

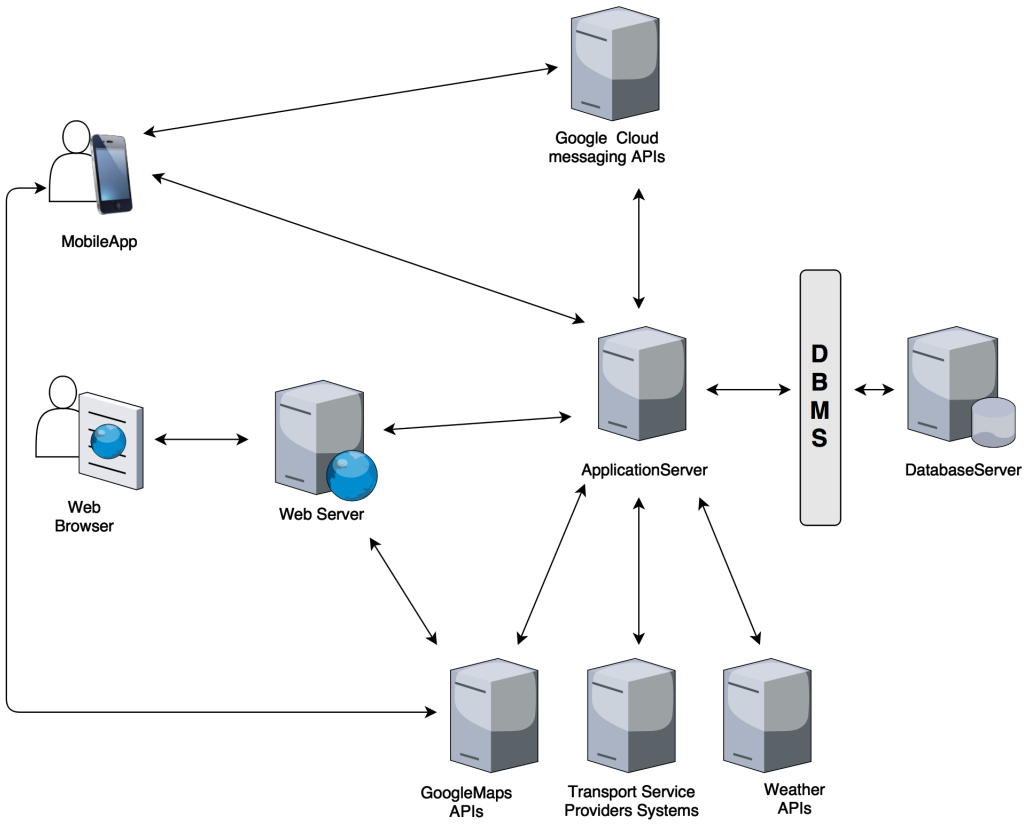


Figure 2.1: General Architecture

2.2 Component view

In this section we'll present and analyze the main components (and their sub-parts) in which our system is divided and we'll explain the relations between them.

2.2.1 Application Server

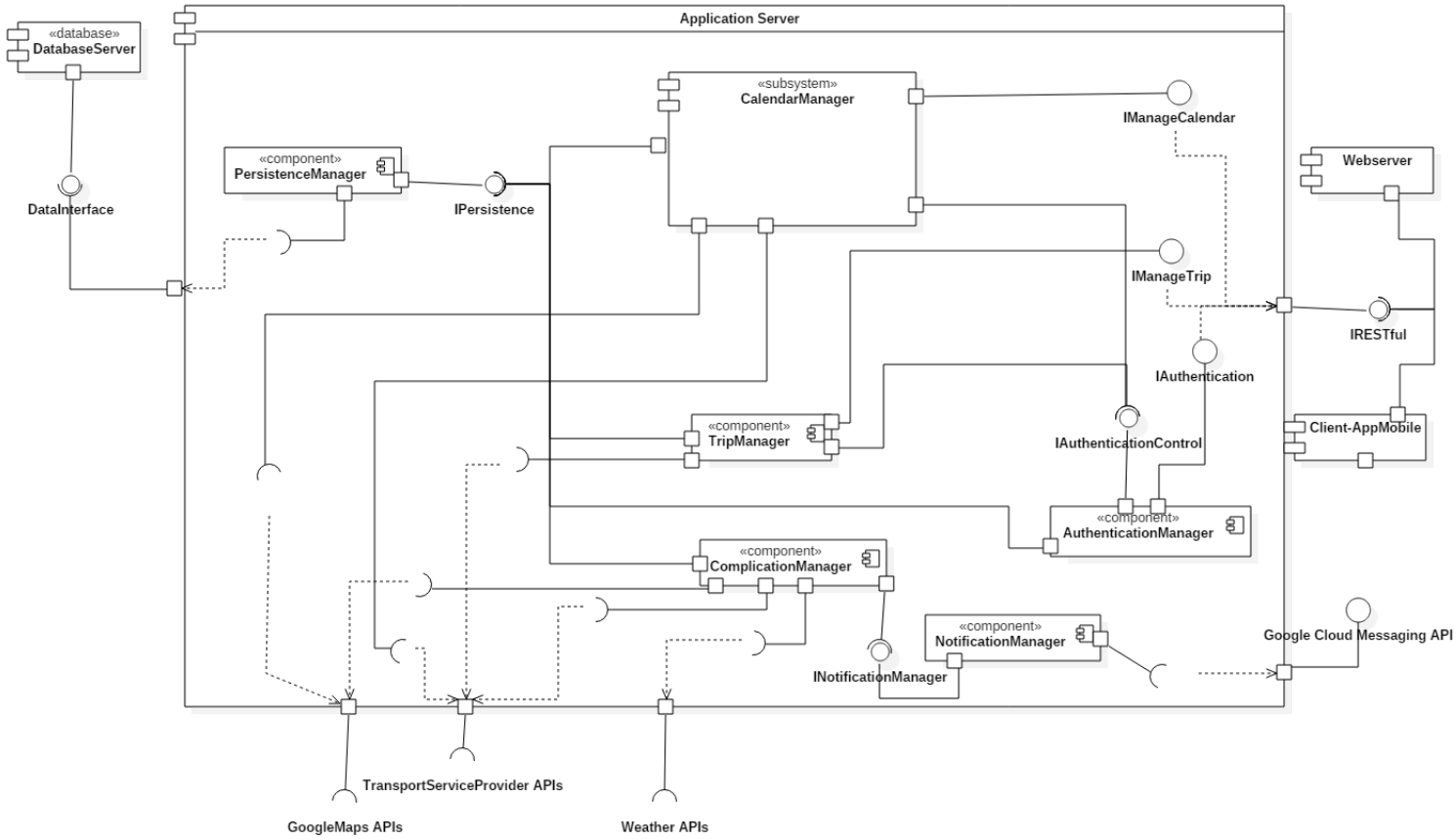


Figure 2.2: The components of the Application Server

The application server must handle the whole business logic of the application and the connection with the data layer. It must expose all the required functionalities to the users. It also has to interact with external systems.

The main components of the Application Server are:

- **AuthenticationManager:** This module will manage the user registration, the user login and it will be involved in any user request to the application server in order to check if the request comes from an authorized device;
- **CalendarManager:** This subsystem will manage all calendar-related functionalities of Travlendar+'s system, its structure will be better explained in the dedicated section below, along with a schema of its internal components;
- **TripManager:** This module will provide the logic needed to arrange the user trips; to do so, this module will interact with external systems of Transport service providers, in order

to allow the user to buy public transportation tickets or to locate the nearest vehicle of a sharing system, when those travel means are involved in the user's travel path;

- **ComplicationManager:** This module will periodically check if the travels that are close in time are still feasible. To do so it will interact with external systems of Transport service providers to gain information about strikes, with open weather APIs to obtain information about the weather and with *Google Maps APIs* to obtain information about the traffic. If this module detects that a travel is no more feasible, it will charge the *NotificationManager* to warn the involved user;
- **NotificationManager:** This module will serve as a gateway to all the notifications to be sent, from others modules, to the user's mobile devices. To do so it will use the *Google Cloud Messaging APIs* in order to ensure a transparent interface with both IOS and Android devices;
- **PersistenceManager:** This module will provide transparent access to the database functionalities from all the others modules. All system database-related functionalities must be developed inside this module. If some users have inserted one or more periodic events, this module will handle their propagation in the time (see also section **??**);

Calendar Manager

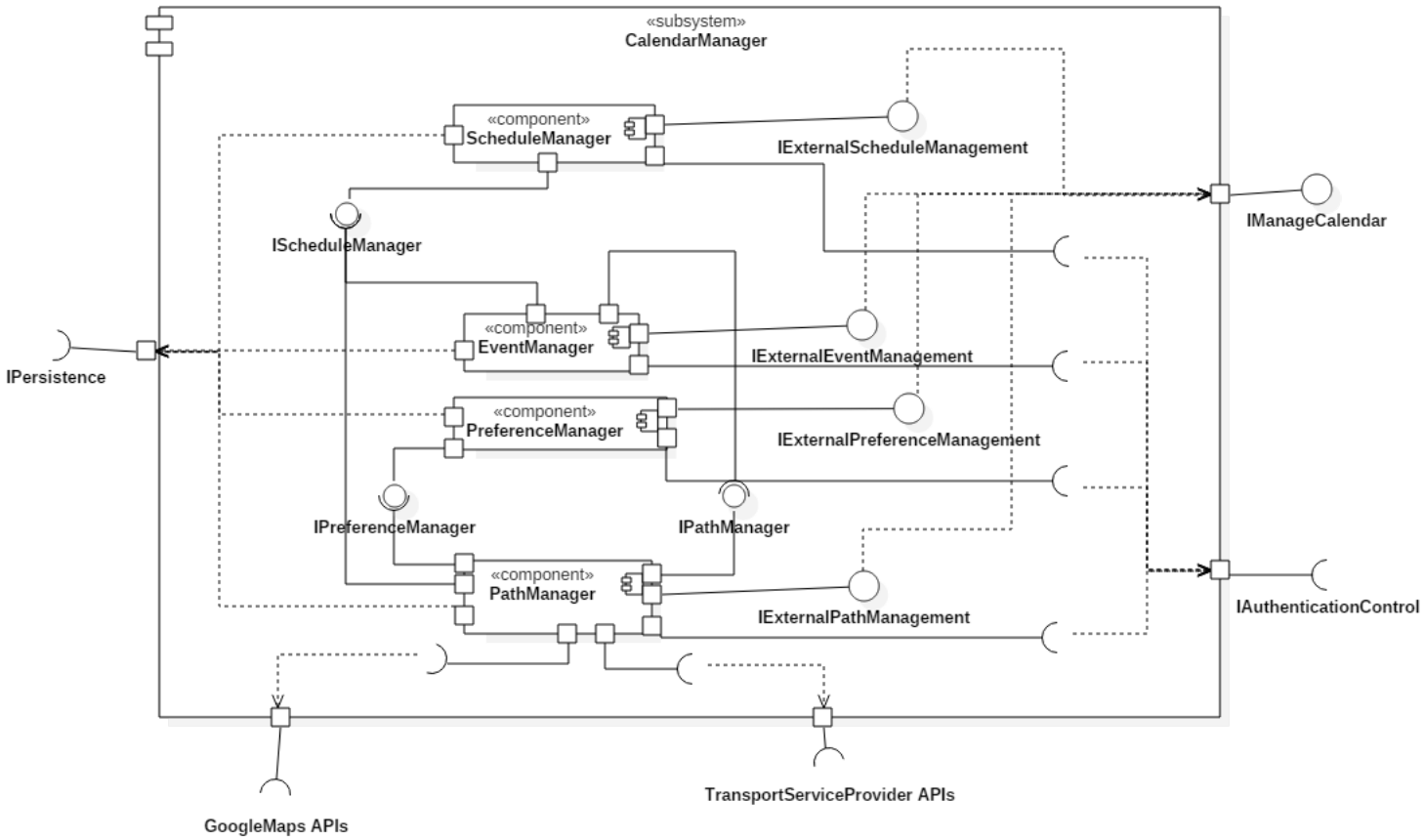


Figure 2.3: The components of the CalendarManager subsystem

This subsystem will implement almost all the core functionalities of our system.
It is composed of four main components:

- **PathManager:** This module will manage the user travels computation, after the insertion or the modification of an event, or also after a specific user requests to change a selected path. It will compute the feasible paths through proprietary internal logic and the involvement of *Google Maps API* and it will require *TSP APIs* of shared vehicles companies in order to include these travels means into the paths. It will also interact with the *PreferenceManager* in order to obtain the information needed to guarantee that the user preferences are respected in every proposed path;

- **EventManager:** This module will manage the insertion, deletion and modification of the events into the user calendar. It will interact with the *ScheduleManager* in order to find out if an event does overlap with other events. When an event is created it will involve the *PathManager* in order to provide feasible paths to reach the inserted events. It will also handle insertion, deletion and modification of the flexible breaks;

- **ScheduleManager:** This module will manage the event's scheduling functionalities: it is able to check if an event overlaps with other events and to guarantee that the flexible breaks are respected. It will also check that the user's travels do not overlap with other events. When an event overlaps with another one, it will put in a separate list of non scheduled events and it will manage the user's requests of rescheduling these events.

- **PreferenceManager:** This module will offer all the functionalities needed to insert, modify and delete the user's event profiles. It will also interact with the module that has to apply the event profiles in order to compute the travel paths (*PathManager*).

2.2.2 Database Server

The *Database Server* must manage the insertion, deletion and modification on data inside his secondary memory. This layer must guarantee ACID distributed transactions.
Since we've already provided an exhaustive description of the model of the data to be memorized into the database (**see Section 2.2.1 of RASD document**) we will not insert here another diagram.
Nonetheless some small additions must be made to that diagram: in the tables it must be added a time-stamp field in order to handle the update of the mobile Application's Local Database with only the actual changes in the database (detailed explanation in section 2.7.4), it must be also memorized an additional table that will store all the IDs of the devices connected to an account and the relative univocal code associated with them (detailed explanation in section 2.7.1).

2.2.3 Web Server

The web server layer connects the users that want to use Travlendar+ through a web browser with the Application Server.
The main functions to be implemented in this layer are basically web interfaces. The only logic in this layer is used to embed a map into the relative web page and to draw the paths the user must travel to reach his events. To do so the web browser will interact with *Google Polylines APIs*. The presentation will be handled by the *DynamicWebPages* component that will contains all the web pages and the logic used to generate them dynamically. The interaction with the Application server and with the maps APIs will be handled by the *WebController* module.

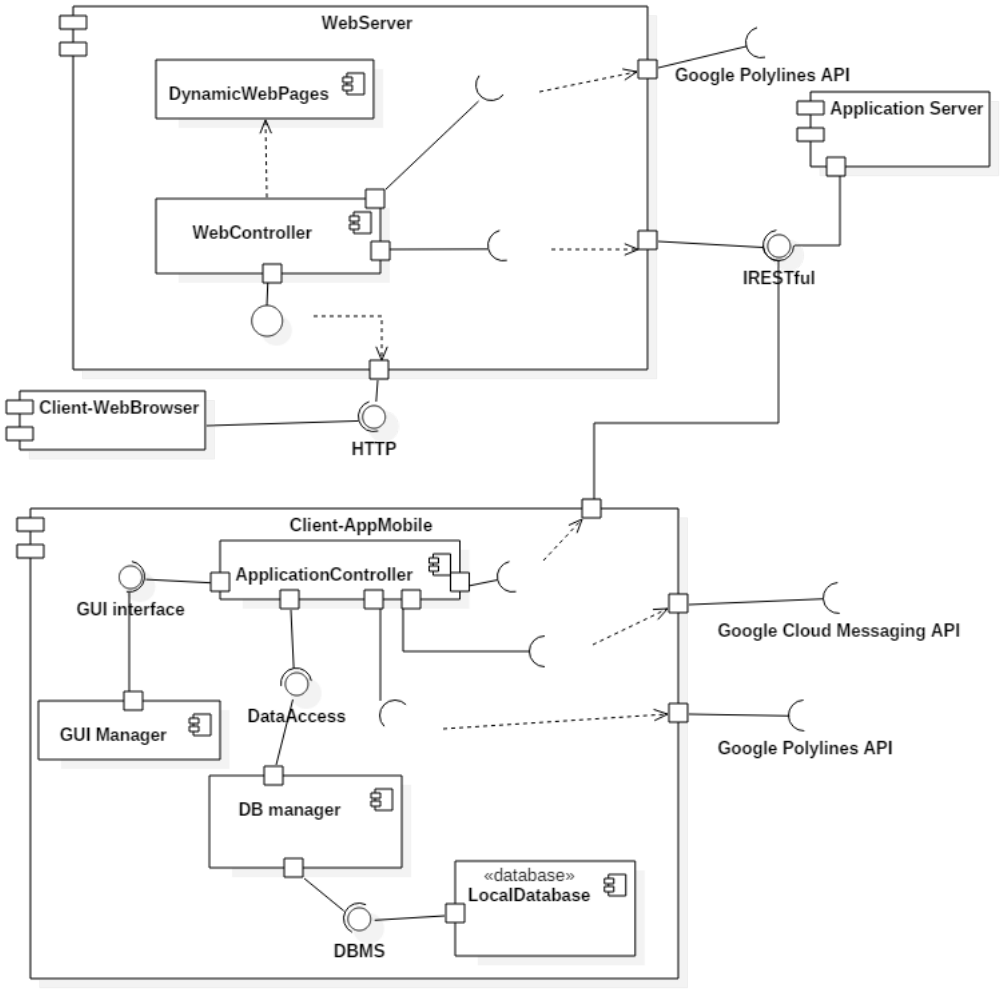


Figure 2.4: The components of the web server, of the client app and of the client browser

2.2.4 App Mobile

The layer that connects the users that want to use Travlendar+ through their mobile devices with the Application Server. To do so it uses three main modules:

- *GUIManager*: it handles all the presentation functionalities of the app;
- *DBManager*: it handles all the users data, such as events and some information about travels duration, in order to enable the user to use some of the app functionalities even when he does not have access to an Internet connection (see also the E-R diagram in image **??**);
- *ApplicationController*: it handles the interaction with the server, using the inputs provided by the users through the *GUIManager* and also requesting data manipulation operations in the local app database with the information received from the server (in order to perform an update of the *Local Database*).

The application controller will also handle the notifications received and it will interact with *Google Polylines APIs* for mobile devices in order to to draw the paths the user must travel to reach his events and provide GPS path following functionalities.

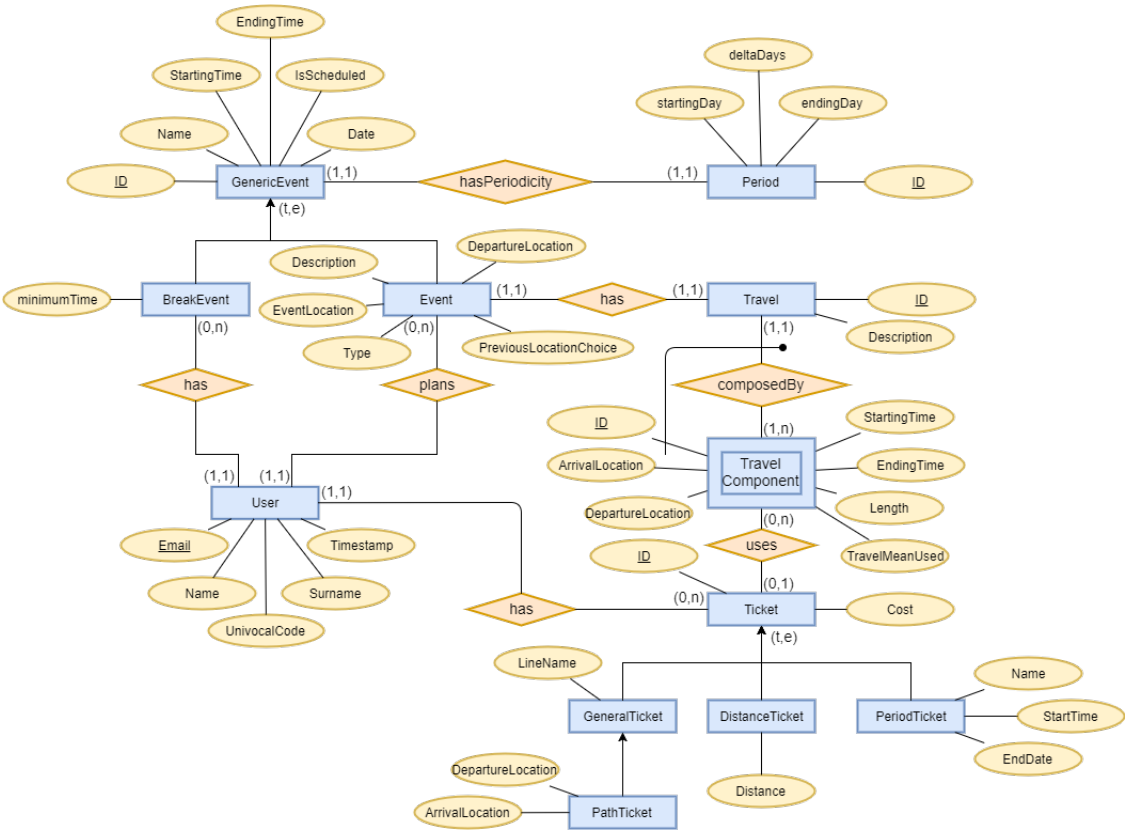


Figure 2.5: E-R diagram representing the mobile app Local Database data structure.

2.3 Deployment view

The main purpose of this section is to show how the various components of the system are actually deployed on the hardware infrastructure.
During the design process of the physical architecture are been taken into account both functional and non functional requirements. In particular we've focused on the following non functional requirements: reliability, availability and security. In order to satisfy those requirements we have identified the following 5-tiered architecture:

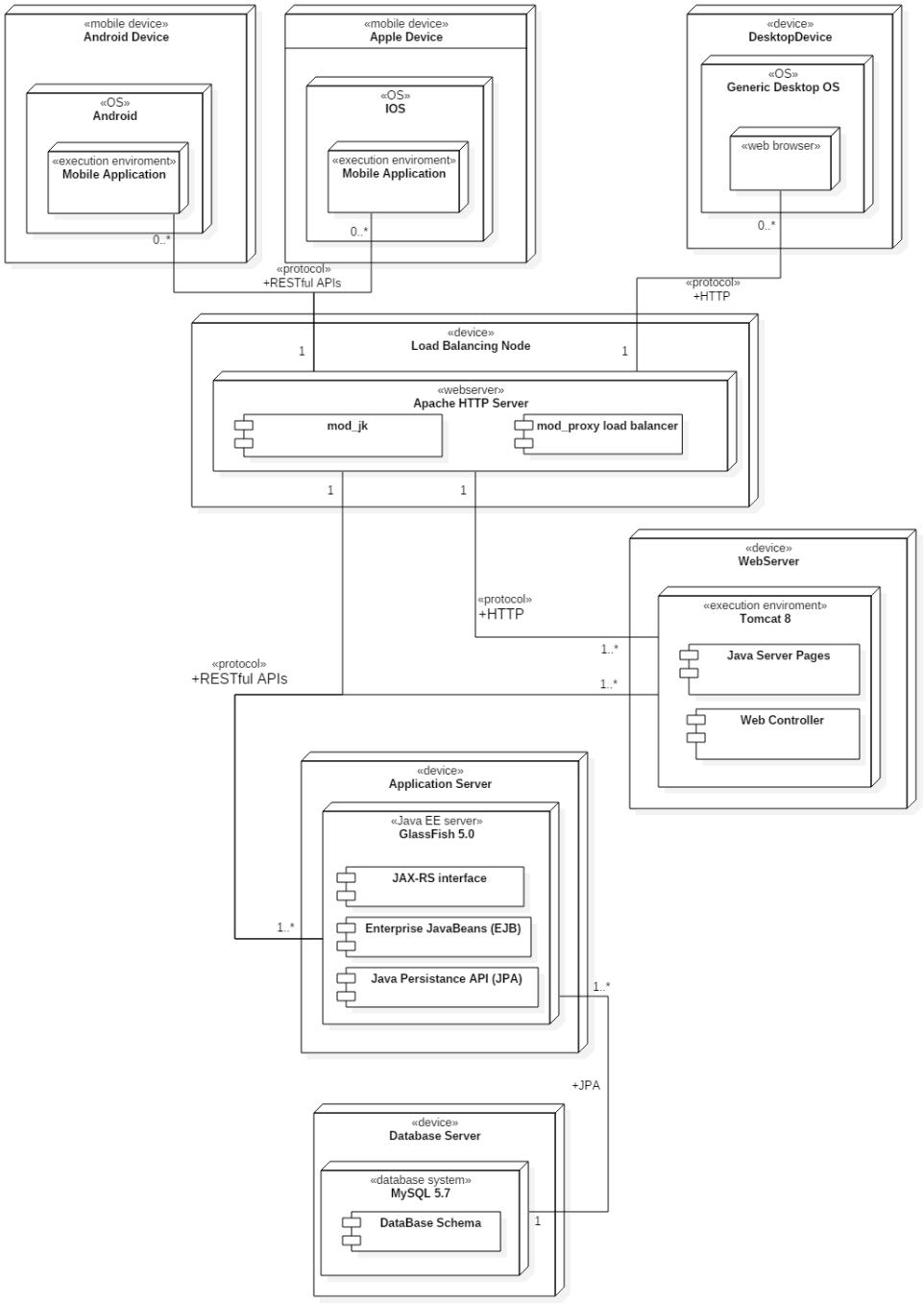


Figure 2.6: Deployment diagram

2.3.1 Recommended Implementation Choices

Here we provide a set of recommended implementation choices in order to actually develop the Travlendar+ system.

- The **Database Server** may be implemented with MySQL 5.7 as Relational DBMS;
- The **Application Server** may be implemented using Java Enterprise Edition (JEE) 8; this choice could allow the developers to focus on the logic to be provided while being supported by reliable APIs and tools, in order to reduce the complexity of the development phase. Using JEE 8 the developers could also guarantee the main non-functional requirements, which are stated in RASD document. The specific JEE 8 choices can be:
  a) GlassFish 5.0 as Application Server implementation;
  b) JPA (Java Persistence API) in order to interact with the database Database Server, in particular Entity Beans can be used to map the data;
  c) EJB (Enterprise Java Beans) in order to implement the business logic, in fact the components described in section 2.2.1 can be implemented as multiple Stateless Session Beans;
  d) JAX-RS in order to implement the interface used through the web with both mobile apps and the web server.
- The **WebServer** may be implemented using Tomcat 8 as HTTP web server implementation and JSP (JavaServer Pages) may be used to provide dynamically generated web pages;
- The **Load Balancing Node** may be implemented using an Apache HTTP Server;
- The **Android App** may be implemented using Java and XML programming languages;
- The **IOS App** may be implemented using Swift programming language.

2.4 Runtime view

The main purpose of this section is to describe the dynamic behavior of Travlendar+ system when the main features are utilized. In particular we will highlight the interactions between every sub component of our system.

2.4.1 Login

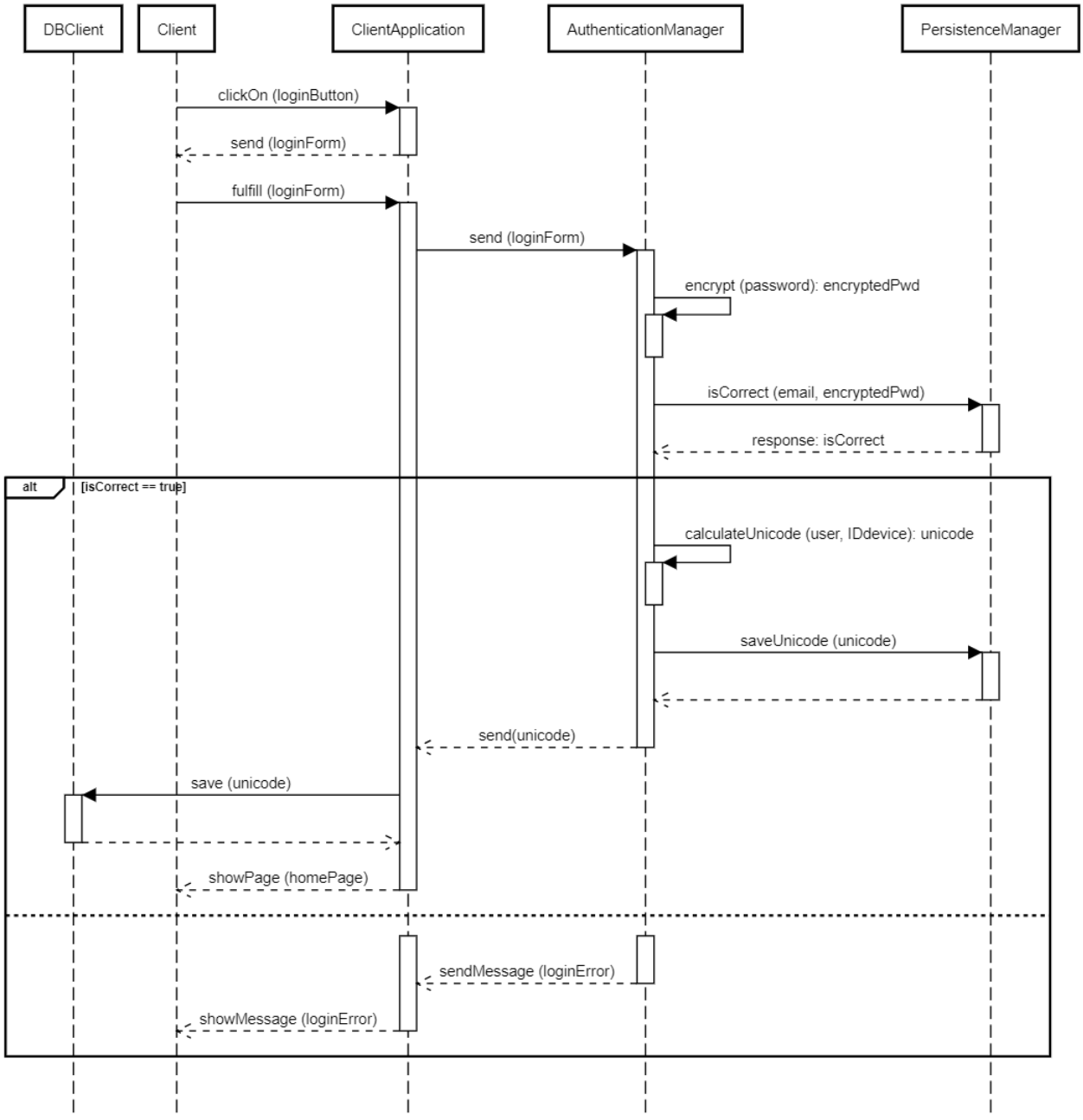


Figure 2.7: Sequence Diagram - Login

The login operation is performed only the first time that the application is used in a new device or after a logout. The *system* calculates a univocal code that is returned to the client and stored

in the *local DB*. In this way the client can identify himself during future requests. The encryption function, specified in this diagram, will be specified below in the document (see section 2.7.2).

2.4.2 Authentication functions for each operation

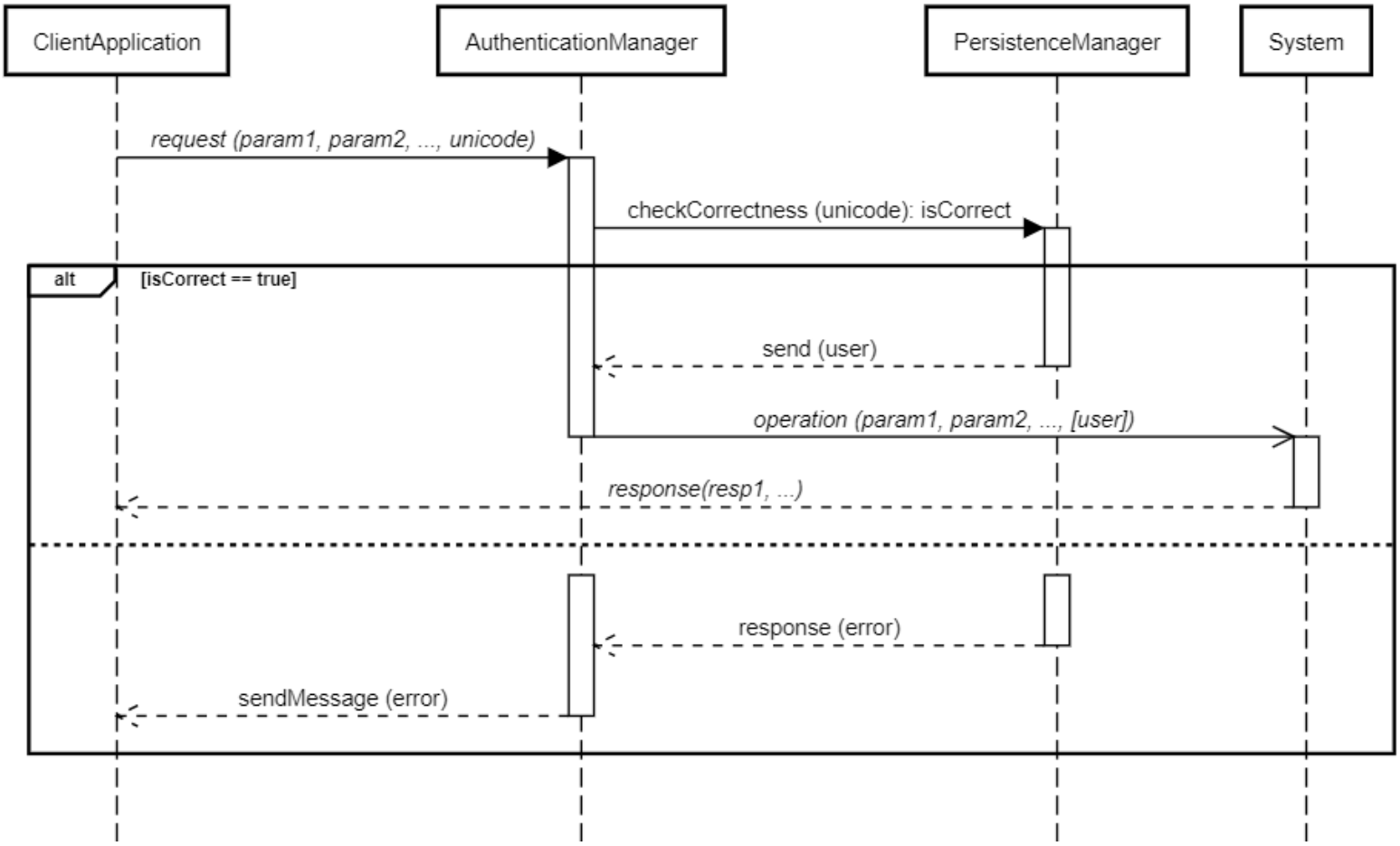


Figure 2.8: Sequence Diagram - Authentication

This is a sample of how a generic request is taken into account by the system: the *Client* sends the univocal code with the other parameters of a request and the *Authentication Manager* performs the check. The *Authentication Manager* forwards the request toward his destination only if the univocal code is correctly recognized, otherwise it rejects the request.

2.4.3 Create event

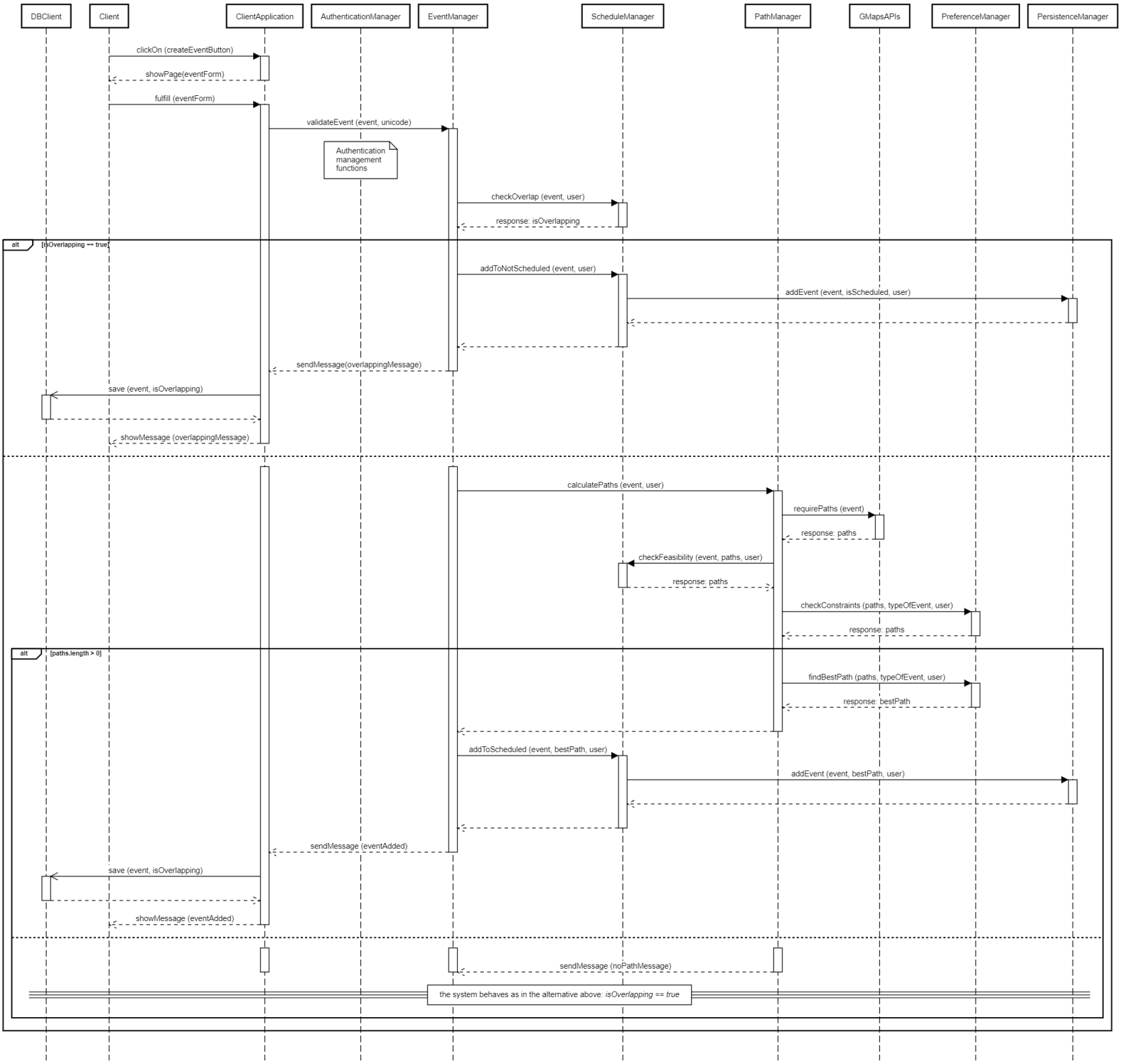


Figure 2.9: Sequence Diagram - Create event

2.4.4 Arrange trip

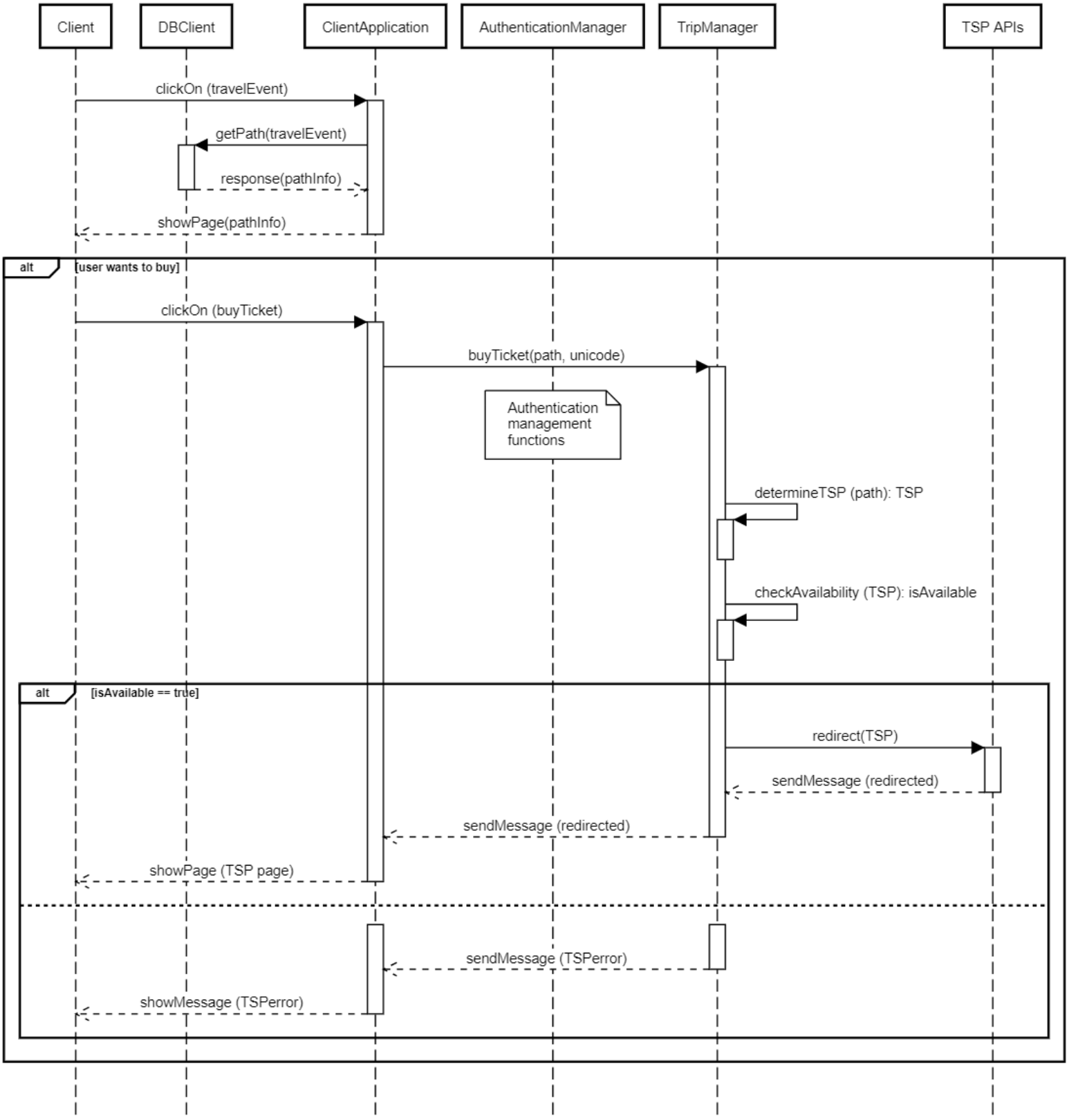


Figure 2.10: Sequence Diagram - Arrange trip

2.4.5 Obtain feasible paths

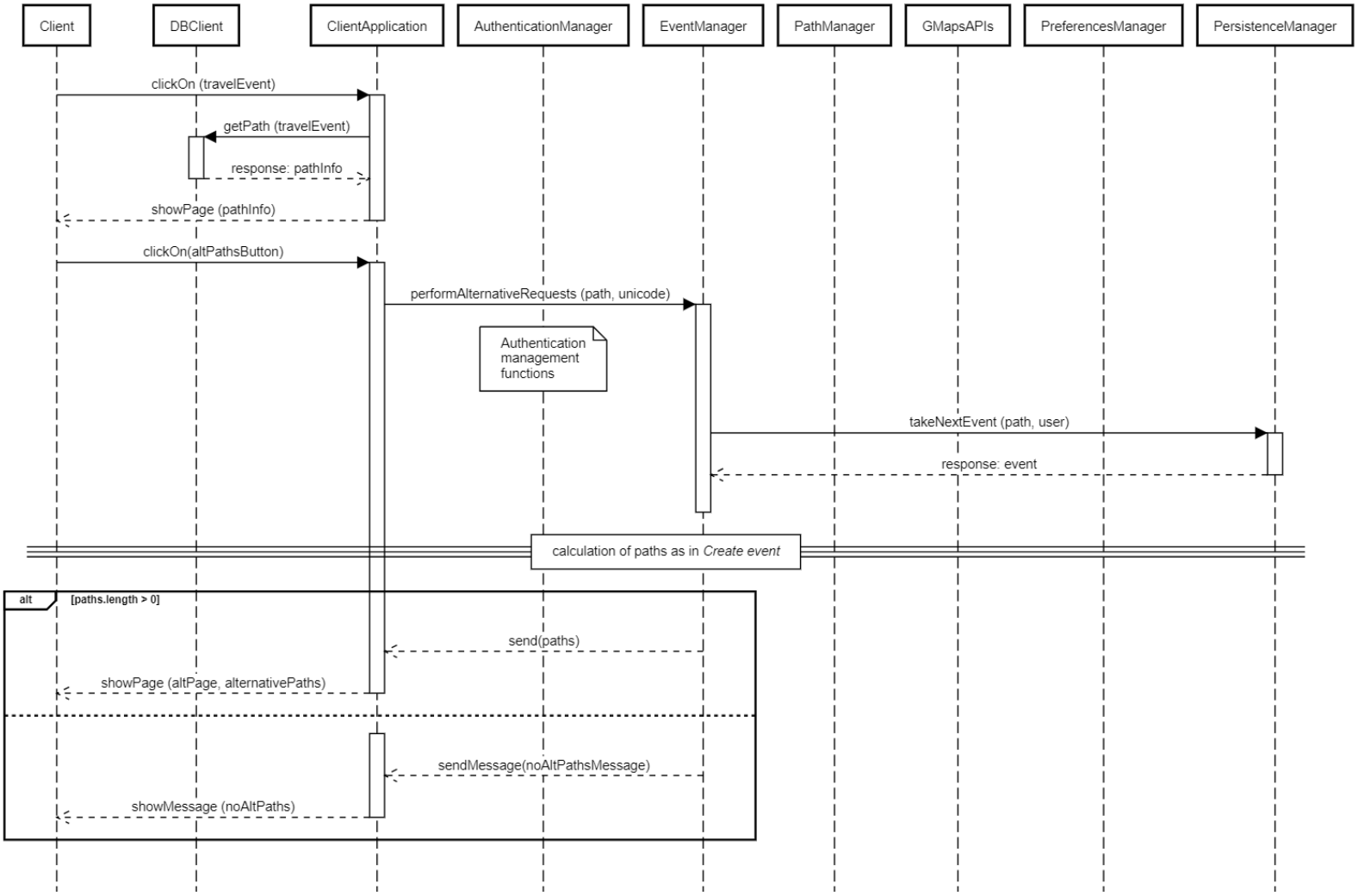


Figure 2.11: Sequence Diagram - Obtain feasible paths

Only the selected path is stored into *local and server DBs*, so if a user wants to see alternative feasible paths, a new request of feasible paths is to be performed. The *system* obtains the event related to the path and then calculates the feasible alternatives with its internal logic and also through the invocations of *GMaps* functions. The system also checks feasibility and constraints of the obtained paths. Obviously not only the best path is returned to the *client*.

2.4.6 Choose between overlapping events

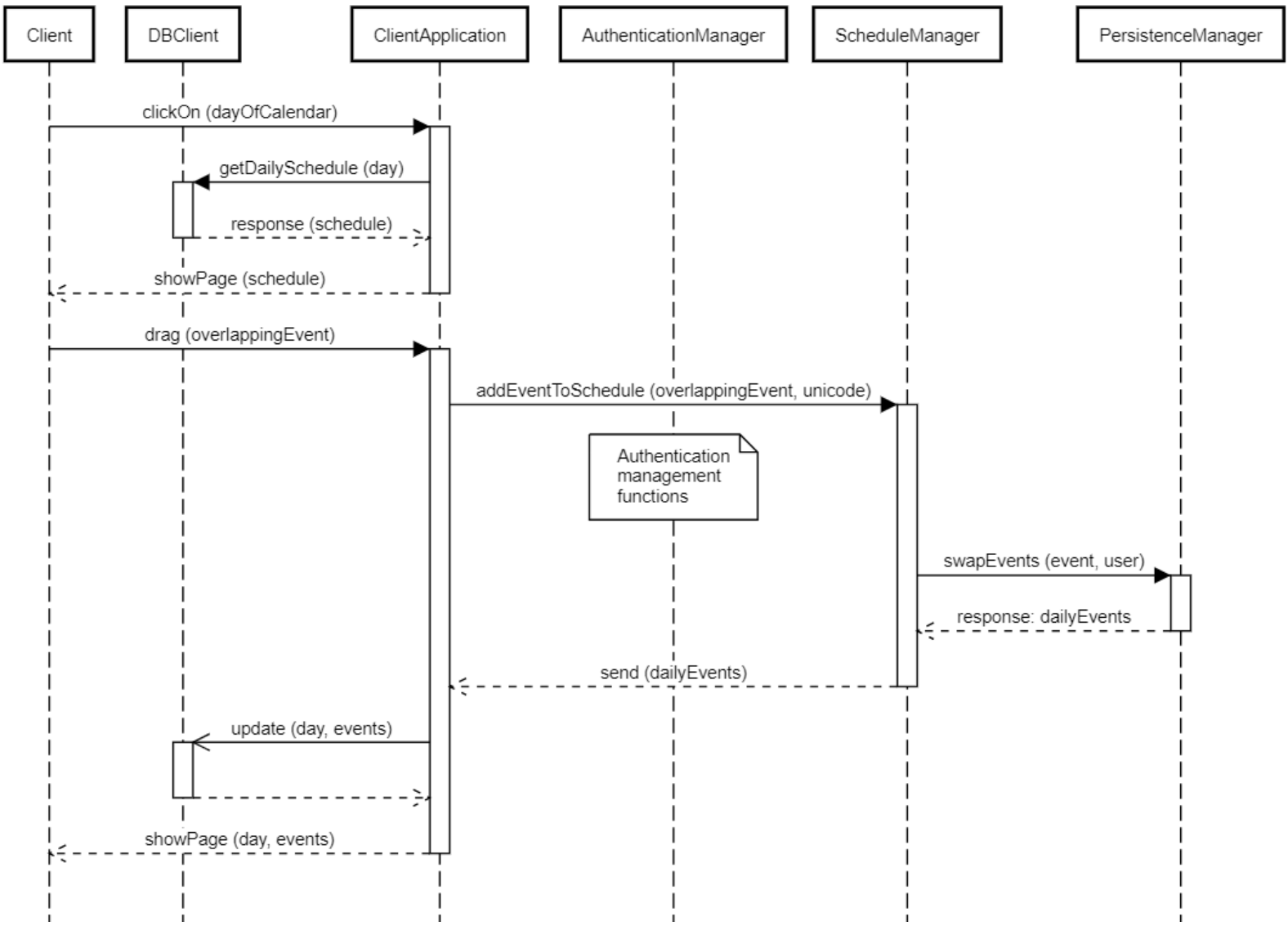


Figure 2.12: Sequence Diagram - Choose between overlapping events

2.4.7 Strike announcement

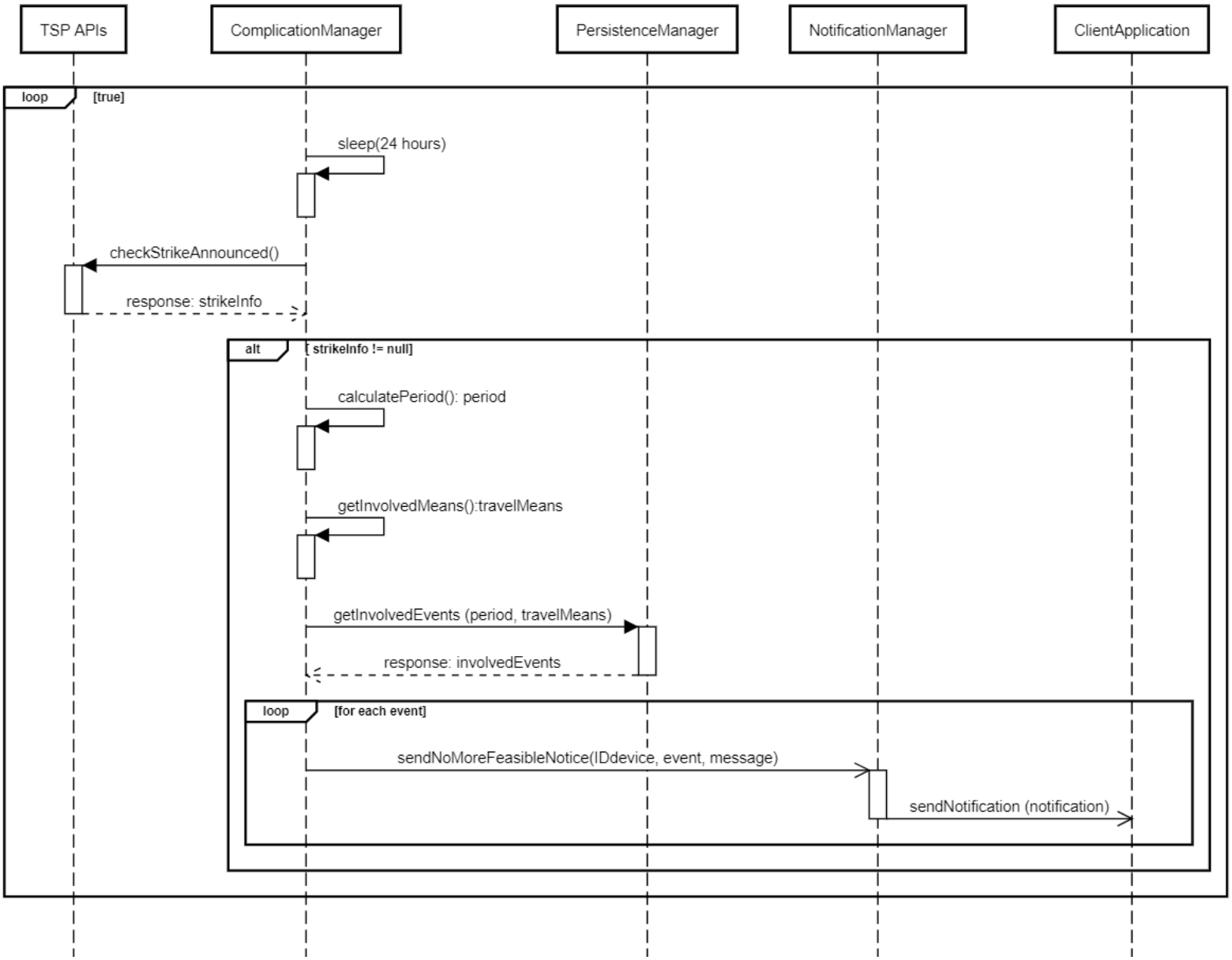


Figure 2.13: Sequence Diagram - Strike announcement

The *Complication Manager* every day performs a list of operations, one of them is to check if a strike for the following days is announced. In case of strike a notification is sent for each event stored in the *database* that is involved: several notifications can be sent to the same user. If the user opens the notification, *obtain feasible paths* function, related to the involved event, is to be invoked.

2.5 Component interfaces

We classify our interfaces in two types: the (internal) interfaces, used by the components of the application server in order to interact with each other, and the external interfaces, offered with as *RESTful web services*, as specified in the previous sections. For the sake of accurateness and

clarity we'll specify for each *RESTful* method the relative *HTTP* reference request method (*GET, POST, HEAD, OPTIONS, PUT, DELETE, TRACE, CONNECT* and *PATCH*).

2.5.1 IManageCalendar

IManageCalendar is the core of the system: it provides all the useful operations to assist the user in the organization of his commitments. This interface is subdivided into four external interfaces:

IExternalScheduleManagenent

This external interface offers, to the user's devices, access to all the schedule-related functionalities. It exposes the following methods:

- *getUpdatedSchedule(unicode, timestampLocal)*: it updates the user schedule into local database (GET method);
- *getDailySchedule(day, unicode)*: it allows to obtain the schedule of a specified day (GET method).

IExternalEventManagement

This external interface offers, to the user's devices, access to all the event-related functionalities. It exposes the following methods:

- *getEventsUpdated(unicode, timestampLocal)*: it provide all the updated the user's events that are to be saved into the local database. This function returns also the best path for each event (GET method);
- *getEvents(unicode)*: It provide all the user events to be saved into the local database. This function returns also the best path for each event (GET method);
- *getEventInformation(event, unicode)*: it allows to obtain the info related to a specific event (GET method);
- *addEvent(event, unicode)*: it begins the task of adding the event specified by the user. Internal methods will be used in order to add correctly the event into the database, to insert if into the schedule and to provide a feasible path (POST method);
- *modifyEvent(event, unicode)*: it allows the user to modify a previously inserted event (PATCH method);
- *deleteEvent(event, unicode)*: it allows the user to delete a previously inserted event (DELETE method);
- *performAlternativeRequests (path, unicode)*: it is used when the user wants to obtain alternative feasible paths of a selected path (GET method);
- *addBreakEvent(breakEvent, unicode)*: it begins the task of adding the break event specified by the user. Internal methods will be used in order to add correctly the event into the database and to insert if into the schedule (POST method);
- *modifyBreakEvent(breakEvent, unicode)*: it allows the user to modify a previously inserted break event (PATCH method).

IExternalPreferenceManagement

This external interface offers, to the user's devices, access to all the preference-related functionalities. It exposes the following methods:

- *getPreferenceProfiles(unicode)*: it allows to obtain the profiles, containing user preferences, defined by a user. These profiles are named "type of events" (GET method);
- *getPreferenceProfile(preferenceId, unicode)*: it allows to obtain info about a specific preference profile, given his ID (GET method);
- *addPreference(typeOfEvent, unicode)*: it allows the user to add a new type of event with a set of preferences and constraints (POST method);
- *modifyPreference(typeOfEvent, unicode)*: it allows the user to modify a previously inserted type of event (PATCH method);
- *deletePreference(typeOfEvent, unicode)*: it allows the user to delete a previously inserted type of event (DELETE method);
- *addPreferredLocation(location, name, unicode)*: it allows the user to add a new Preferred Location (POST method);
- *modifyPreferredLocation(location, name, unicode)*: it allows the user to modify a Preferred Location (PATCH method);
- *deletePreferredLocation(name, unicode)*: it allows the user to delete a Preferred Location (DELETE method);
- *getAllPreferredLocations(unicode)*: it allows the user to obtain the data relative to his preferred locations (GET method);
- *getPreferredLocation(locationName, unicode)*: it allows the user to obtain the data relative to a specific preferred location given his name (GET method).

IExternalPathManagement

This external interface offers, to the user's devices, access to all the travel paths-related functionalities. It exposes the following methods:

- *getPathInfo(event, unicode)*: it is used to obtain info related to a specific path that is part of the schedule (GET method);
- *getMap(unicode)*: it is used to obtain info that allow to draw the user travels on the map (GET method);
- *swapSchedule(overlappingEvent, unicode)*: it is used when the user want to force into the schedule an overlapping event (PATCH method).

2.5.2 IAuthentication

This external interface offers, to the user's devices, authentication, registration and profile-management functionalities. It exposes the following methods:

- *register(registrationForm, IdDevice)*: it is used when a user registers himself into the system. To do so the encrypted registration form (email, name, surname, password and captcha) is sent to the Application Server , which returns an univocal code related to the device (POST method);

- *submitLogin(credentials, IdDevice)*: it is used when a user has to log into the system, it returns an univocal code related to the device (POST method);
- *editProfile(registrationForm)*: it is used when a user wants to modify his profile, to do so it sends the same info contained in the registration form (PATCH method);
- *requestPublicKey(IDdevice)*: it is used when an user is about to log into the system and so it must obtain a public key to encrypt his password (GET method);
- *deleteProfile(mail, password, IdDevice)*: it allows the user to remove his profile from Travlender+ (DELETE method);
- *askNewCredentials(mail)*: it allows the user to request a new password, that will be sent to his email (PATCH method).

2.5.3 IManageTrip

This external interface offers, to the user's devices, the arrange trips functionalities of Travlendar+. It exposes the following methods:

- *addTicket(ticket, unicode)*: it adds a ticket into the user personal tickets (POST method);
- *deleteTicket(ticket, unicode)*: it deletes a ticket from the user personal tickets (DELETE method);
- *modifyTicket(ticket, unicode)*: it modifies a ticket contained in the user personal tickets list (PATCH method);
- *buyTicket(path, unicode)*: it allows the user to obtain the specific URLs where he can buy the tickets needed for a travel (GET method);
- *getTickets(unicode)*: it allows the user to obtain the data relative to his saved tickets (GET method);
- *selectTicket(ticket, unicode, path)*: it allows the user to connect a ticket to a specific travel path (PATCH method);
- *deselectTicket(ticket, unicode, path)*: it allows the user to undo the ticket association to a specific travel path (PATCH method);
- *getNearSharingVehicles(location, unicode)*: it allows the user to retrieve the location of the near (to him) sharing vehicles selected in his travel (GET method).

2.5.4 IScheduleManager

This interface is used internally in the calendar manager subsystem to allow event and path managers to interact with *ScheduleManager*, the methods used in these interactions are the following:

- *checkOverlap(event, user)*: it is used during the creation of an event and indicates if it's already present an event in the period specified for the new one;
- *addToNotScheduled(event, user)*: it adds the event into the database as not scheduled event;
- *addToScheduled(event, bestPath, user)*: it adds the event into the database as scheduled event;

- *checkFeasibility(event, paths, user)*: it allows to discard paths that are non feasible. This function is used during the calculation of the paths related to an event;
- *isScheduleFeasible(user)*: it indicates if the schedule proposed to the user is free from overlappings;
- *isBreakEventScheduled(breakEvent, user)*: it indicates if a specified break event is included into the proposed schedule.

2.5.5 IPreferenceManager

This interface is used internally in the calendar manager subsystem to allow the *PathManager* to interact with *PreferenceManager*, the methods used in these interactions are the following:

- *checkConstraints(paths, typeOfEvent, user*): it allows to discard paths that do not respect constraints defined by the user. This function is used during the calculation of the paths related to an event;
- *findBestPath(paths, typeOfEvent, user)*: it indicates, among different paths, which is the best one according to a parameter of the type of event defined by the user ;
- *isVehicleAllowed (event, vehicle)*: it indicates if a vehicle can be used for a given event, according to the type of that event and any constraint defined by the user on hour of travel or max length with a travel mean.

2.5.6 IPathManager

This interface is used internally in the calendar manager subsystem to allow the *EventManager* to interact with *PathManager*. The methods used in these interactions are the following:

- *calculatePaths(event, user)*: it manages the interaction with *Google Maps APIs* in order to obtain all the paths related to a certain event;
- *addPathPathsWithSharingVehicle(vehicle, event)*: it is called by *calculatePaths* and allows to consider, according to user preferences, the sharing vehicles in the calculation of paths (NB. *Google Maps APIs* don't provide information about sharing vehicles).

2.5.7 IPersistence

IPersistence provides all the data-related functionalities to the other internal components of Travlendar+ system. This interface exposes a set of methods used by the others server components to perform read and write operations. In particular update, delete, insert and select queries methods are exposed for all the data types used, created and consumed.

2.5.8 INotificationManager

INotificationManager provide methods to be used by the Application server components in order to send notifications to the user's mobile applications. At this stage we've identified these three methods:

- *sendNotification(IdDevice, message)*: it allows to send a text notification to a specific mobile device, the notification will be displayed on the device;
- *sendNoMoreFeasibleNotice(IdDevice, event, message)*: it allows the system to notify a specific device that an event travel is no more feasible;

- *sendUpdateNotice(IdDevice)*: it is called to notify a device that his local database is no more up to date, and to order the device to update itself.

2.5.9 IAuthenticationControl

AuthenticationManager provides an internal (in the Application Server) interface that provides one method:

- *filter(containerRequestContext)*: it allows all the application server components to verify the identity and the authorizations of a request, this method practically checks the univocal code contained in every request payload.

2.6 Selected architectural styles and patterns

The following architectural styles and patterns have been used in order to ensure a well-formed and efficient architecture:

2.6.1 Layered Architecture

This architectural style allows to organize the system through abstraction levels and doing so it separates presentation, application logic and data management functionalities.
Furthermore, since each layer can be separately instantiated on a different machine, this architecture guarantees great flexibility in terms of hardware configurations and simplifies the actual development of the system, cause all components can be implemented and tested separately.

2.6.2 Client/Server

Our application is strongly based on the client server paradigm and it's used at various levels:

- the mobile application (client) makes requests to the Application Server (server) that replies to them;
- the web browsers (clients) communicates with the Web Server (server), which also acts as a client with respect to the Application Server.
- the Application Server acts as client when it interrogates the database Server or when it interacts with external APIs.

2.6.3 Model-View-Controller

The proposed Travlendar+ system follows this pattern. This allows to separate the application into three communicating and interconnected abstract parts which fulfills different goals. In particular our system implements the so-called 'Apple MVC version', which always keeps the Controller between the model and the view interactions in order to guarantee security and access control.

2.6.4 Publish/Subscribe

This architectural style is primarily used in the notification system. When a user mobile device logs into the system subscribes as listener and will receive notice of all possible problems involving the Travlendar+ user experience. This paradigm provides us great flexibility with respect to future expansions.

2.6.5 Five Tier Architecture

This Architectural style allows us to deploy the various physical components of our system on different devices, in order to separate responsibilities and also to introduce redundancy in order to improve the availability and reliability.

2.7 Other design decisions

2.7.1 Authentication

When the users log into the system for the first time from a device, the Application server will assign to that device an univocal code, so for further requests from that device (web browser or mobile application) the users can avoid to login again. Every new request will include into the request payload that code, enabling the Application server to validate the request and to recognize the specific user that sent them. The univocal code will be generated by an internal algorithm in the AuthenticationManager (explained in section 3.2). The AuthenticationManager will check every time the consistency of the univocal code and it will also check if it comes from the same device it is associated to. If a log in is performed from an already registered device but it comes from a different Travlendar+ account, the previous univocal code associated to that device will be deleted, and the previous user will have to log in again with his previous account.

2.7.2 Encryption

The encryption is used only to send password from the client to the server. For the other operations, in fact, the client exploits the univocal code described below in order to authenticate itself. This code is used in combination with the ID of the device that sends the request, so encryption for the univocal code is not necessary.
To encrypt the password is suitable a *public key algorithm* because it allows to send the string in a secure and easy way on internet and the size of data to encrypt is small.
A public key algorithm uses a pair of keys: a *public key* distributed to the clients and a *private key* known only by the server. The client encrypts the password with the public key and sends it to the server; the server is the only one able to decrypt the message thanks to the private key.
The encryption in the system is implemented according to the *RSA algorithm*.

2.7.3 Notifications

In order to send notifications to the mobile devices our system will memorize the Identifiers of the user devices, and when a notification is to be sent the Application Server will interact with the interfaces offered by *GCM* (*Google Cloud Messaging*) in order to actually forward them to the user devices. We choose to use *GCM* services cause they handle all aspects of queuing of messages and delivering to client applications that runs on different target devices, and also because it is completely free.

2.7.4 Local database update

When the application performs an operation that requires a connection, both online and local database are updated. The user however can perform an operation on a different mobile device or with a browser, in these cases the local database of the application must be updated.
When the online database is modified, the system sends a notification to all the mobile devices related to the user whose local database isn't updated yet. The notification mechanism works as specified in section 2.7.3. Only the changes made after the last writing operation on local database are sent from server to local DB, in order to update it.

2.7.5 Periodic events

Once a day, an internal module is run to check that every periodical event inserted by a user is covered for a year, e.g. if an event like "Friday lunch with parents" with a weekly periodicity is created on 2018/01/05, on 2018/01/12 the module will create a new event on 2019/01/12.

Chapter 3

ALGORITHM DESIGN

In this section we will provide an overall description of the main algorithms required in order to enable Travlendar+ system to work as intended and to satisfy both functional and non functional requirements. The following algorithms are explained with different methods, this allow us to focus on different details that must be described in different algorithms.

3.1 Path calculation

In order to compute optimal paths that include also sharing vehicles, not included into the set of possible travel means offered by *Google Map APIs*, the system will have to implement some additional logic. We will handle the sharing vehicles option as an additional possible public travel mean, for example an user could want to take a bike in sharing instead of taking the subway.
The algorithm that implements this functionality will follow the behavior explained in the pseudo-code below:

```
calculatePaths (event, user){
    paths = requirePaths(event)   /*require all possible feasible paths relatives to the event (invocation of
                                    Google maps APIs)*/
    if ( the user preferences allow BIKE as Vehicle in sharing ) {
        addPathsWithSharingVehicle (BIKE)  in paths
    }
    if ( the user preferences allow CAR as Vehicle in sharing ) {
        addPathsWithSharingVehicle (CAR)  in paths
    }
    forall ( path in paths ) {
        if (path is not feasible in the allotted time  )  \* allotted time = time between the event the path is
                                                            related to and the previous one) */
            remove path
        else if ( path does not respect the constraints related to the type of event selected )
            remove path
    }
    return paths \\find the best possible path according to the user preferences
}
```

The addPathsWithSharingVehicle method is explained below:

```
addPathsWithSharingVehicle( VEHICLE ){
    for all ( path in paths )
        if ( path.getTravelMeans == VEHICLE and VEHICLE is a sharing vehicle available on that entire path )
            paths.add( new Path where VEHICLE becomes VEHICLE _IN_SHARING)
        else if ( path.areUsedMultipleVehicles and VEHICLE not in path.usedVehicles )
            for all ( vehicle used in path )
                if ( in vehicle's sub path is available VEHICLE as a sharing vehicle){
                    newSubPath = requirePaths( subPath where vehicle becomes VEHICLE _IN_SHARING)
                    paths.add ( newPath where subPath becomes newSubPath)
                    /*the two methods above basically add a new path with one vehicle changed into
                      VEHICLE _IN_SHARING, the changed sub path is recomputed*/
                }
}
```

3.2 Univocal code

In the *registration process* email and password (sent with the encryption algorithm specified in section 2.7.2) are written in database.
A log in operation is required in order to use the functionalities of the system.

When the user performs a login, email and password are sent to the system together with the ID device, obtained by the mobile app through the exploit of a *GCM's function*. In the DB several ID of devices can be associated to the same user, there is a field that marks the ID as mobile (related to phones and tablets) or browser (related to PCs). Browser IDs are deleted periodically in order to improve the access-control mechanism (after an ID deletion the users will have to re-insert their credentials).

The system behaves as specified in the flowchart below (figure **??**) according to one of the following situations:

- *first access*: device is not present in the database;
- *login with a different device*: device is in the database, but it is associated to a different user;
- *login with the same device* (not the first access): a new univocal code is calculated and it will be updated on the database.

The univocal code is a randomly generated string concatenated to the email (unique), so that the univocal code generated will be different from all previous ones. The random string is calculated starting from the *Current Unix Timestamp*.

The univocal code is sent to device (a parameter that will be stored locally in the case of a mobile application, a cookie in the case of a browser) and will be used for every interaction with the system: in this way the identification of the client happens. For each request the system takes the univocal code received and the ID of the device and checks in the database if they are related to each other, before performing any required operation.

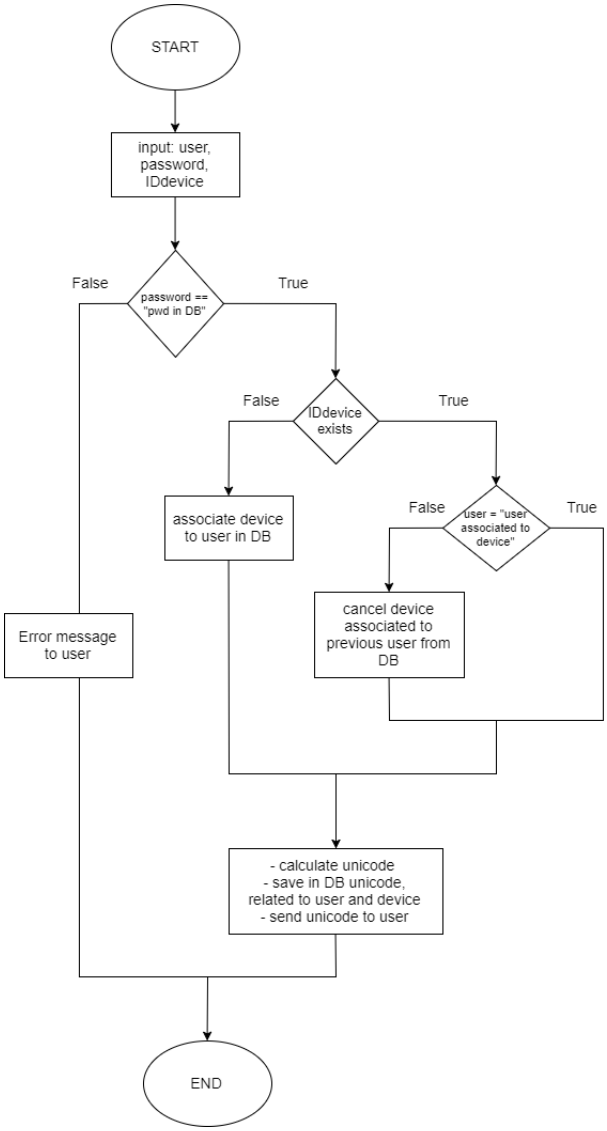


Figure 3.1: Flowchart - Univocal code

3.3 Swap events in the schedule

The purpose of this algorithm is to check and manage the schedule when an overlapping event is forced into it, maintaining thus a feasible list of scheduled events.

To explain it we will use the following terms:

- *ForcedEvent*: the event that is going to be put into the schedule;
- *ScheduledEvent*: an event that will be removed from the schedule as a consequence of the ForcedEvent insertion;
- *ScheduleList*: a list containing all the scheduled events for a specific day;
- *OverlappingList*: a list containing all the overlapping events for a specific day.

The algorithm works in this way:

1. Checks if there are *ScheduledEvents* with:

   startingTime(ScheduledEvent) <endingTime(ForcedEvent)
   and
   startingTime(ForcedEvent) <endingTime(ScheduledEvent)

   If the condition results True for one or more scheduled events, it removes them from the schedule and puts them in the *OverlappingList*;

2. Computes the possible travels before and after the *ForcedEvent*, depending on the information inserted about event location and departure location, and eventually depending also on previous and following existing events into the schedule;

3. If the travel used to reach the *ForcedEvent* requires a time that is greater than the available time, the previous *ScheduledEvent* is removed from the *ScheduleList* and added to the *OverlappingList*. Repeat 2;

4. Else if the travel used to reach the event after the *ForcedEvent* in the schedule requires a time that is greater than the available time, the *following ScheduledEvent* is removed from the *ScheduleList* and added to the *OverlappingList*. Repeat 2;

5. Else if it is not possible to find a combination of paths (before and after the *ForcedEvent*) such that the minimum amount of free time for an involved *break event* can be ensured, the break event is removed from the *ScheduleList* and added to the *OverlappingList*. Repeat 2;

6. Else it is the case in which at least one feasible path combination is found: the best one, according to user preferences, can be chosen and assigned to related events;

7. *ForcedEvent* has been successfully inserted in the schedule. Database is updated with all the modifications made on users events.

3.4 Flexible breaks

The purpose of this algorithm is to check if a flexible break inserted is respected by the events present in the schedule.

To explain it we will use the following terms:

- *BreakList*: a list containing all the events that overlap, even partially, with the flexible break time slot;

The algorithm works in this way:

1. Puts in BreakList all the scheduled events that take place during the allotted break time slot, even if partially;

2. Checks if at least one of the spare time slots between an event of BreakList and the following one (including travels) is larger enough to include the break event;

SPECIAL CASE: if only one event is present in BreakList, the algorithm must check that the remaining spare time in the slot is larger enough to allocate the break event.

3.5 Periodical events

The purpose of this algorithm is to manage the creation of periodical events.

The algorithm works in this way: when a periodical event is inserted, a number of events is generated in order to cover a span of one year, e.g. if an event like "Friday lunch with parents" with a weekly periodicity is created, 52 different events will be created. To better understand the reasoning behind this, take a look at section 2.7.

Chapter 4

USER INTERFACE DESIGN

In this section we provide an overview on how our system will look like.
A minimal part of the mockups had already been inserted in the RASD document, here we will illustrate them with an higher level of detail.
We've included an unique *UX diagram* and an unique *BCE diagram* because the web application and the phone application share the same functionalities, therefore it would have been redundant and not that useful to repeat the same concepts twice.

4.1 Mockups

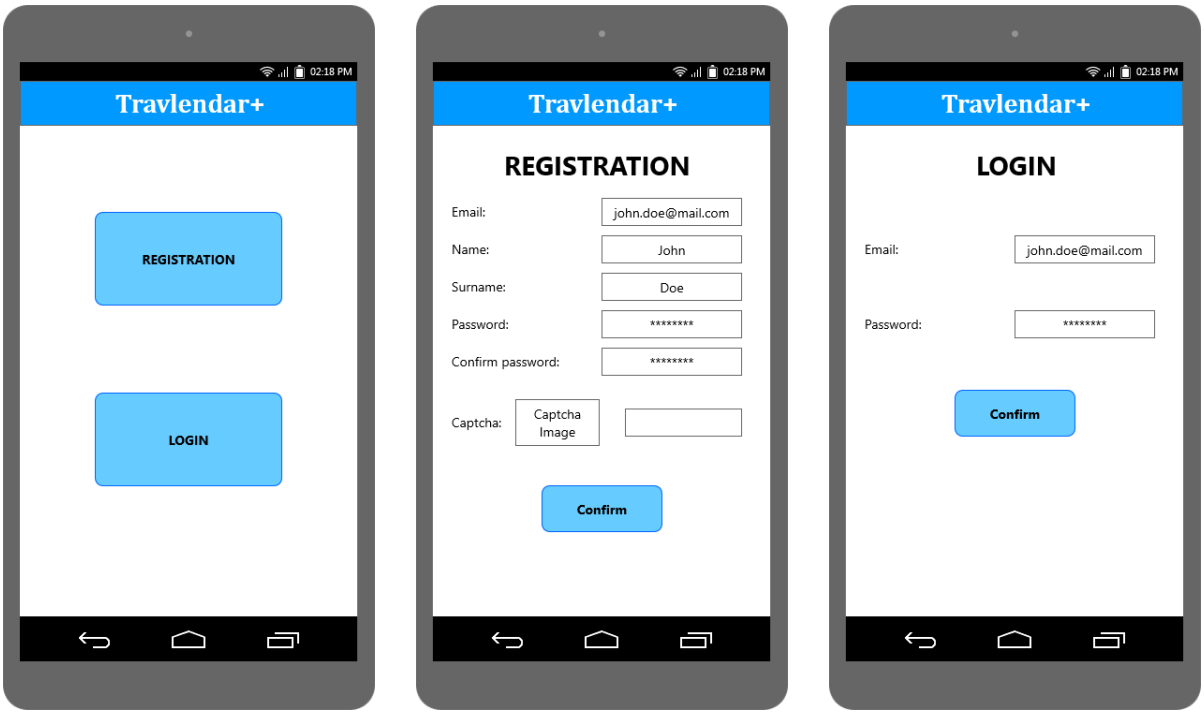


Figure 4.1: The user can register and log into the system.

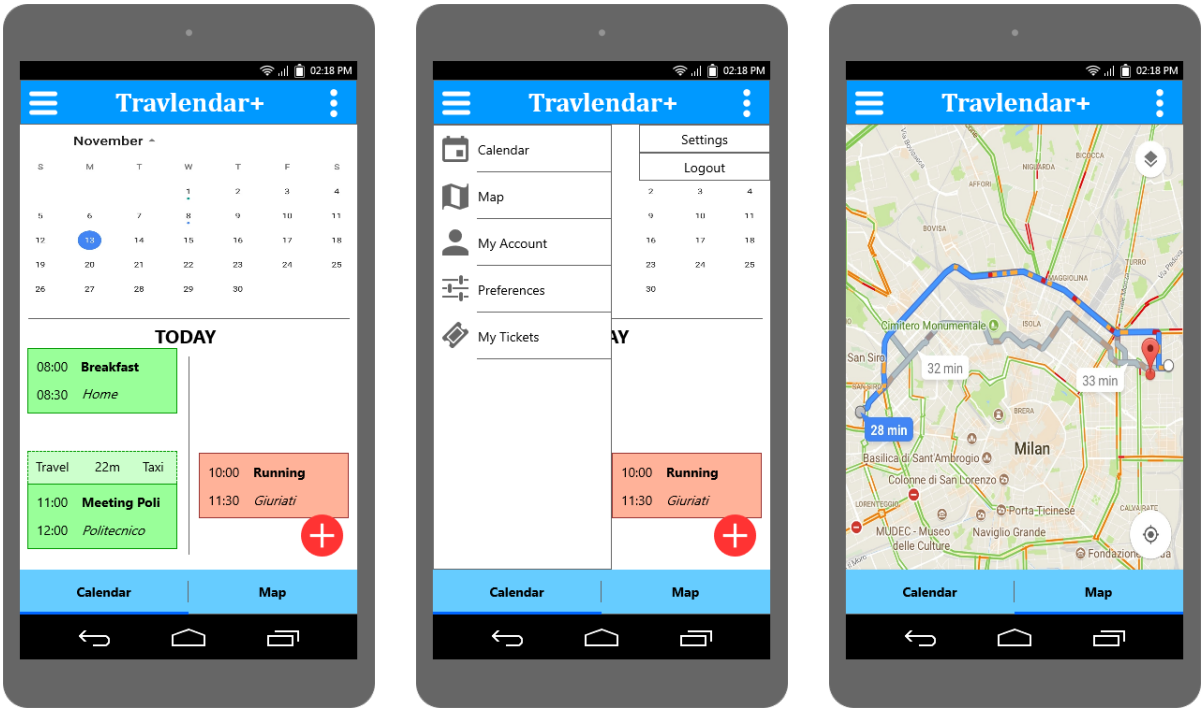


Figure 4.2: The main screens of the application.

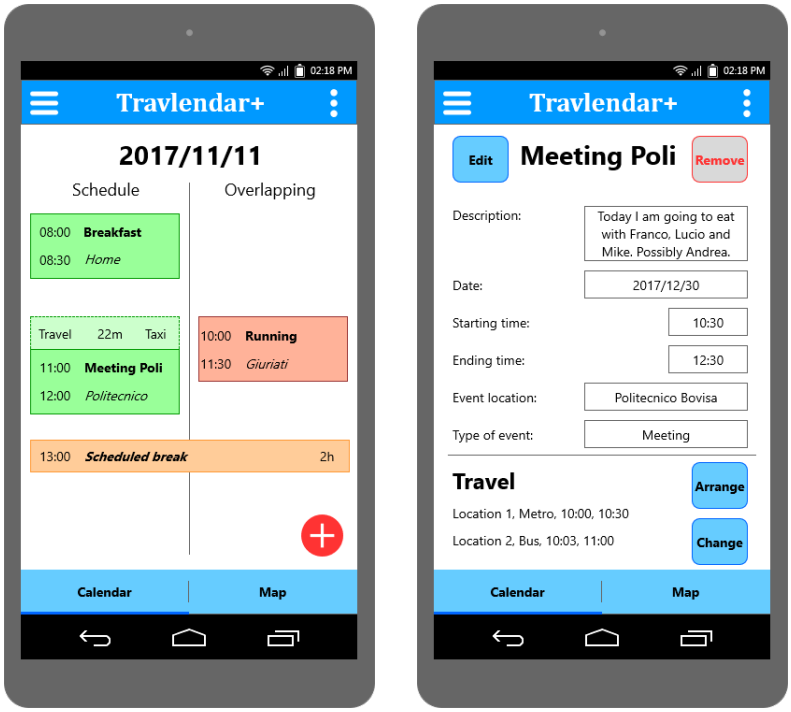


Figure 4.3: The user can see his schedule and view its events.

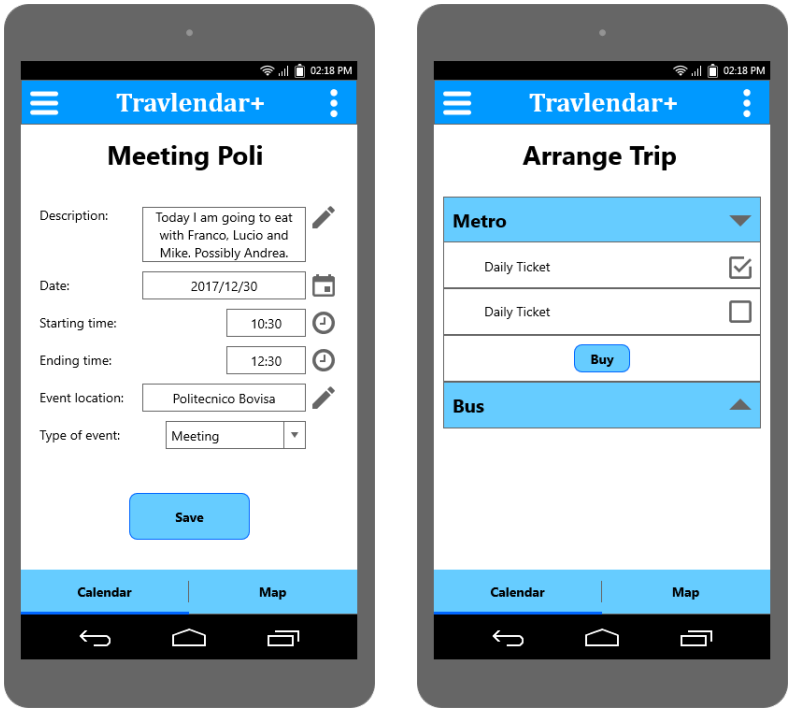


Figure 4.4: The user can edit and arrange its trip.

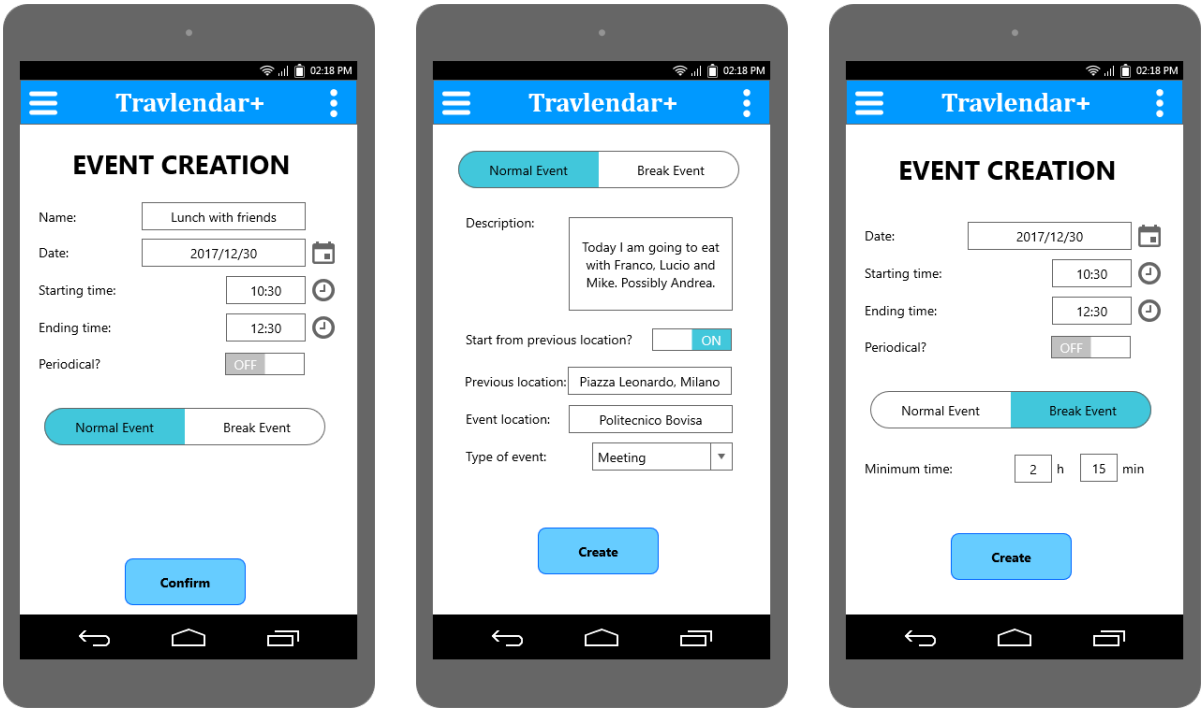


Figure 4.5: The user can create an event, specifying if it's a break event or a normal one.

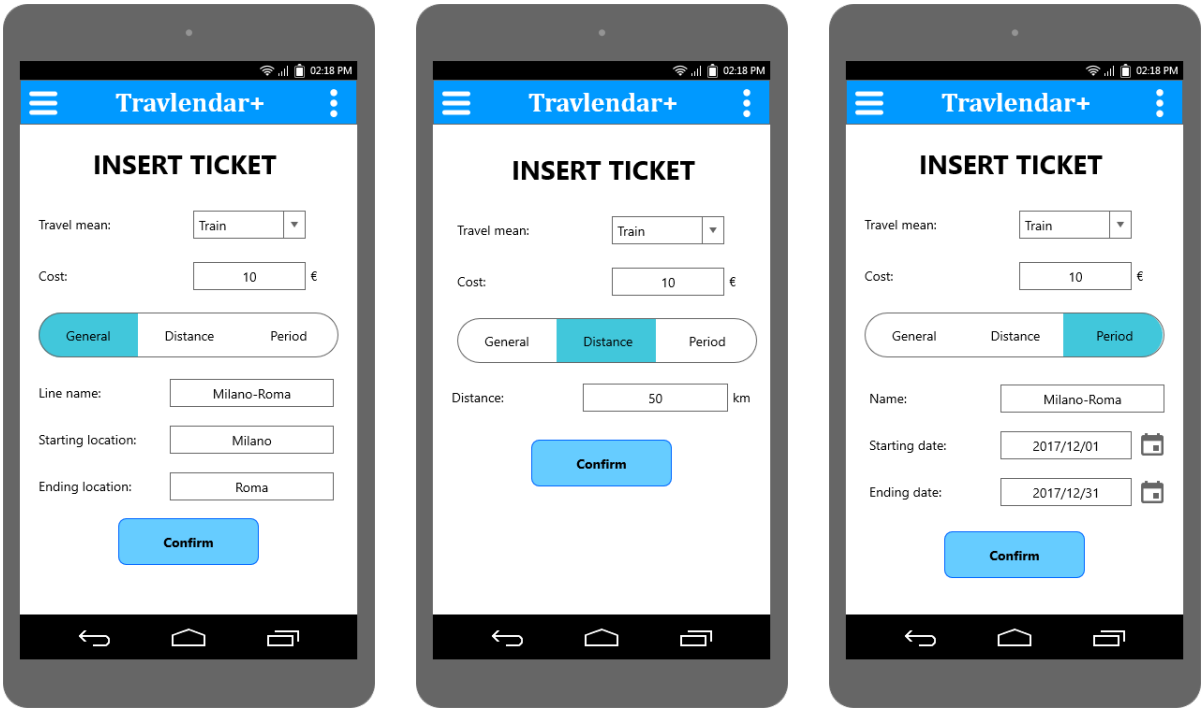


Figure 4.6: The user can handle his tickets.

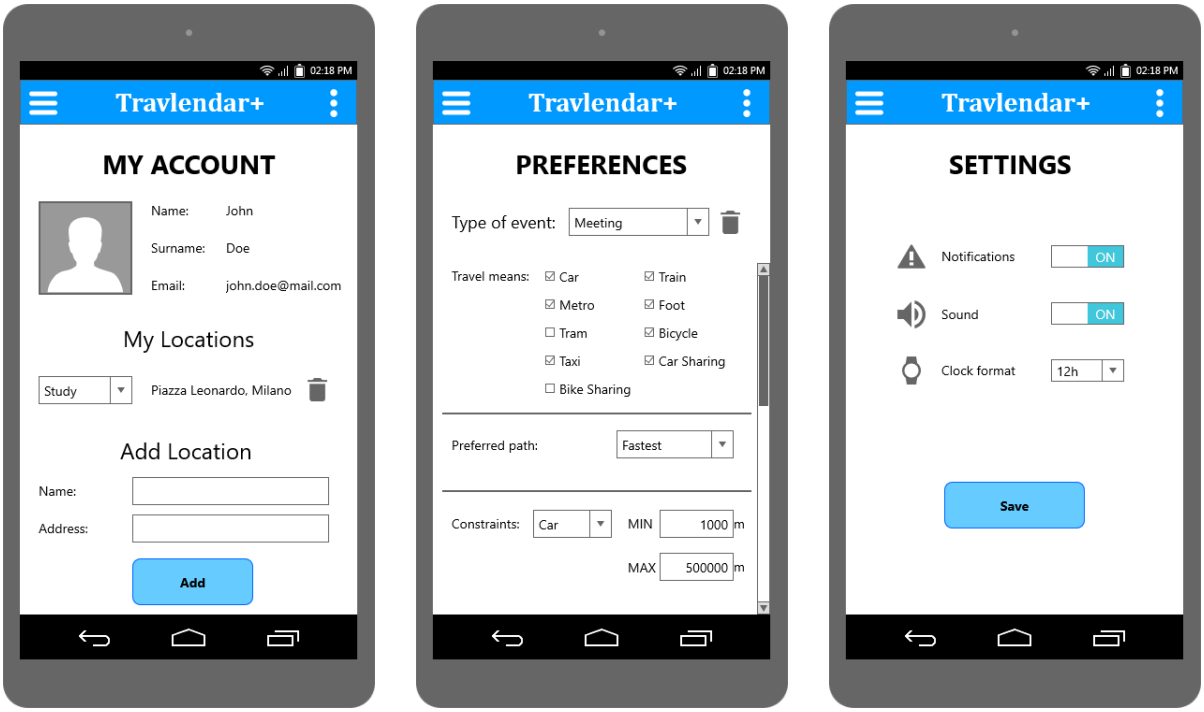


Figure 4.7: The user can modify his preferences and the system settings.

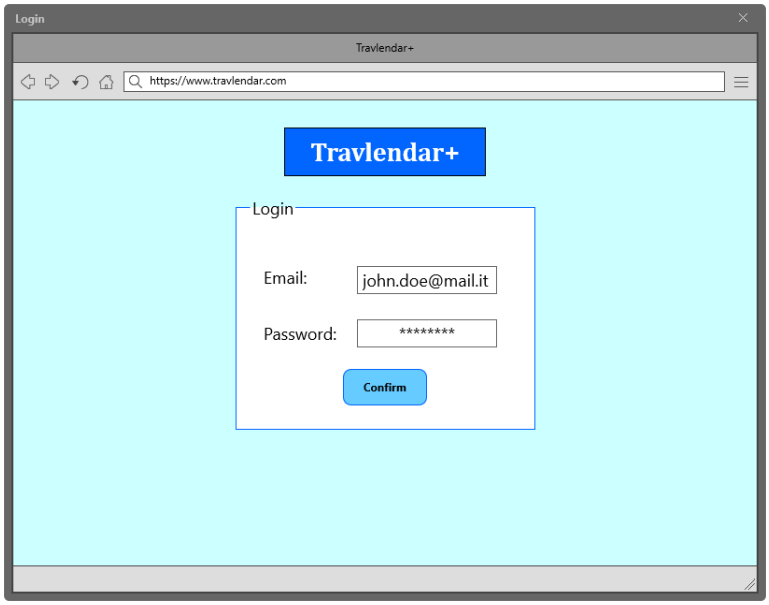


Figure 4.8: Web browser login view.

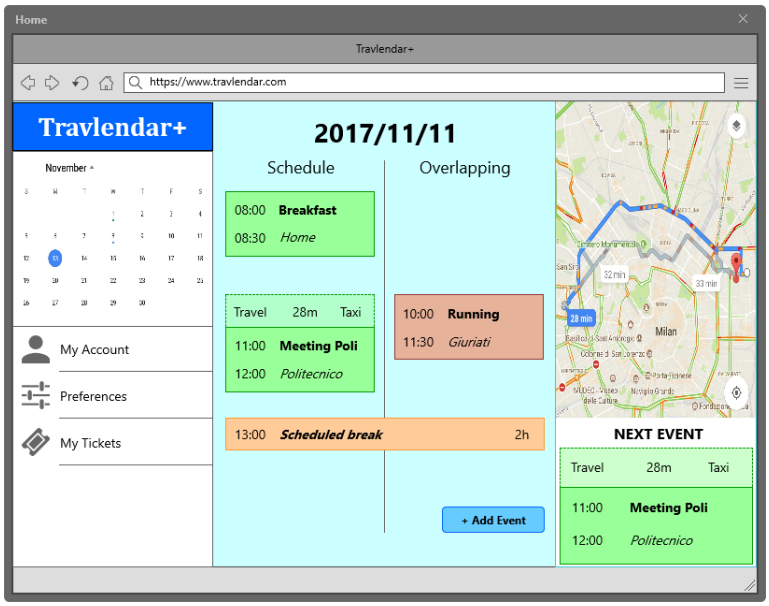


Figure 4.9: Web browser home view.

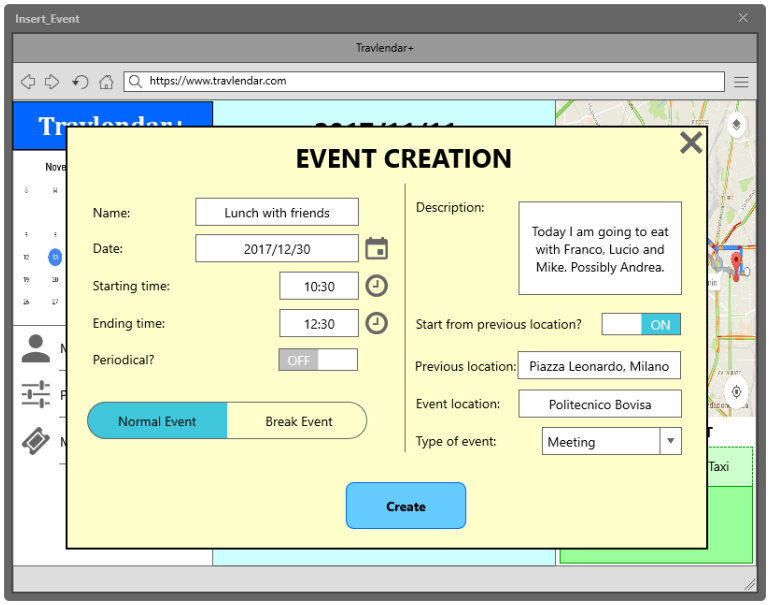
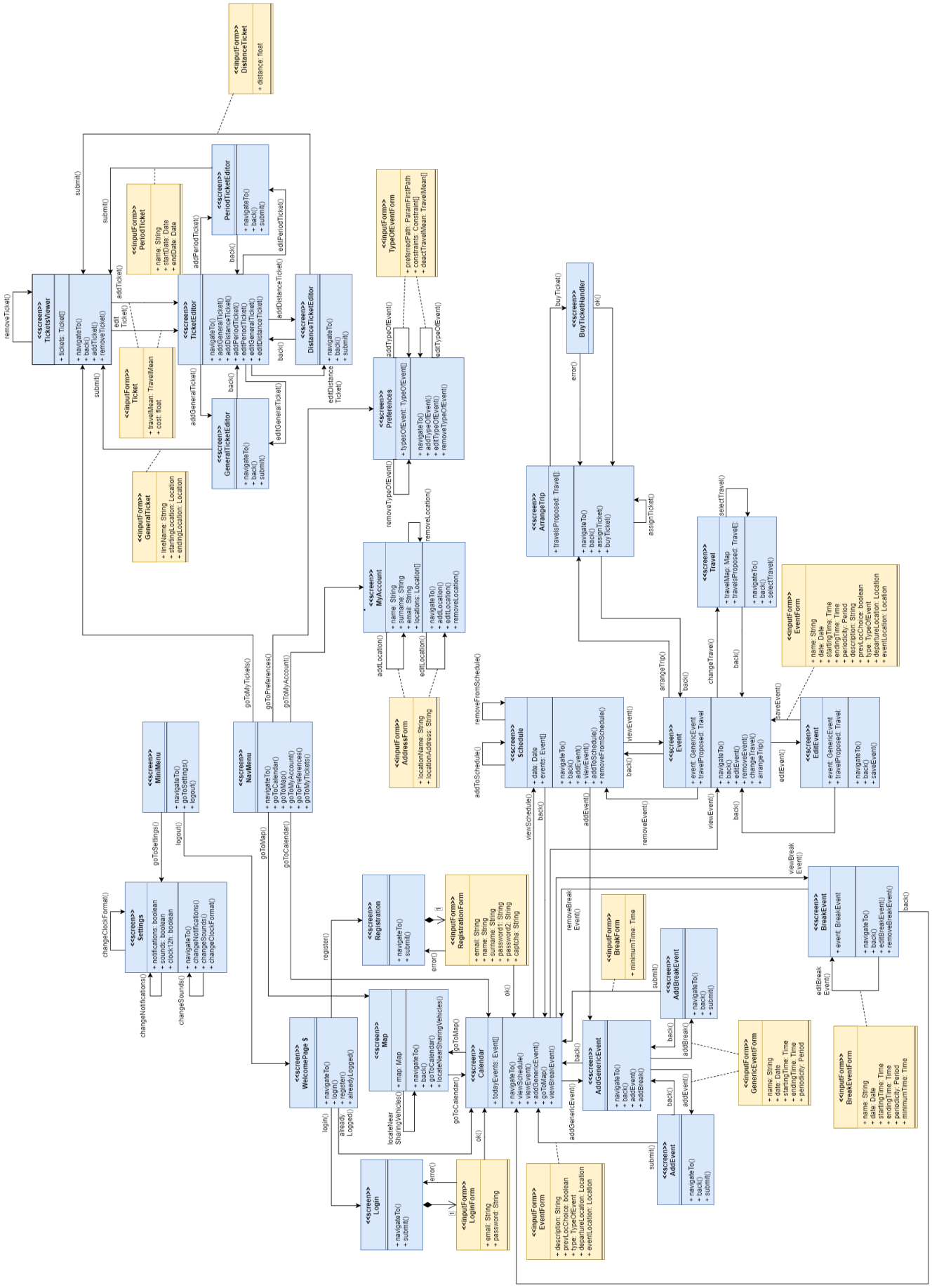


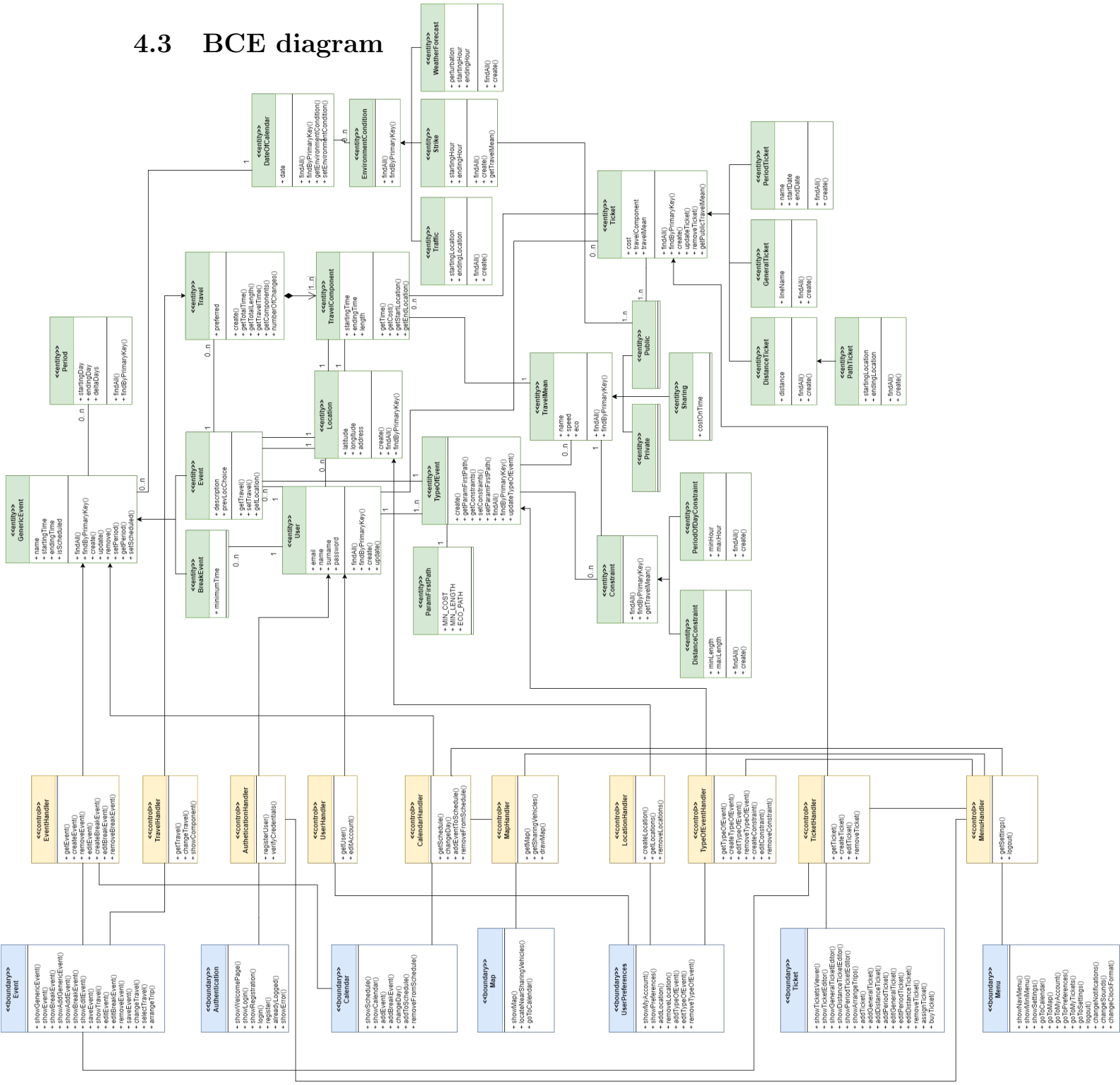
Figure 4.10: Web browser event creation view.

4.2 UX diagrams

The following UX diagram provides additional information about the user interface. Screens are modeled using <<*screen*>> stereotyped classes, while input forms and screen compartments are represented with separate <<*input form*>> stereotyped classes.


<<inputForm>>
DistanceTicket
+ distance: float
<<screen>>
PeriodTicketEditor
+ navigateTo()
+ back()
+ submit()
<<inputForm>>
PeriodTicket
+ name: String
+ startDate: Date
+ endDate: Date
<<screen>>
TicketsViewer
- tickets: Ticket[]
+ navigateTo()
+ back()
+ addTicket()
+ removeTicket()
<<screen>>
TicketEditor
+ navigateTo()
+ addGeneralTicket()
+ addDistanceTicket()
+ addPeriodTicket()
+ editPeriodTicket()
+ editGeneralTicket()
+ editDistanceTicket()
<<screen>>
DistanceTicketEditor
+ navigateTo()
+ back()
+ submit()
<<inputForm>>
Ticket
+ travelMean: TravelMean
+ cost: float
<<screen>>
GeneralTicketEditor
+ navigateTo()
+ back()
+ submit()
<<inputForm>>
GeneralTicket
+ lineName: String
+ startingLocation: Location
+ endingLocation: Location
<<inputForm>>
TypeOfEventForm
+ preferredPath: ParamFirstPath
+ constraints: Constraint[]
+ deactTravelMean: TravelMean[]
<<screen>>
Preferences
+ typesOfEvent: TypeOfEvent[]
+ navigateTo()
+ addTypeOfEvent()
+ editTypeOfEvent()
+ removeTypeOfEvent()
<<screen>>
BuyTicketHandler
+ navigateTo()
+ back()
+ submit()
<<screen>>
ArrangeTrip
+ travelsProposed: Travel[]
+ navigateTo()
+ back()
+ assignTicket()
+ buyTicket()
<<screen>>
Travel
+ travelMap: Map
+ travelsProposed: Travel[]
+ navigateTo()
+ back()
+ selectTravel()
<<screen>>
MyAccount
+ name: String
+ surname: String
+ email: String
+ locations: Location[]
+ navigateTo()
+ addLocation()
+ editLocation()
+ removeLocation()
<<inputForm>>
AddressForm
+ locationName: String
+ locationAddress: String
<<screen>>
MiniMenu
+ navigateTo()
+ goToSettings()
+ logout()
<<screen>>
NavMenu
+ navigateTo()
+ goToCalendar()
+ goToMap()
+ goToMyAccount()
+ goToPreferences()
+ goToMyTickets()
<<screen>>
Settings
+ notifications: boolean
+ sounds: boolean
+ clock12h: boolean
+ navigateTo()
+ changeNotifications()
+ changeSounds()
+ changeClockFormat()
<<screen>>
Registration
+ navigateTo()
+ submit()
<<inputForm>>
RegistrationForm
+ email: String
+ name: String
+ surname: String
+ password1: String
+ password2: String
+ captcha: String
<<screen>>
WelcomePage $
+ navigateTo()
+ login()
+ register()
+ alreadyLogged()
<<screen>>
Map
+ map: Map
+ navigateTo()
+ back()
+ goToCalendar()
+ localizeYourSharingVehicles()
<<screen>>
Calendar
+ todayEvents: Event[]
+ navigateTo()
+ viewSchedule()
+ viewEvent()
+ addGenericEvent()
+ goToMap()
+ viewBreakEvent()
<<screen>>
Schedule
+ date: Date
+ events: Event[]
+ navigateTo()
+ back()
+ addEvent()
+ editEvent()
+ addToSchedule()
+ removeFromSchedule()
<<screen>>
Event
+ event: GenericEvent
+ travelProposed: Travel
+ navigateTo()
+ back()
+ editEvent()
+ removeEvent()
+ removeTravel()
+ arrangeTrip()
<<screen>>
EditEvent
+ event: GenericEvent
+ travelProposed: Travel
+ navigateTo()
+ back()
+ saveEvent()
<<inputForm>>
EventForm
+ name: String
+ date: Date
+ startingTime: Time
+ endingTime: Time
+ periodicity: Period
+ description: String
+ removeOnClose: Boolean
+ type: TypeOfEvent
+ departureLocation: Location
+ eventLocation: Location
<<inputForm>>
BreakForm
+ minimumTime: Time
<<screen>>
AddBreakEvent
+ navigateTo()
+ back()
+ submit()
<<screen>>
BreakEvent
+ event: BreakEvent
+ navigateTo()
+ back()
+ editBreakEvent()
+ removeBreakEvent()
<<screen>>
AddGenericEvent
+ navigateTo()
+ back()
+ addEvent()
+ addBreak()
<<inputForm>>
GenericEventForm
+ name: String
+ date: Date
+ startingTime: Time
+ endingTime: Time
+ periodicity: Period
<<inputForm>>
BreakEventForm
+ name: String
+ date: Date
+ startingTime: Time
+ endingTime: Time
+ periodicity: Period
+ minimumTime: Time
<<screen>>
AddEvent
+ navigateTo()
+ back()
+ submit()
<<inputForm>>
EventForm
+ description: String
+ removeOnClose: Boolean
+ type: TypeOfEvent
+ departureLocation: Location
+ eventLocation: Location
<<screen>>
Login
+ navigateTo()
+ submit()
<<inputForm>>
LoginForm
+ email: String
+ password: String
removeTicket()
submit()
addTicket()
editTicket()
addPeriodTicket()
editPeriodTicket()
addDistanceTicket()
editDistanceTicket()
addGeneralTicket()
editGeneralTicket()
back()
addTypeOfEvent()
editTypeOfEvent()
removeTypeOfEvent()
addLocation()
editLocation()
removeLocation()
goToMyTickets()
goToPreferences()
goToMyAccount()
goToSettings()
logout()
goToMap()
goToCalendar()
changeClockFormat()
changeNotifications()
changeSounds()
register()
error()
ok()
login()
alreadyLogged()
localizeYourSharingVehicles()
buyTicket()
assignTicket()
selectTravel()
arrangeTrip()
changeTravel()
removeFromSchedule()
addToSchedule()
viewSchedule()
addEvent()
removeEvent()
viewEvent()
saveEvent()
editEvent()
viewBreakEvent()
removeBreakEvent()
editBreakEvent()
addBreak()
addGenericEvent()

4.3 BCE diagram



Given an adopted MVC pattern design, this diagram present a plausible representation of the system, separating internal representations of information from information presentation. *Boundaries* are objects, such as user interfaces, that interface with system actors; *Controls* are objects that manage interactions between boundaries and entities implementing the application logic required to process user requests; *Entities* are used to model access to data.

Chapter 5

REQUIREMENTS TRACEABILITY

Component	Requirement
Authentication Manager	**[R1]** the system checks if the e-mail inserted is real; **[R2]** a user cannot sign up with the same e-mail twice; **[R3]** the e-mail and password inserted must be correct; **[R4]** incorrect credentials prevent the user from logging in; **[R38]** a user must be logged.
Path Manager	**[R10]** every travel path proposed must be feasible in the available time (the interval between two consecutive events); **[R11]** if the travel involves two or more travel means, the starting location of the first proposed travel path and the ending location of the last proposed travel path must coincide respectively with the starting location and the ending location of the whole planned travel; **[R12]** the system does not consider paths that violate constraints on travel means defined by the user; **[R13]** the system checks user preferences to decide which feasible travel path is the best; **[R15]** appropriate travel means must be suggested according to the type of event that they are related to; **[R18]** the system must show to the user all possibilities to reach a location in according with the requirements of [G4]; **[R19]** alternative feasible travel paths must not generate overlappings with other events of the schedule; **[R25]** the system does not consider solutions that violate constraints; **[R27]** the system does not consider solutions that include deactivated travel means; **[R28]** for each travel path, the system estimates its carbon footprint produced; **[R35]** the system provides information about time of departure and arrival of the proposed travels; **[R37]** the system provides information about travel time with shared vehicles.
Event Manager	**[R5]** a user must specify all mandatory fields to add the new event; **[R6]** the system reserves the specified time-slot for the event;

	[R7] the system warns the user if the inserted event overlaps with an already existing one; **[R8]** if the ending time of an event is not specified, the systems considers as ending time the hour of departure for the successive event; **[R9]** when an event is inserted after an event without a specified ending time, the ending time of the first event is anticipated as stated in [R8]; **[R29]** the system allows the user to specify a flexible interval and a minimum amount of time to schedule a break;
Schedule Manager	**[R6]** the system reserves the specified time-slot for the event; **[R7]** the system warns the user if the inserted event overlaps with an already existing one; **[R8]** if the ending time of an event is not specified, the systems considers as ending time the hour of departure for the successive event; **[R9]** when an event is inserted after an event without a specified ending time, the ending time of the first event is anticipated as stated in [R8]; **[R14]** the system warns the user if it is not possible to arrive at an event location before its starting time; **[R19]** alternative feasible travel paths must not generate overlappings with other events of the schedule; **[R20]** the combination of the travel paths proposed for the day must be feasible in the allotted time; **[R21]** if there are multiple events at the same time the system will propose in the schedule only the first event added; **[R22]** if the user forces into the schedule an event that overlaps with events already present in the schedule, these are removed from the schedule; **[R30]** if there is enough time for a break, the system reserves it within the specified flexible interval; **[R31]** if there is not enough time into the flexible interval specified, a warning is thrown.
Preference Manager	**[R12]** the system does not consider paths that violate constraints on travel means defined by the user; **[R13]** the system checks user preferences to decide which feasible travel path is the best; **[R15]** appropriate travel means must be suggested according to the type of event that they are related to; **[R23]** the system requires minimum and maximum length allowed for a path to impose a constraint on a travel mean; **[R24]** the system requires an interval of time allowed to impose a constraint on a travel mean; **[R25]** the system does not consider solutions that violate constraints; **[R26]** the system allows the user to specify one or more travel means that cannot be used; **[R27]** the system does not consider solutions that include deactivated travel means.

Trip Manager	**[R32]** the system allows the user to specify all the ticket he already owns; **[R33]** the system shows to the user if he already holds a ticket for a proposed travel; **[R34]** the system allows the user to buy public transportation tickets according to proposed travels; **[R36]** the system shows to the user where sharing vehicles are located.
Complication manager	**[R16]** if a strike occurs, the system will not consider travel means involved in it; **[R17]** if the weather forecasts rain or other adverse conditions, the system will not consider paths involving the bicycle.
Notification Manager	**[R14]** the system warns the user if it is not possible to arrive at an event location before its starting time; **[R31]** if there is not enough time into the flexible interval specified, a warning is thrown.

Table 5.1: Application server components

The *Notification Manager* is used to satisfy the requirements above and also to warn the user when a strike is announced or when the weather forecasts for the following days are adverse.

The *Persistence Manager* allows the system to interact with database and helps the system to keep up-to-date the local database. In order to perform any operation concerning the data, the system queries and updates the server database. Information about events and paths are stored also in the local database of mobile phones: in this way the mobile app doesn't require a connection to show user's schedules.

Chapter 6

IMPLEMENTATION, INTEGRATION AND TEST PLAN

6.1 Implementation Plan

In this section we explain the order in which we plan to implement the subcomponents of the Travlendar+ system. The main criteria we've adopted is to implement first a core of functionalities that we consider to be essential for engaging an initial user base, then we will proceed to insert new and nice-to-have features but that are not mandatory in a first release.
These are the main steps of the implementations we want to follow, in order to reach Travlendar+ full operativity on **server side**:

1. *CalendarManager* (*PathManager*, *EventManager*, *ScheduleManager*, but not *PreferenceManager*), *PersistenceManager* and *AuthenticationManager* are the core of our system and so they are the first to be implemented among with their subcomponents. In particular in this phase the user will not be able to edit his travels but he can only visualize the proposed ones;

2. *PreferenceManager* will be inserted at a later time, in order to filter the user paths. Doing so there will be added functionalities that allow the user to edit and choose the travels he prefers from the feasible ones, which have been proposed to him;

3. *ComplicationManager* will be added together with the *NotificationManager* in order to allow the system to discover if a user's travel is no more feasible and notifies him;

4. *TripManager* will be the last module to be implemented, allowing the user to arrange his trips.

Of course at each implementation phase some data structures are to be added in the database (through the *PersistenceManager*) to properly support the new functionalities.

These are the main implementations steps we want to follow to reach Travlendar+ full operativity on **client side**:

1. Android App is to be developed first, proper updated will be performed when a new functionality is added on server-side, but the first application version is to be released together with the Application Server core functionality release;

2. IOS App will be developed in a second phase: when the Android application is completed (first release). The workforce dedicated to his development will be moved on this task but Android App's support will be taken care of nonetheless;

3. When both IOS and Android applications are released we will proceed to develop the web server that will allow the users to use Travlendar+ from any web browser.

6.2 Integration Entry Criteria

Before starting the integration and testing phase there are a number of conditions that have to be met:

- **Documents:** this document (DD) and RASD have to be completed;
- **Proper Documentation:** Every method and class, before being tested must be provided with proper documentation and/or comments in order to ease the testing process and also make easier the reuse of the classes and their methods;
- **Code Inspection and Analysis:** At least one method, either code inspection or automated data flow analysis, has to be performed on the modules and classes before we proceed to the integration and test phase that involves them;
- **Starting Condition:** The integration process can start only if the component involved offers at least 75% of their functionalities to-be-implemented, this means that this phase can start in parallel with the completion of that component, and not that the testing phase can end with some functionalities still to be implemented;
- **Unit tests:** Before starting the integration of one component, it must be tested through proper unity tests, in order to guarantee a correct behavior of his internal mechanisms.

6.3 Elements to be integrated

Basically all the system components, specified in section 2.2, need to be integrated together; here we specify in detail all the integrations that need to be performed.
On Application Server side:

- *ScheduleManager*, *EventManager* and *PathManager* need to be integrated together;
- *PreferenceManager* needs to be integrated with *PathManager* (NB: by doing that the entire *CalendarManager* subsystem will be integrated);
- *TripManager* needs to be integrated with *ComplicationManager*;
- *ComplicationManager* needs to be integrated with both *NotificationManager* and *TripManager*;
- *PersistenceManager* needs to be integrated with our *DBMS*;
- All the components that rely on *RESTful APIs* to implement the interaction with the clients need to be integrated and tested;
- Basically all the application server components rely on *PersistenceManager* and therefore they need to be integrated with it, the same consideration is valid for the *AuthenticationManager* (except for the *NotificationManager* that does not interact with it).

On client side (App Mobile):

- *DBManager* needs to be integrated with the *LocalDatabase*;
- *ApplicationController*, *GUIManager* and *DBManager* needs to be integrated together.

On web server side *WebController* and *DynamicWebPages* need to be integrated together. All the interactions that single components have with external services APIs have to be tested and their correctness have to be guaranteed before integrate and test phase involving those components begins.

6.4 Integration Testing Strategy

The integration testing strategy we will adopt will be defined by a mix of two strategies and according to the implementation plan.
The first strategy we are going to use is based on a **bottom-up approach**. This choice will allow us to start from the less-dependent components and then climb the "uses" hierarchy, through the use of proper drivers. The bottom-up approach will enable us to follow the implementation plan, that follows a similar approach. Doing so we will improve the efficiency and the parallelism of the development process.
The bottom-up strategy will be mixed with a **critical-module-first approach**, in order to start the integration and testing of the most critical modules of our system, those contained in *CalendarManager*. We've considered also this second strategy in order to avoid issues related to core components of our system and threats to the correct behavior of our system, especially to guarantee the absence of failures or bugs that could prevent the users to correctly take advantage of our platform and its functionalities.

6.5 Sequence of Components integration

The following subsections aim to describe the order in which Travlendar+ will have to be integrated and tested. We will use as notation an arrow going from component A to component B means that component A is necessary to component B and so it must have been already implemented before performing the integration.

6.5.1 Software integration Sequence

Since our system relies on some external systems and interacts with theirs external interfaces (APIs) we will assume that those system are already proper tested. In order to test the interaction with all the external systems, we'll use first proper stubs that emulate their behavior and then we'll use the real external system; this is to be done in order to avoid that test are slowed, that connectivity issues cause test to fail and that during the tests we hit some APIs rate limits (for example with *Google Maps APIs*).

PersistenceManager

The first component to be integrated is the *PersistenceManager*, since all other modules that have to access the data need this module in order to work properly. *PersistenceManager* is to be integrated with the *DBMS*. To do some drivers will be needed in order to perform queries, data insertions and deletions; A test database has to be introduced to perform such tests.

Figure 6.1: I&T - PersistenceManager

AuthenticationManager

Nearly every core-component requires the *AuthenticationManager* to work properly. It is to be integrated with the *PersistenceManager*. A driver will simulate every possible authentication request.

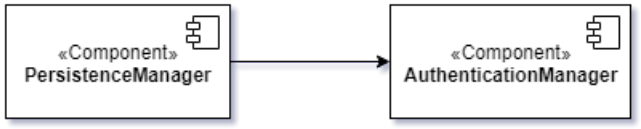


Figure 6.2: I&T - AuthenticationManager

CalendarManager

The *CalendarManager*'s components interact all with *AuthenticationManager* and *PersistenceManager*, and so each component will be integrated with them. Since all *CalendarManager*'s components offer an external interface to be exposed as *RESTful* services, a driver for each one of this components will be needed.
The *CalendarManager*'s components are strictly correlated and so, in order to minimize the number of drivers and stubs, their integration order will be the following:

1. *ScheduleManager* and *PreferenceManager* are to be integrated with *AuthenticationManager* and *PersistenceManager*.

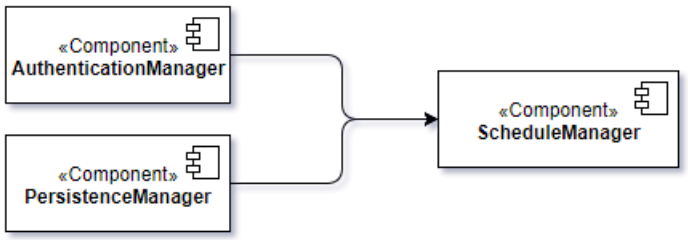


Figure 6.3: I&T - ScheduleManager

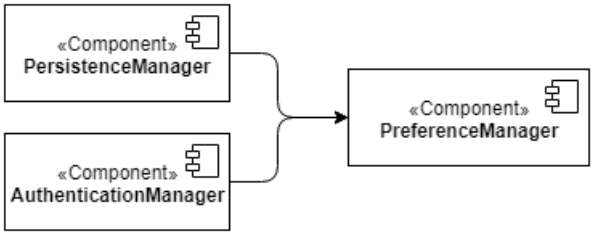


Figure 6.4: I&T - PreferenceManager

2. *PathManager* is to be integrated with *PersistenceManager*, *AuthenticationManager*, *GoogleMaps APIs*, *PreferenceManager*, *ScheduleManager* and *TSP APIs*, following this order no stubs will be needed to perform the integration. It is required another driver, beside the one used to test the external interface, in order to simulate the calls performed by *EventManager* to *PathManager*.

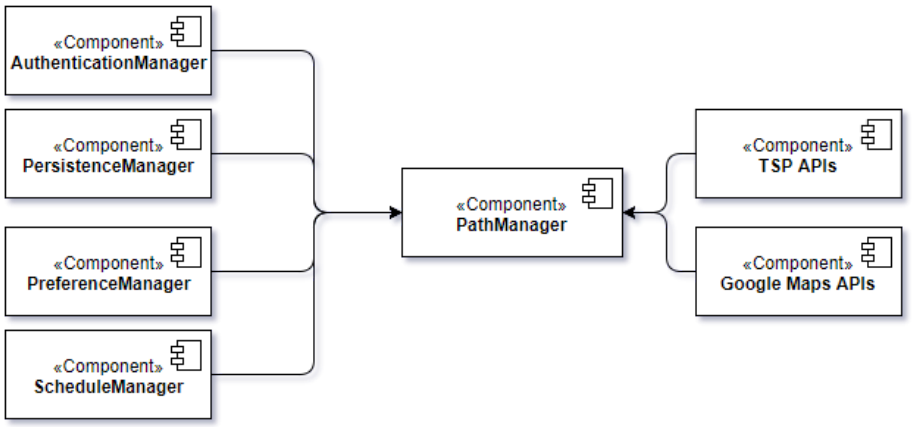


Figure 6.5: I&T - PathManager

3. *EventManager* is to be integrated with *PersistenceManager*, *AuthenticationManager*, *ScheduleManager* and *PathManager*, following this order no additional drivers or stubs will be needed.

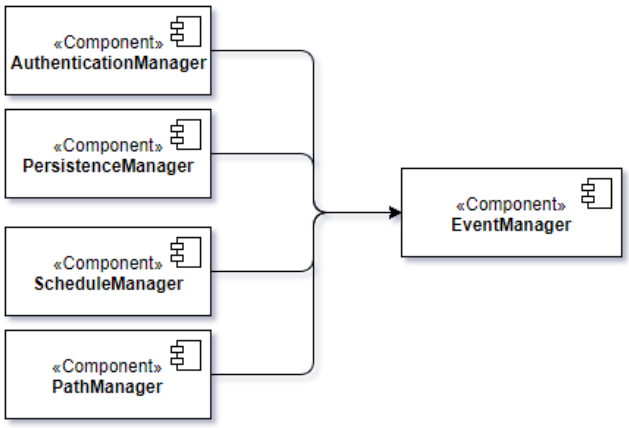


Figure 6.6: I&T - EventManager

TripManager

TripManager is to be integrated with *PersistenceManager*, *AuthenticationManager* and then with *TSP APIs*. Following this order a stub will be needed for the methods of *TSP APIs* called by *TripManager*. The driver needed for this module will simulate user's requests related to the trips-arranging methods provided.

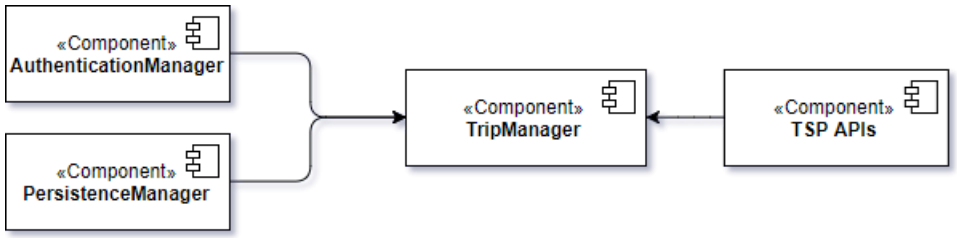


Figure 6.7: I&T - TripManager

NotificationManager

NotificationManager interacts only with *GCM APIs* and so it is to be integrated with.

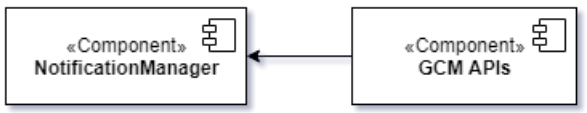


Figure 6.8: I&T - NotificationManager

ComplicationManager

ComplicationManager is to be integrated with *PersistenceManager*, *TSP APIs*, *GoogleMaps APIs*, *Weather API's* and *NotificationManager*, following this order a stub will be needed for the methods of *NotificationManager* called by *ComplicationManager* (basically when *ComplicationManager* will have to notify the users it will try to forward notification that will be received by the

stub). The integration of the three external APIs can be done in any possible order, since the interactions are independent from one another.

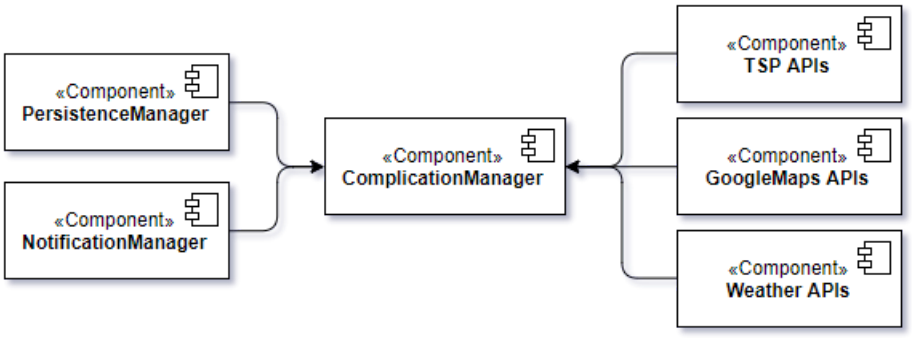


Figure 6.9: I&T - ComplicationManager

Overall integration of the Application Server

The following picture is just a recap of the integration and test plan of the *Application Server*. The integration order starts from the top of the picture and ends in the bottom.

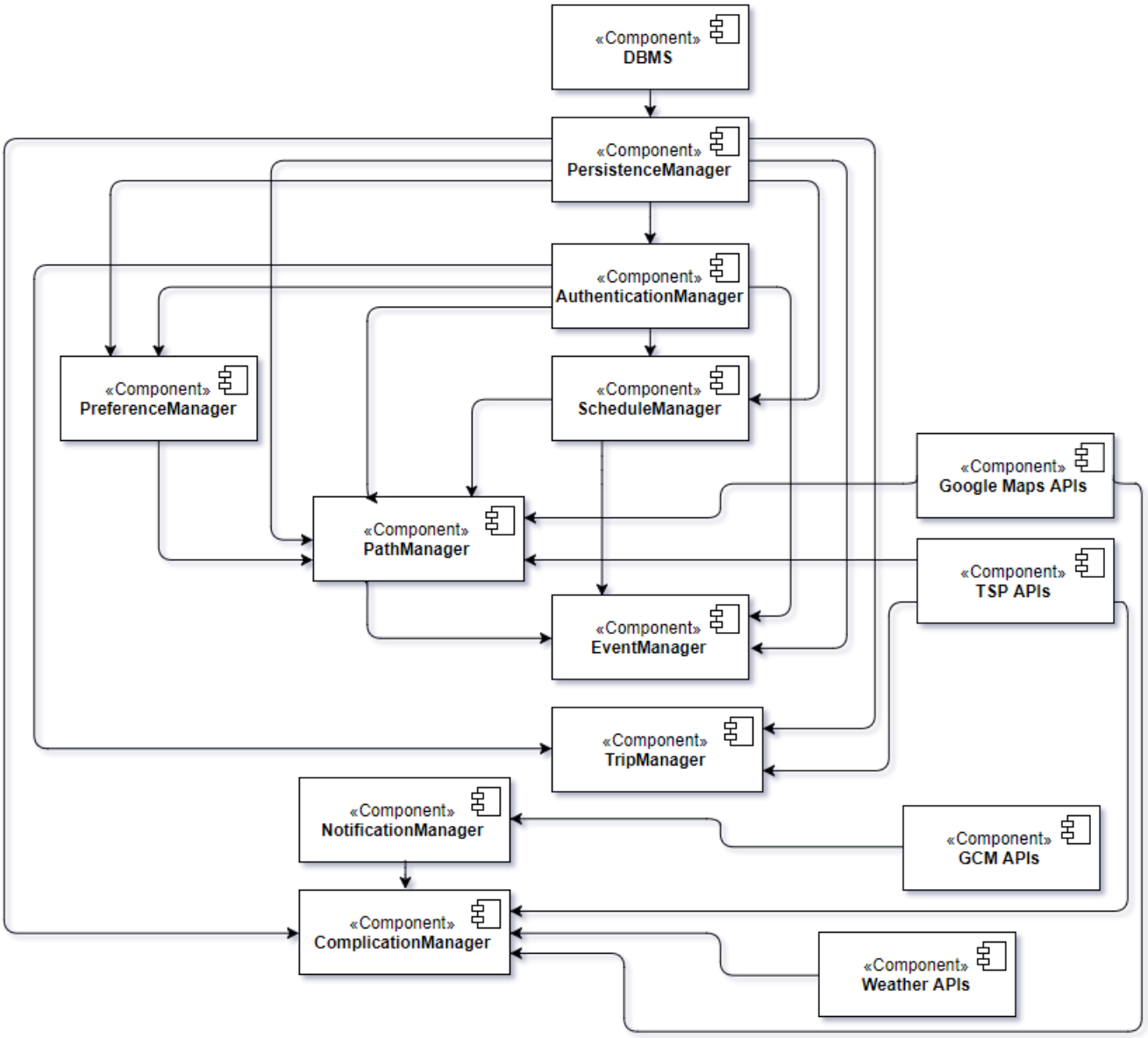


Figure 6.10: I&T - Application Server

WebServer

In the web server subsystem the *WebController* component is to be integrated with *DynamicWebPages* and then with *Google Polylines APIs*.

Figure 6.11: I&T - WebServer

AppMobile

The order of integration and testing of the *AppMobile*'s components is the following: first *DBManager* with *LocalDBMS* and then *ApplicationController* with *DBManager*, *GUIManager*, *GCM API's* and *Google Polylines APIs*.

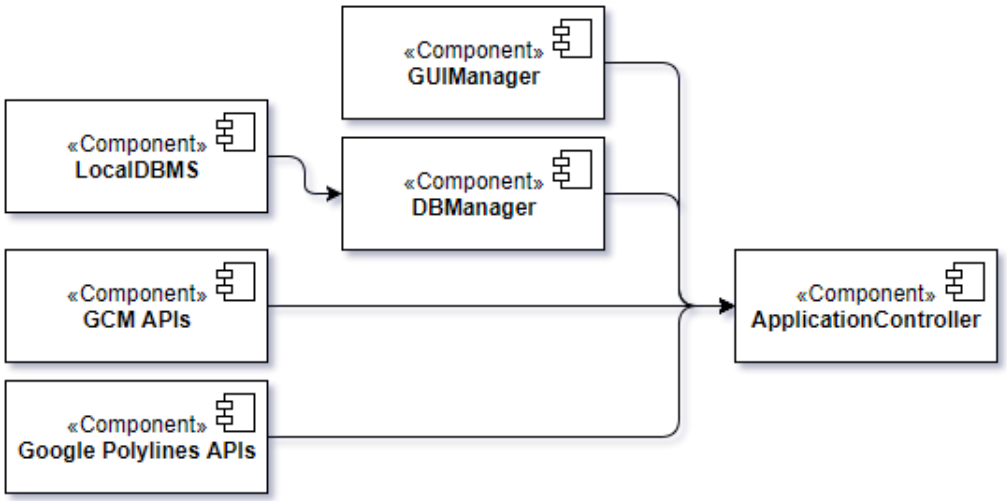


Figure 6.12: I&T - AppMobile

6.5.2 Subsystems integration Sequence

Once all subsystems will be fully integrated, this is the order in which they are to be integrated together in order to deploy the full Travlendar+ infrastructure.

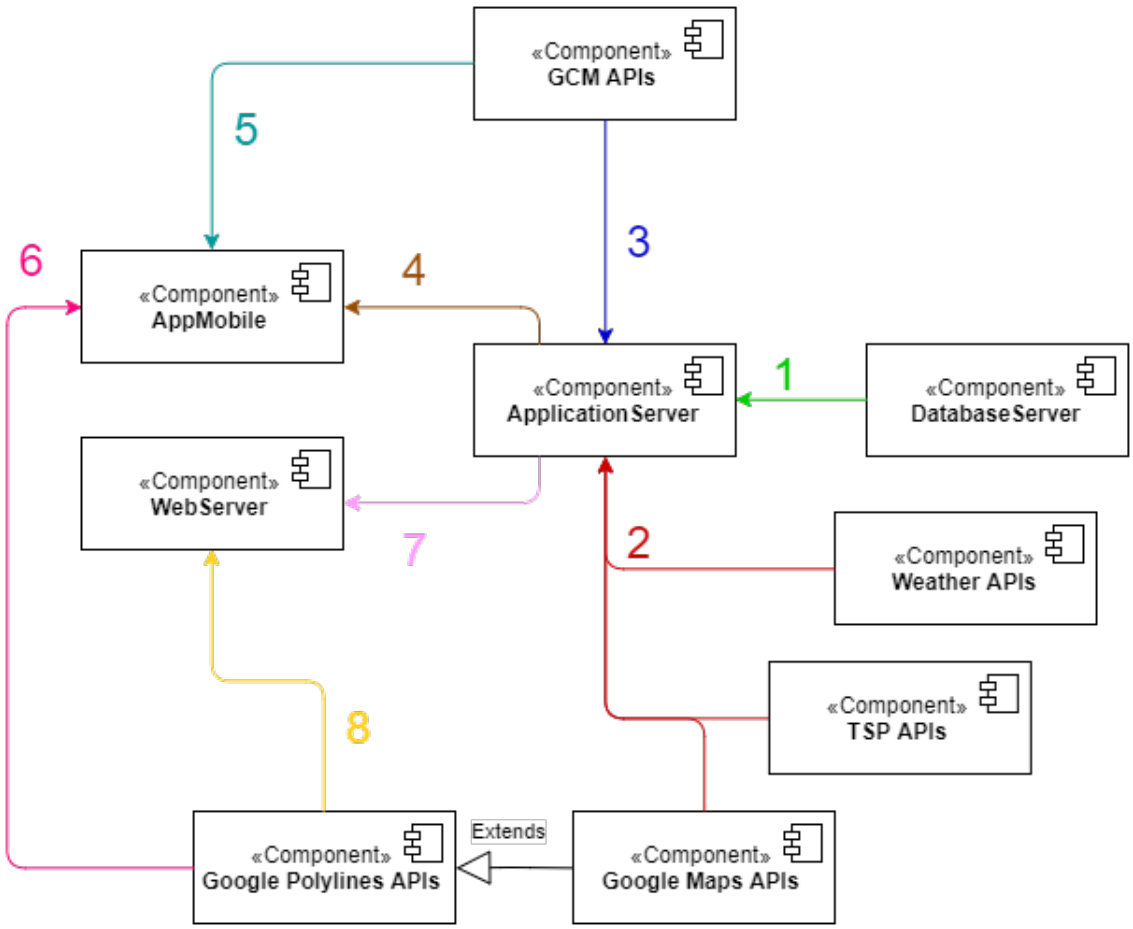


Figure 6.13: I&T - AppMobile

Chapter 7

EFFORT SPENT

7.1 Pietro Melzi

Date	Task	Hours of work
06/11/17	Group session - Overall architecture proposed by Salvadore	1
09/11/17	Group session - Discussion on architecture components	1
11/11/17	Group session - Requirements traceability, considerations on sequence diagrams and other parts of document's section two	8
12/11/17	Sequence diagrams written on paper	2
13/11/17	Group session - Sequence diagrams written with software, discussion on algorithms	7
14/11/17	Considerations on sequence diagrams and requirements traceability, security algorithm on paper	3
15/11/17	Considerations on other algorithms with Salvadore	1
18/11/17	Group session - Encryption, univocal code and local DB updating algorithms, general considerations	8
19/11/17	Corrections in the document, description of calendar interface components	5
20/11/17	Group session - Fixes on component interfaces and sequence diagrams	2
25/11/17	Group session - UX and BCE revision and document review	6:30

7.2 Alessandro Pina

Date	Task	Hours of work
06/11/17	Group session - Overall architecture proposed by Salvadore	1
09/11/17	Group session - Discussion on architecture components	1
11/11/17	Group session - UX Diagram	8
13/11/17	Group session - App mockups extension	7
14/11/17	Web mockups revision	1
14/11/17	UX diagram and mockups	4
16/11/17	Document revision	2
18/11/17	Group session - Algorithm and ER diagram	8
20/11/17	Group session - BCE diagram	2
21/11/17	BCE diagram	2
23/11/17	BCE diagram and document review	3
25/11/17	Group session - UX and BCE revision and document review	6:30

7.3 Matteo Salvadore

Date	Task	Hours of work
06/11/17	Group session - Overall architecture proposed by me	1
09/11/17	Group session - Discussion on architecture components	1
11/11/17	Group session - Main decisions on system's structure and definition of system components	8
12/11/17	Component diagrams and overview of the system	4
12/11/17	Deployment diagram, some parts of the first chapter and architectural styles	4
13/11/17	Group session - Component diagrams, Other design decisions, implementation plan	7
14/11/17	Integration and test plan	1:30
15/11/17	Algorithm discussion with Melzi	1
17/11/17	Component and Deployment diagrams improved	1:30
18/11/17	Group session - I & T diagrams, computePath algorithm	8
19/11/17	ComputePath algorithm, I & T, implementation choices	5
20/11/17	Group session - Fixes on component interfaces and computePath	2
20/11/17	Component interfaces	1
21/11/17	DD revision: 1st chapter, software used, overview, component diagram	1:30
22/11/17	DD revision: Deployment and Runtime views, component interfaces, architectural styles, other design decisions, algorithms, UI design	2
23/11/17	DD revision: I & T, fixed missing interface in component diagram and his propagation through all the document	1
25/11/17	Group session - UX and BCE revision and document review	6:30

Chapter 8

REFERENCES

8.1 Bibliography

(I) Hans van Vliet - Software Engineering: Principles and Practice;

(II) ICSEA 2015 - Best Practices for the Design of RESTful Web Services: `http://cm.tm.kit.edu/CM-Web/05.Publikationen/2015/[GG+15b]_Best_Practice_for_the_Design_of_RESTful_Web_Services.pdf`;

(III) MVC in apple systems: `https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html`;

(IV) RSA Wikipedia page: `https://it.wikipedia.org/wiki/RSA`;

(V) Agile Modeling website- UML component and deployment diagrams `http://agilemodeling.com/`.

8.2 Software used

This document is written in LaTex, a markup language used for text editing, and includes sections realized with different software tools. Here we show the list of used software:

- Texmaker 5.0.2 (compiled with Qt 5.8.0) for writing the document;
- StarUML 2.8.0 for component and deployment diagrams;
- draw.io for overall architecture, E-R, UX, BCE and I&T diagrams;
- Mockplus 3.2.6 for mockups;
- sequencediagram.org for sequence diagrams.