Politecnico di Milano
AY 2017/2018

Travlendar+

Design Document

Mirko Salaris, Piervincenzo Ventrella, Pietro Cassarino

November 26, 2017

Deliverable:	DD
Title:	Design Document
Authors:	Mirko Salaris, Piervincenzo Ventrella, Pietro Cassarino
Version:	1.0
Date:	26-November-2017
Download page:	https://github.com/mirkosalaris/CassarinoSalarisVentrella/
Copyright:	

Table of Contents

List of Figures

1 Introduction

1.1 Purpose

This document is the Design Document for the Travlendar+ application. Its aim is to provide a description of the system in terms of architectural components. DD contains a description of the architectural design using component diagrams and sequence diagrams. It shows how each component is built, how it interacts with other components and how with the external actors involved. This document's aim is also to provide a technical explanation of the behaviour of some component using algorithms. It also shows user interfaces through graphical screen representation.

1.2 Scope

Travlendar+ is an application in which the User can register and handle his appointments, eventually inviting other people. The app allows visualizing the best travel solutions to reach the place of an appointment, it also allows checking the planned schedule and to personalize the system by setting preferences and constraints (for instance, the User can choose eco-friendly solutions or set a flexible break window time for lunch).
The application contains a Notification system and its role is to notify users about incoming appointments, bad weather, traffic and strikes. In this way there is the possibility to organize, in an optimal way, all the travel plans related to the appointments.
Travlendar+ also allows purchasing tickets of the transportation means belonging to the companies affiliated with the system. If the User doesn't want to take a public transportation mean, he/she can choose to opt for a sharing vehicle, another service offered by the application.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

Here we provide a list of definitions of words and expression used in the documents. Every time such words or expressions will be used they will be preceded by the symbol "(↑)" that will be a link to this section.

– ***Incoming Appointment***: the next scheduled appointment for which the S2B send a reminder to the user. The reminder is sent a certain amount of time before the appointment starts, to allow the user to get on time in the location.

– ***Future Notification***: a notification that is generated but it will be eventually sent only in a future time. The time has to be specified during the creation.

– ***Dynamic Configuration***: with this term we mean a reconfiguration that can be done without powering off the server or any physical component.

1.3.2 Acronyms

- *S2B*: System to Be;

- *API*: Application Programming Interface;

- *RASD*: Requirements Analysis and Specification Document

1.4 Revision history

1.5 Document Structure

1. ***Introduction***: This serves as an introduction to the document to illustrate its purpose, scope, the conventions that will be used and its structure.

2. ***Architectural Design***: After providing an overview of the system, in this sections we also include all the details of the architecture and the related design decisions. We start from the data model and we describe the components and their role. After a static description, a runtime analysis of the interesting components is provided. Finally we describe the main component interfaces.

3. ***Algorithm Design***: In this section we describe the most interesting Algorithms we identified in the system, how they works and their context.

4. ***User Interfaces Design***: After the technical description of the previous section, here we provide indications on the User interactions with the App and mockups of the screens related to the main functionalities.

5. ***Requirements Traceability***: This section explains the rationale behind our design decisions in terms of a mapping between the goal/requirements defined in the RASD and the components illustrated in this document.

6. ***Implementation, Integration and Test Plan***: In this last technical section, we provide a Plan for the whole development process, giving indications on the general approach, the priorities and the details of the process.

7. ***Effort Spent and Team Work***: Here we include a summary of the Effort Spent by each member of the team and indications on his responsibilities.

8. ***References***: This is the section in which we include details on Software and Tools used and the Reference Documents on which we based our work.

2 Architectural Design

2.1 Overview

Here we provide a high-level representation of client-server interaction and of the submodule of the server. Moreover, we include a brief indication of the deployment.
The orchestrator is needed only on the client request: his role is to dispatch the request to the appropriate component based on the type of request. After that, the component can communicate directly with the client.
In the server, the orchestrator and all the other submodules are stateless. We thought to use the elastic component architectural pattern for the components. Moreover, the orchestrator can eventually be duplicated using a fixed dimension pool whose size is configurable by the system administrator.
The diagram only shows the overall interaction between client and server and the role of the orchestrator: interactions between the submodules are not shown. Further details on the submodule and the overall interactions will be provided in the following sections.

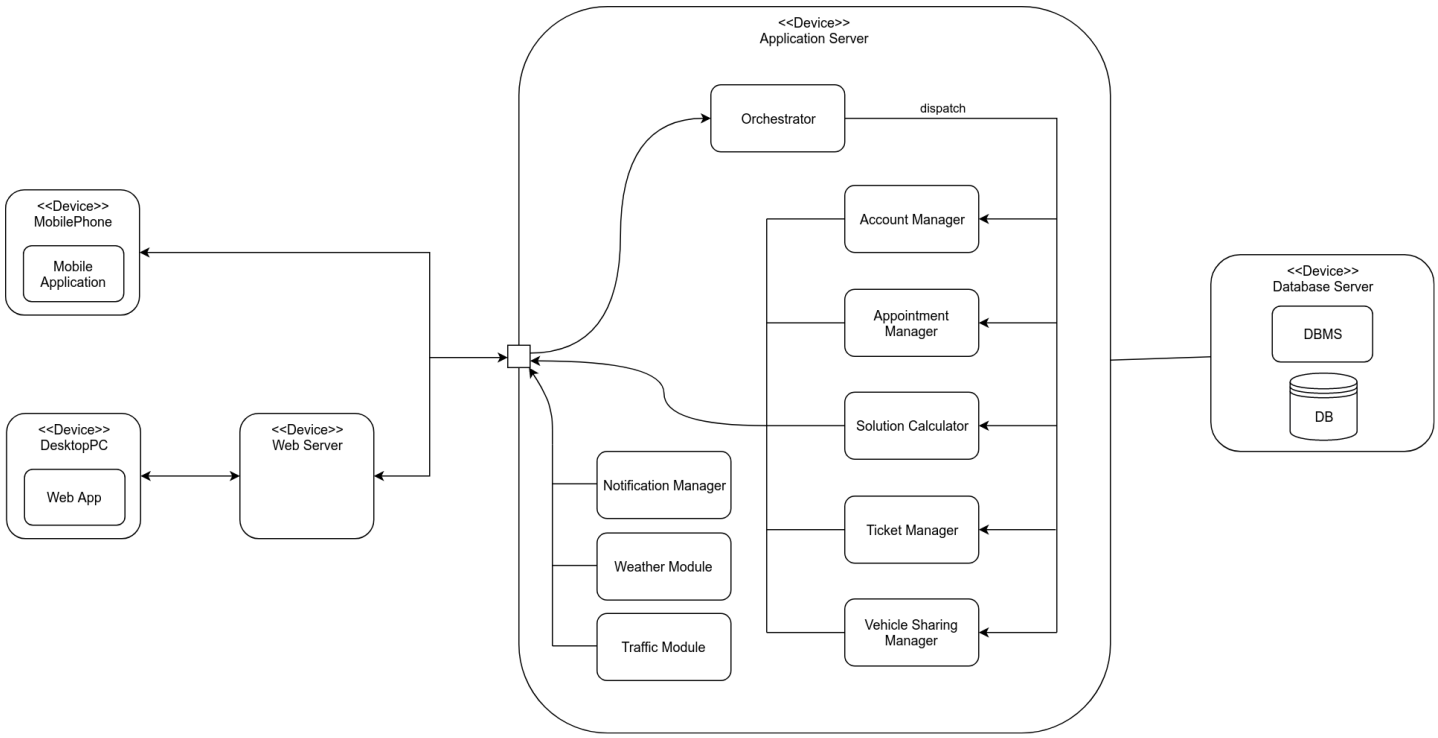

Figure 1: Overview - High level architecture

2.2 Component view

This section starts with the presentation of the Entity-Relationship diagram. Some of the diagrams following the ER model, we include a fictitious component, App, that will be highlighted in green and serves the purpose of representing both the mobile app and the web app *(through the web server)*, without adding complexity to the diagrams.

2.2.1 ER diagram

The following ER diagram represents the conceptual schema of the system database. Ticket Entity has to be considered as an *"abstract"* entity, meaning that we don't know and don't specify all the attributes because they depend from the ticket Typology and from the Transportation Company to which they belong. We only provide a classification of Ticket into two categories, ordinary and pass tickets. Below these two, we include examples of "real tickets" just to convey how the schema will be at implementation time.

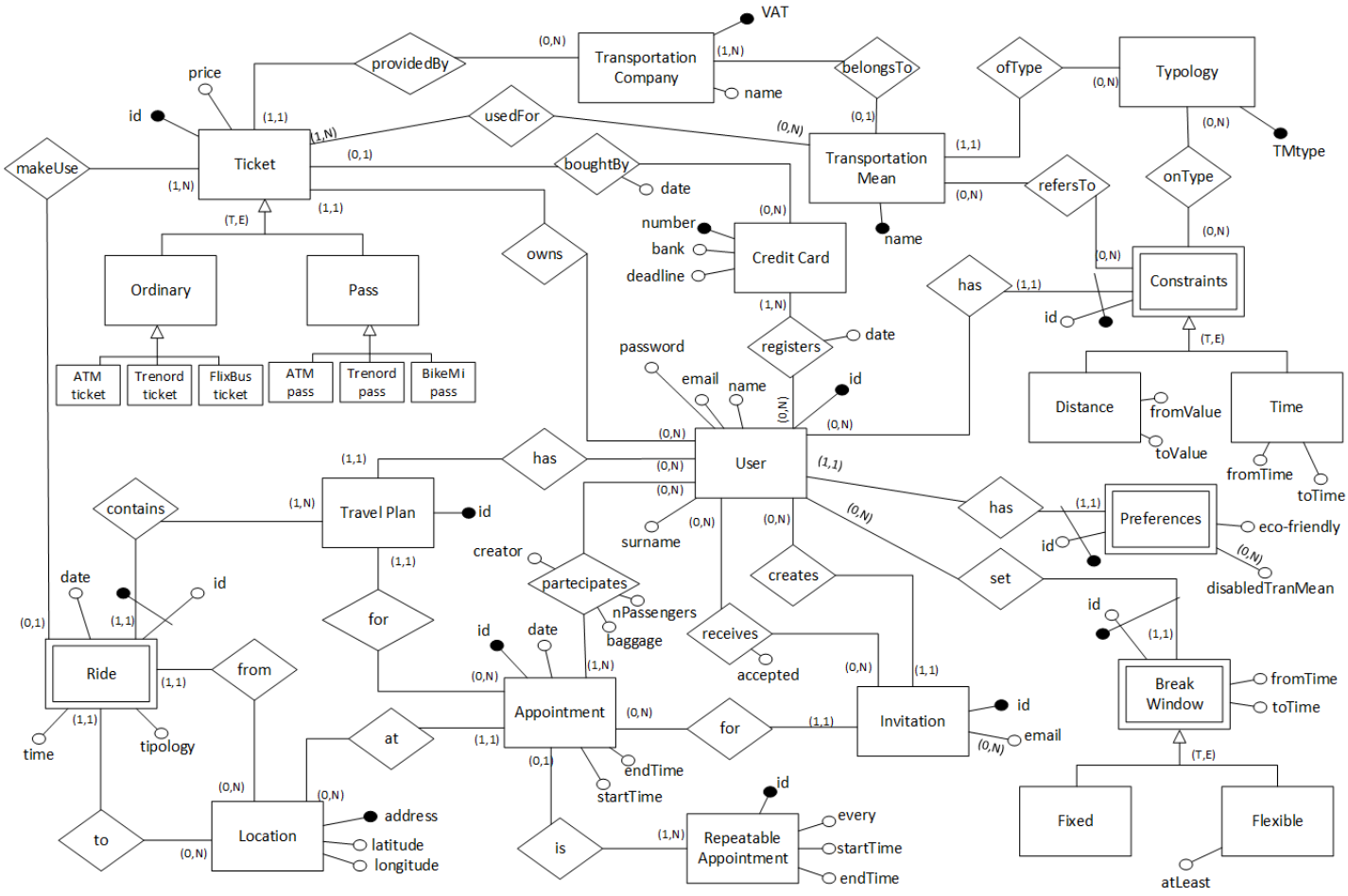

Figure 2: ER diagram

2.2.2 Appointment Manager and Notification Manager

Here we define the Appointment Manager and the Notification Manager. The Notification Manager will appear in others diagrams, but only here is detailed with its subcomponents.

Appointment Manager has three subcomponents:

– *Appointment Handler*: it is the only subcomponent communicating with the database: it manages all the read and write operations related to appointments. It is responsible for the final consistency check of the appointment before writing it to database. It exports an interface to let external components read appointments details and an internal interface to let subcomponents of Appointment Manager read and write appointments' details.

– *Appointment Editor*: this provides an interface to easily change appointments' details and store it to database through the interface of Appointment Handler. Every time an appointment is created or details of an appointment change, this sends to Incoming Appointment Scheduler the appointment. This module interacts with an external module "Solution Calculator" to compute travel solutions. In this diagram it is displayed in dashed line because it will not be described here.

– *Incoming Appointment Scheduler*: for each appointment (provided by Appointment Editor) this module executes an algorithm to decide when that specific appointment will become an (↑)*incoming appointment*. An appointment becomes incoming according to a combination of the time of the appointment and the travel plan chosen, to let the User receive a notification at an appropriate time. This is done generating a (↑)*future notification* through the interface provided by Notification Manager.

Notification Manager provide an interface for the creation of new notification and it is composed by two module:

– *Notification Scheduler*: this is the module whose interface is actually exported and it manages the creation of new notifications, stores them in the database and schedules their dispatch. When it is time to send a notification this module uses the interface provided by Notification Dispatcher and delegates to it the dispatching procedure.

– *Notification Dispatcher*: it is in charge of the actual dispatch of the notification.

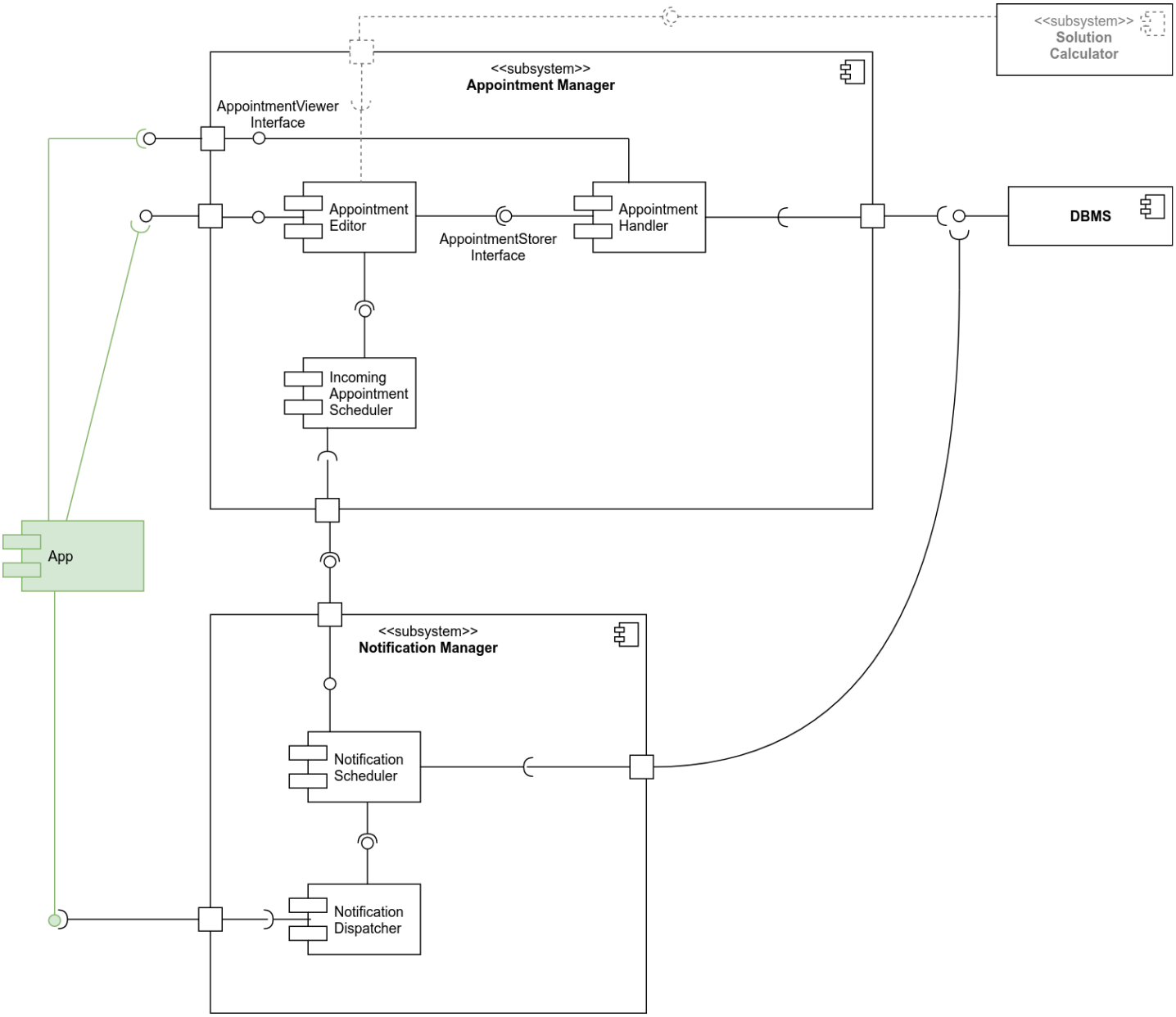

Figure 3: Component Diagram - Appointment Manager & Notification Manager

2.2.3 Invitation Manager

The Invitation Manager component is the module that manages all the aspects related to invitations, from their creation to the acceptance/refusal of the invited Users/Persons. It is divided into two sub-components. A sequence diagram is provided in section 2.3.

– *Invitation Generator*: it manages the invitation from the point of view of the creator of an appointment: creation and management of invitations

– *Invitation Handler*: it is concerned with the acceptance/refusal of the invitations sent.

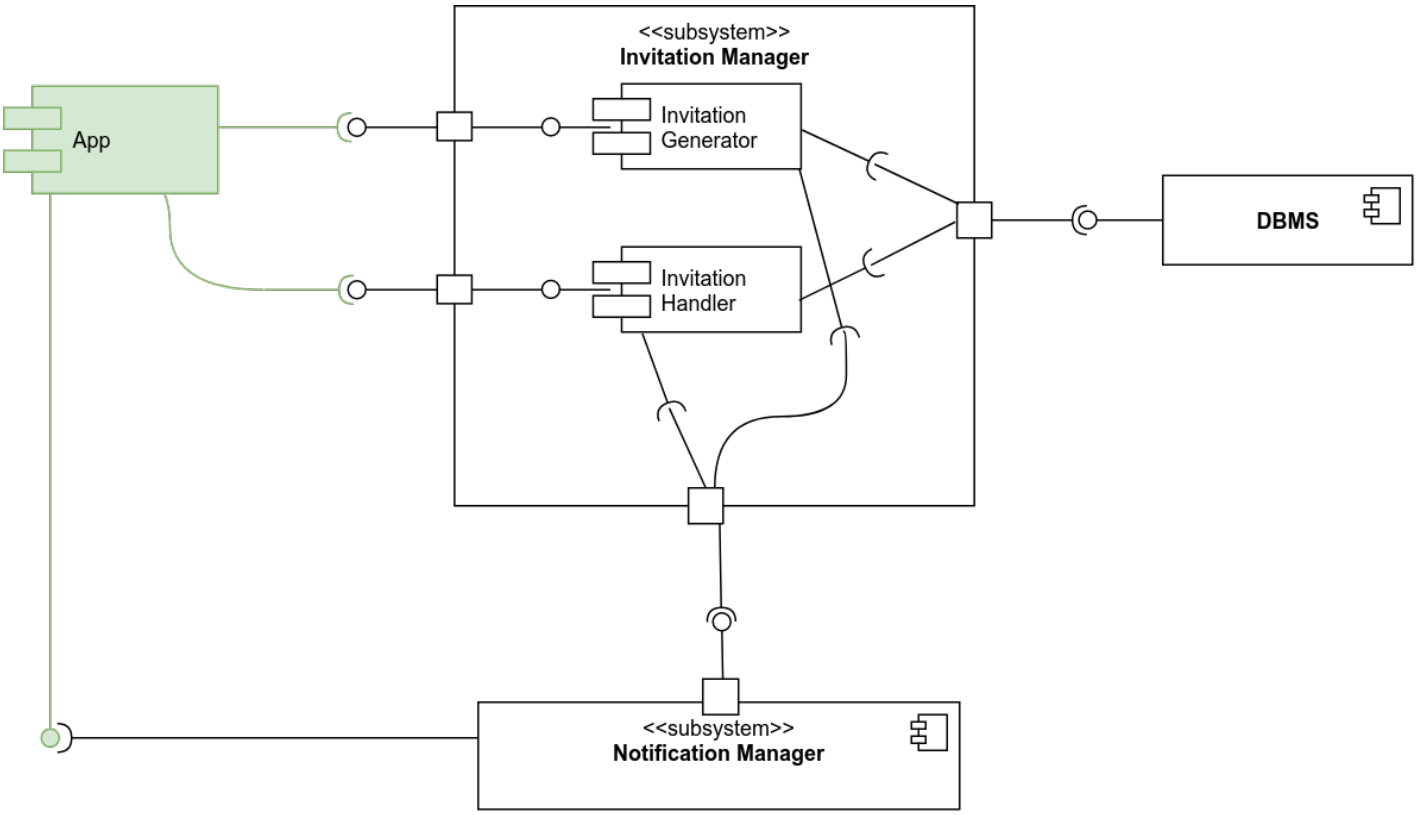

Figure 4: Component Diagram - Invitation Manager

2.2.4 Weather and Traffic Modules

In the diagram, we provide the representation of the Weather Module. The traffic module has exactly the same architecture and interactions. You can just derive the traffic diagram substituting *"Weather"* with *"Traffic"*. Both modules are needed to notify the User of changes in traffic/weather that can influence his/her daily travels and they are used by the component Solution Calculator to avoid bad solutions like, for instance, taking the bike when it rains. With the same convention as before, we represent here Solution Calculator but it will not be detailed here.
The Weather *(traffic)* module has a quite articulated architecture and it will be explained here and in section 2.3 with an object diagram.
It is composed by:

– *Address Solver*: this component has a really basic functionality and it is exploited by Weather Manager to "understand" *(see below)* addresses. Provided an address as input, it decomposes it in its components and returns a hierarchy.
Example: "Italy, Milan, via Pacini 32" -> transformed to:
"State:Italy"→"City:Milan"→"CityZone:NorthEast"→"Street:Giovanni Pacini"→"Number:32"

– *Weather Manager*: this serves as a registry to support the publish-subscribe architecture between Weather Querier and Weather Notifier. Moreover, this component provides an external interface to let other components have information about weather. When it receives a request of weather information for a specific address, it asks Address Solver to interpret the address and then it asks the appropriate Weather Querier for the information.

– *Weather Querier*: it is the module responsible for retrieving the weather information and send them to the Weather Notifier.

– *Weather Notifier*: this is the component that decides if and when a User has to be notified. It subscribes to a specific Weather Querier and receives the information from it.

Both the Querier and the Notifier are instantiated by zone. The zones of the Querier are meant to be very wide, like Italy, Spain, France and the zone of the Notifier are meant to be more specific, for instance: West Milan, East Milan, Venice, Turin.
The Notifier has to execute an algorithm for each appointment to decide if and when to notify the User, so its load will depend on the number of appointments in that specific zone and (especially in the Traffic Module) on the day of the week and the time of the day. For this reason, a load balancing mechanism is required. When a 'node' is under load it can be split into two or more parts: "Milan"→"WestMilan" and "EastMilan".
The Querier has a more stable load and, for this reason, **no** load balancing mechanism has to be implemented. By the way, the zones of the Querier can be configured by the System Administrator, even dynamically.
The load balancing mechanism of the Notifier and the (↑)*dynamically configurable* Querier has to be supported by the Manager: it is not a simple registry but it registers all the active Queriers and Notifiers and it manages the subscriptions when they are modified.
The process is described in more details in section 3.

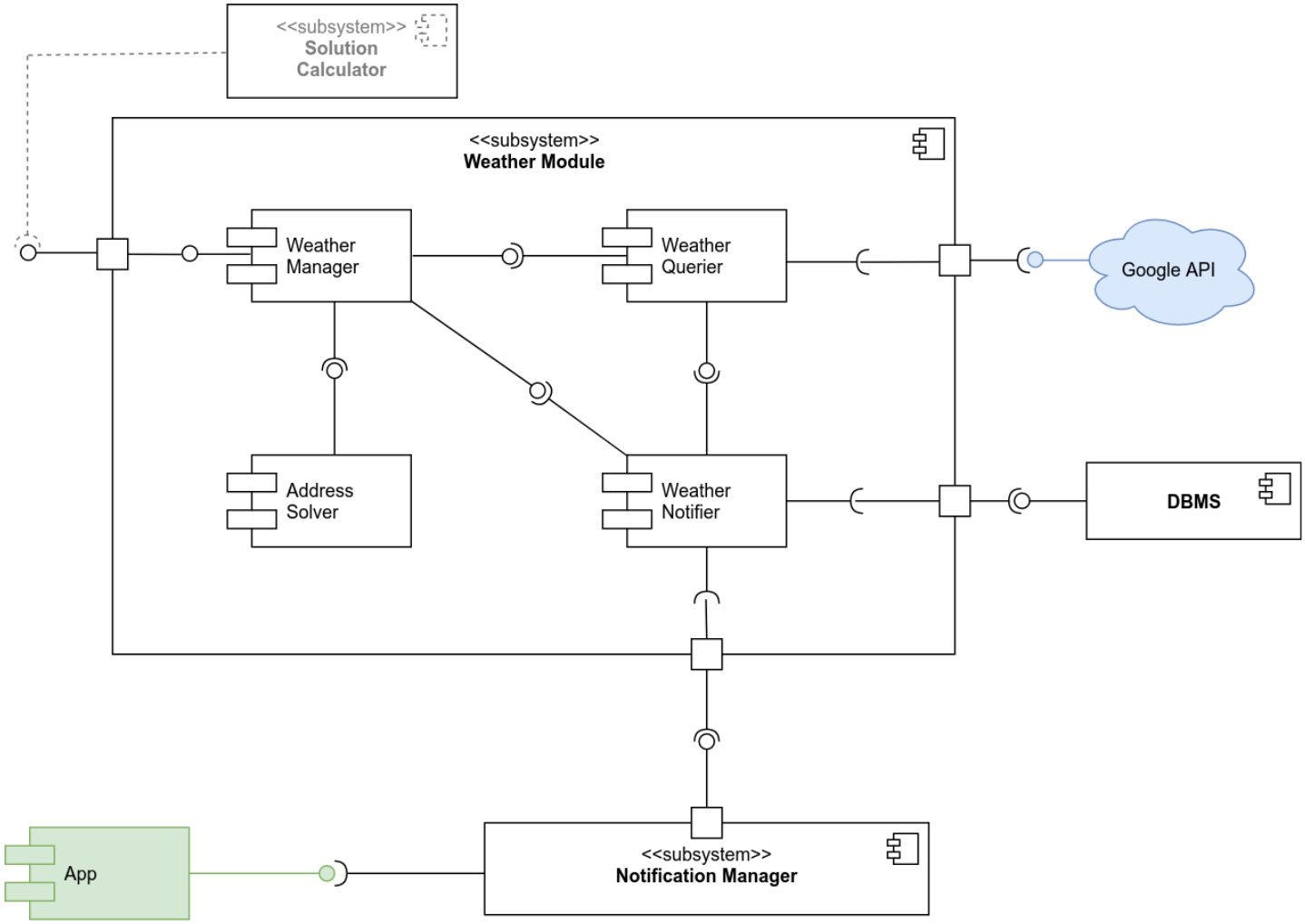

Figure 5: Component Diagram - Weather Module

2.2.5 Ticket Manager

Ticket Manager role is to handle the process that goes from the User selection of a travel plan solution to the purchase or the registration of a ticket.
It is composed by:

– *Ride Controller*: it receives the travel plan selected by the User and, for each ride, checks if it contains a transportation mean of an affiliated company which requires tickets. Ride Controller forwards to Purchase Manager all the rides for which the User could buy a ticket. At the end of the process, Ride Controller provides the list of possible tickets to the user.

– *Purchase Manager*: it provides to Ride Controller the list of possible tickets to purchase. The DB contains only the ticket models of the affiliated transportation companies, so it could be necessary to interface with the Transportation Companies API in order to obtain, in real time, additional information about the ride (e.g., number of available seats, departure time, etc.). For instance, if we need an ATM bus ticket, Purchase Manager takes it from the DB, because the ticket model is enough (for this kind of ticket we don't need to specify departures time or seat number); at the contrary, if we need a Trenitalia ticket, Purchase Manager interfaces with the Transportation Companies API to obtain, in real time, all the necessary information.
Purchase Manager role is also to register into the DB an eventual pass bought by User externally from the system.

– *Payment Handler*: it handles the phase in which the User chooses the ticket and proceeds to the payment. It interfaces with the DB to have information about User's credit cards data and also with the Transportation Companies API, in order to forward the payment.

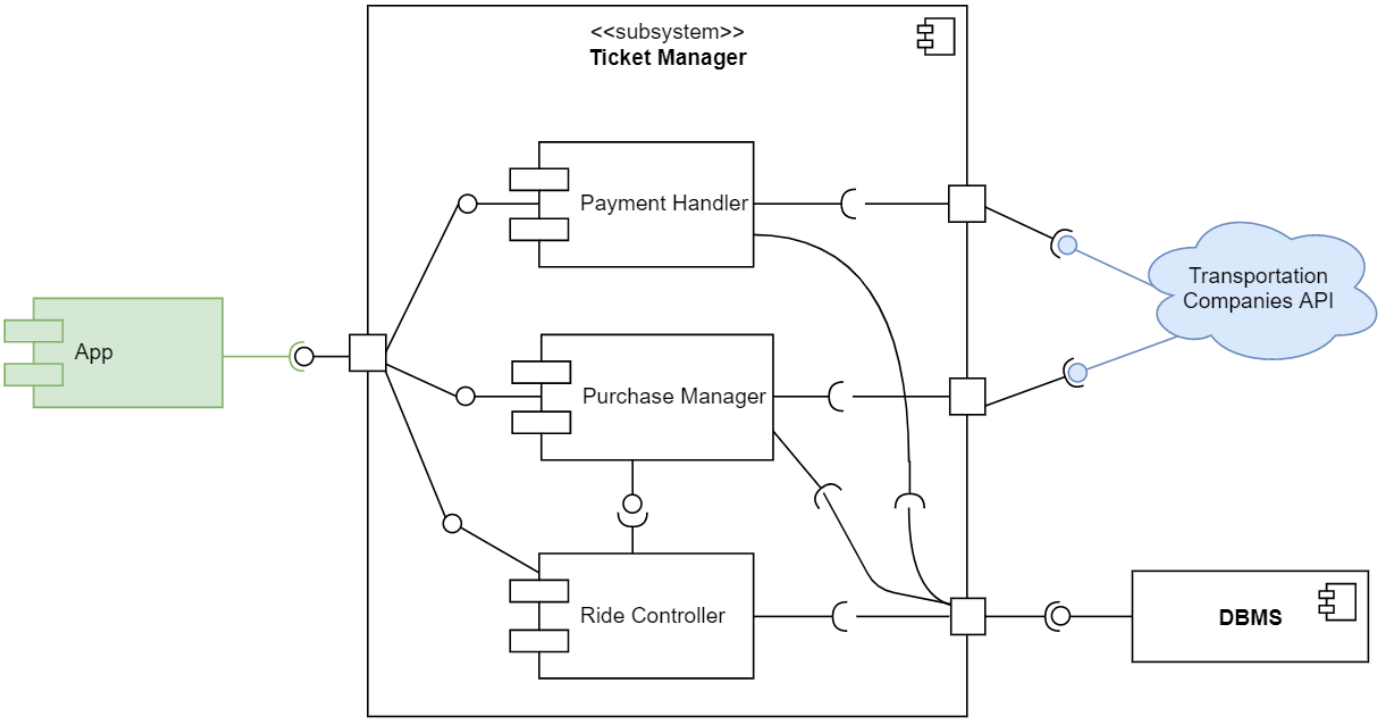

Figure 6: Component Diagram - Ticket Manager

2.3 Runtime view

We provide in this section a dynamic view of the system, illustrating the main interactions among components.
Five sequence diagrams are provided: Sign Up, Modify Settings, Invitation Creation, Select TravelPlan Solution, Locate Nearest Vehicle. These cover all the interesting interactions. The only interaction not analyzed is the one related to Appointment Creation: we chose to not include here a sequence diagram because we had provided in the RASD a detailed activity diagram. After that, we also provide an Object Diagram of Weather & Traffic Modules, as an explicative example of how instances of their components are related.

2.3.1 Sequence Diagram - Sign Up

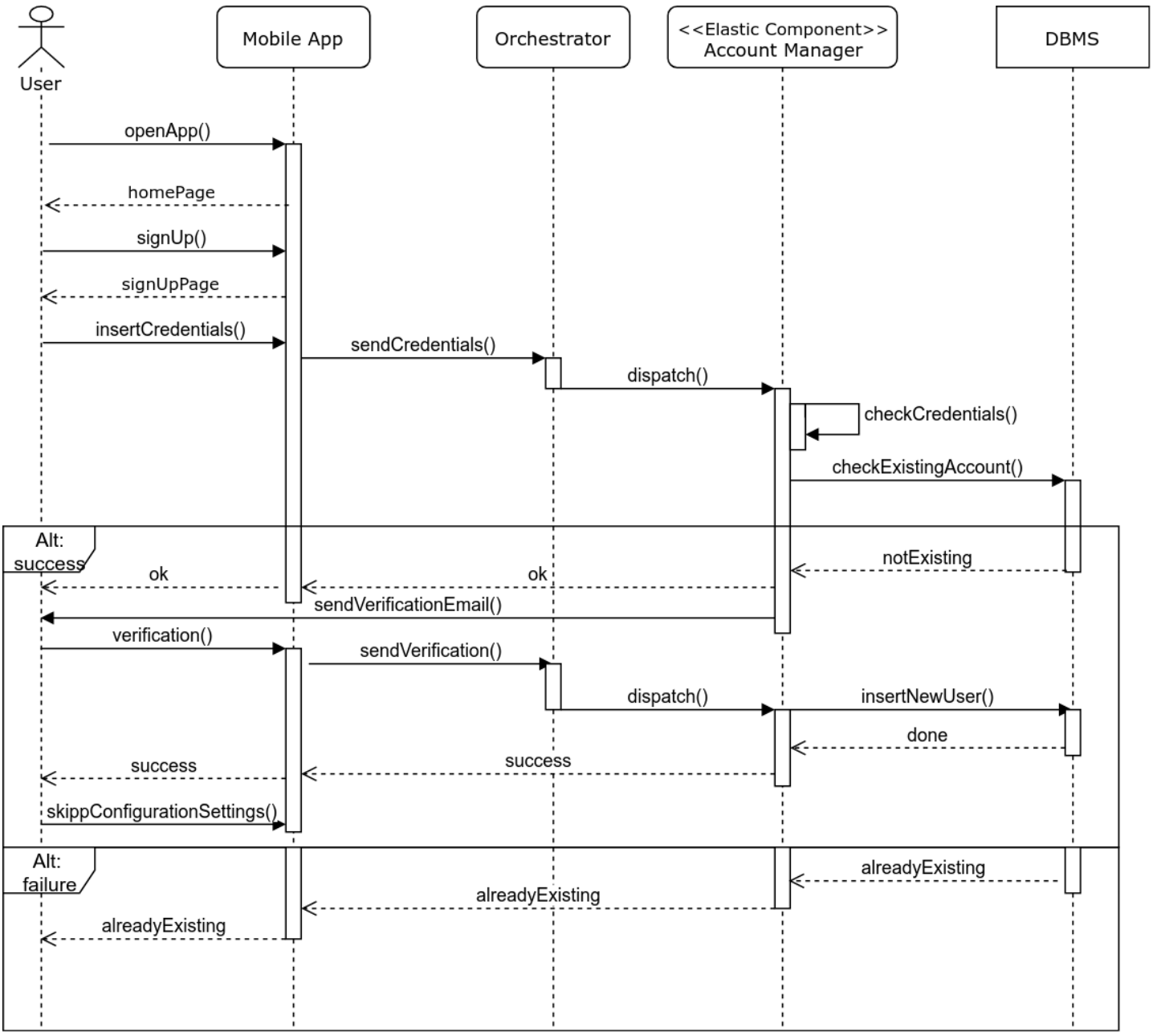

Figure 7: Sequence Diagram - SignUp

2.3.2 Sequence Diagram - Modify Settings

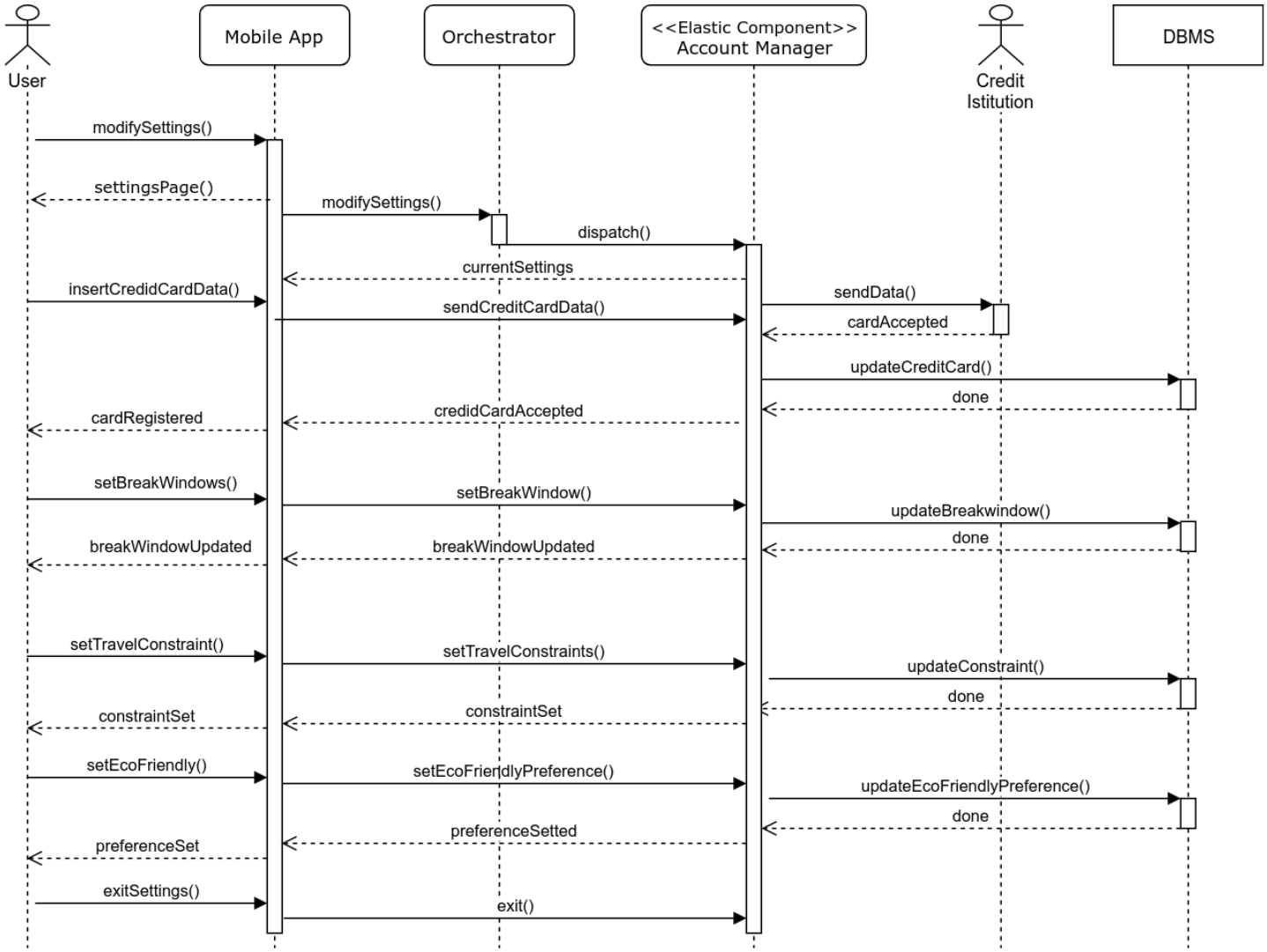

Figure 8: Sequence Diagram - Modify Settings

2.3.3 Sequence Diagram - Invitation Creation

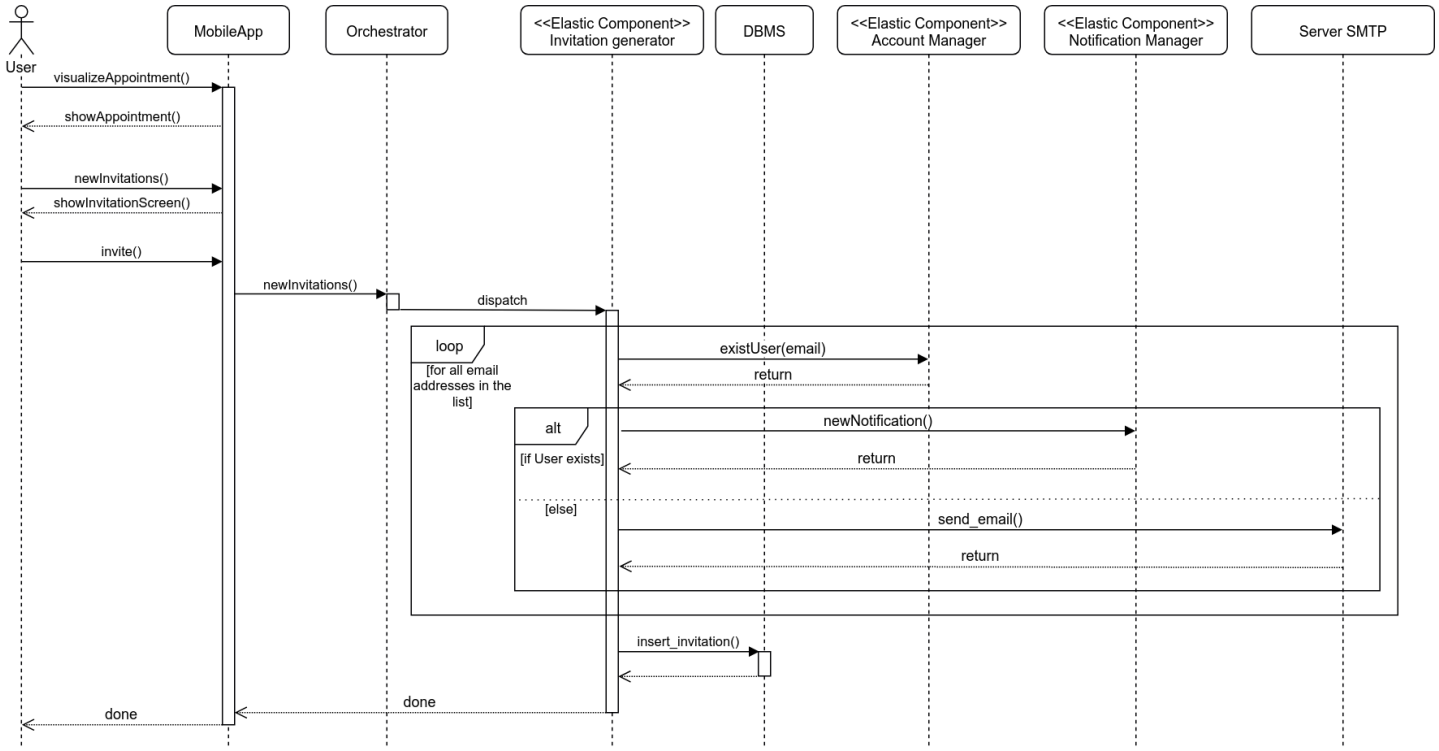

Figure 9: Sequence Diagram - Invitation Creation

2.3.4 Sequence Diagram - Select TravelPlan Solution

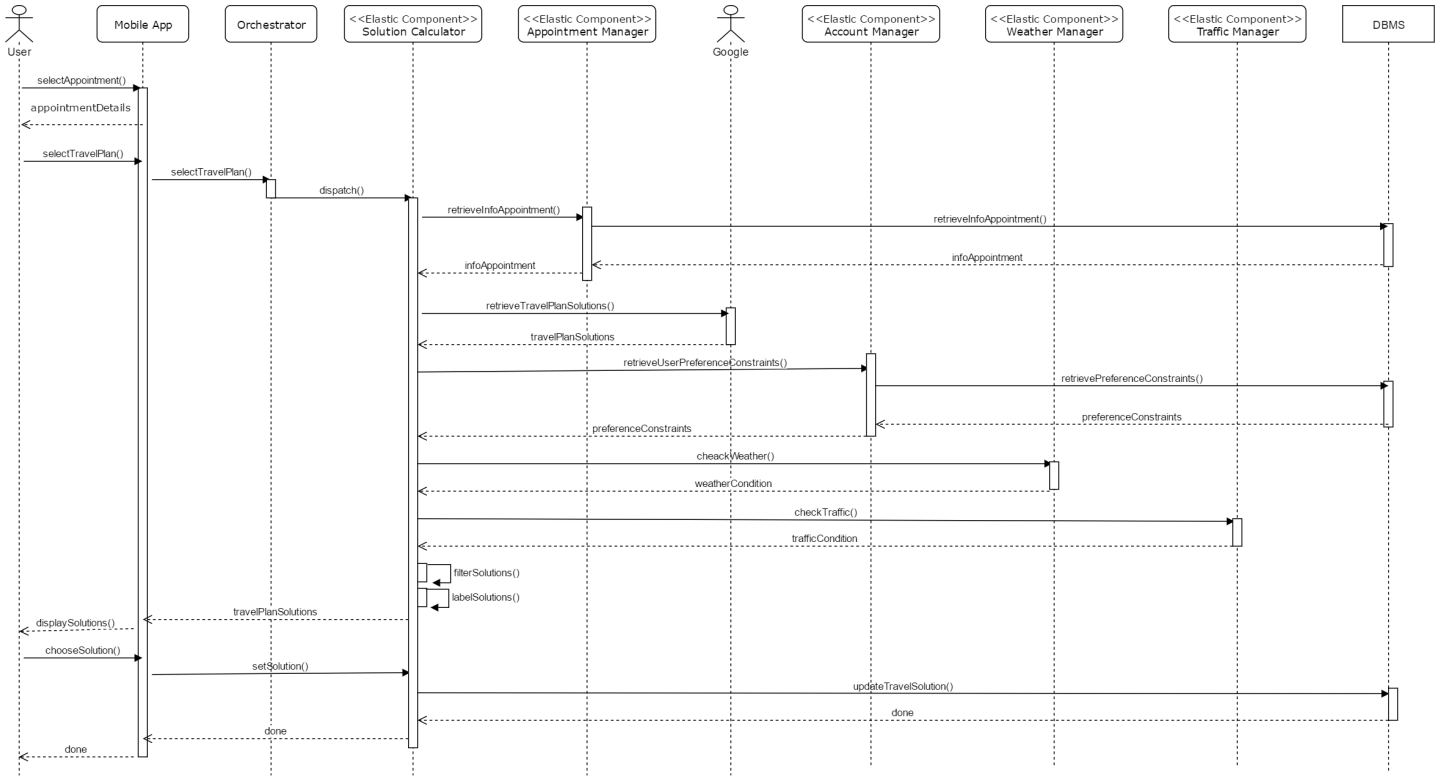

Figure 10: Sequence Diagram - Select TravelPlan Solutions

2.3.5 Sequence Diagram - Locate Nearest Vehicle

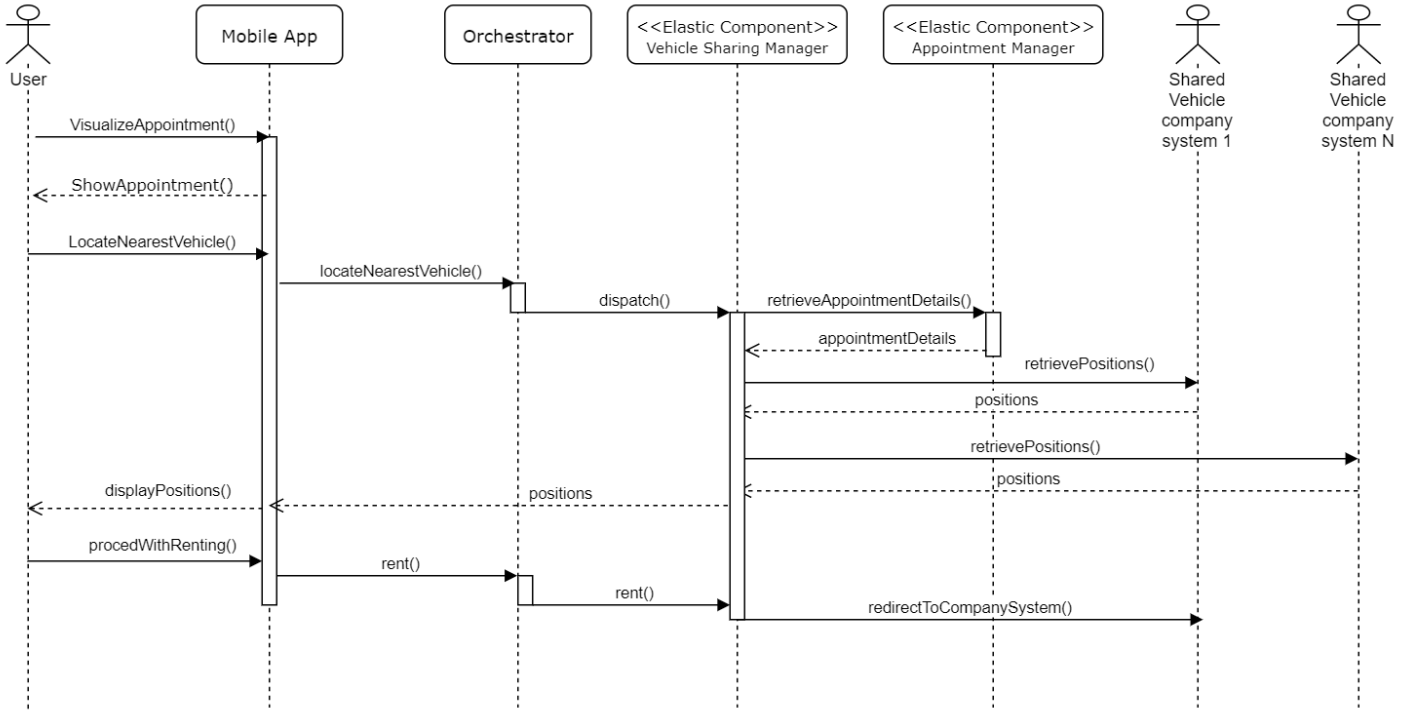

Figure 11: Sequence Diagram - Locate Nearest Vehicle

2.3.6 Object Diagram - Weather & Traffic Modules

As said in the component diagram description, this architecture is exactly the same used in the Traffic Module and so we provide here only the object diagram of one of them, the Weather Module, for consistency with the component diagram.
The two diagrams illustrate an evolution in the instances caused by a load balancing operation and an unexpected crash. Both diagrams are not complete of all the instances and we used dashed lines to represent the fact that some instances have been cut, to have simpler and easier to understand diagrams. In the first diagram, we can see two Querier, one for Italy and one for France. This partition of the region has been chosen by the System Administrator (another option could have been Europe). The Queriers are linked to the related Notifiers and to the Manager. The Manager keeps track of the Queriers, the active Notifiers and the Notifiers in the standby list.

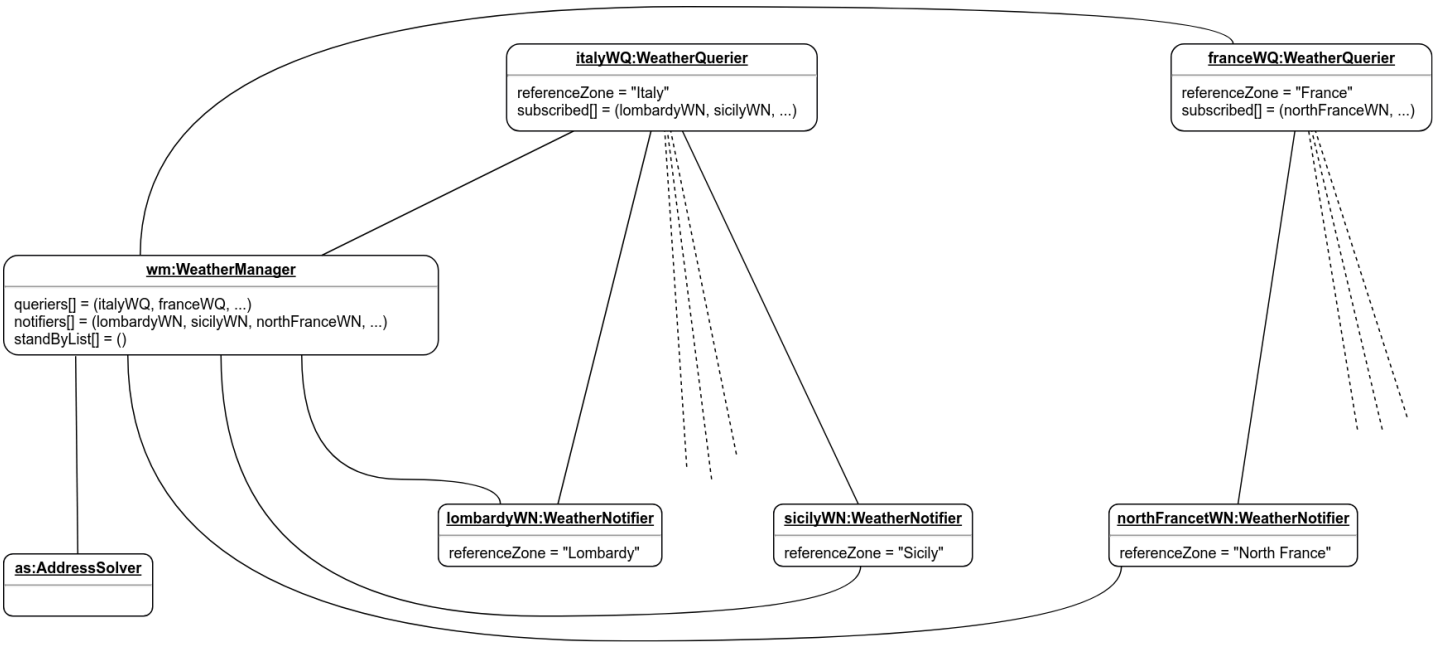

Figure 12: Object Diagram - Weather Module Original State

At some point, the Notifier related to Lombardy is under load and the load balancer split it into two Notifiers, respectively related to North Lombardy and South Lombardy. Their subscription to the Querier `italyWQ` is managed directly by the Manager as described in algorithm section (3.1).
The `franceWQ` Querier unexpectedly crashes and the Notifiers that were subscribed to it are put in the standby list by the Manager.
The next diagram represents the final situation.

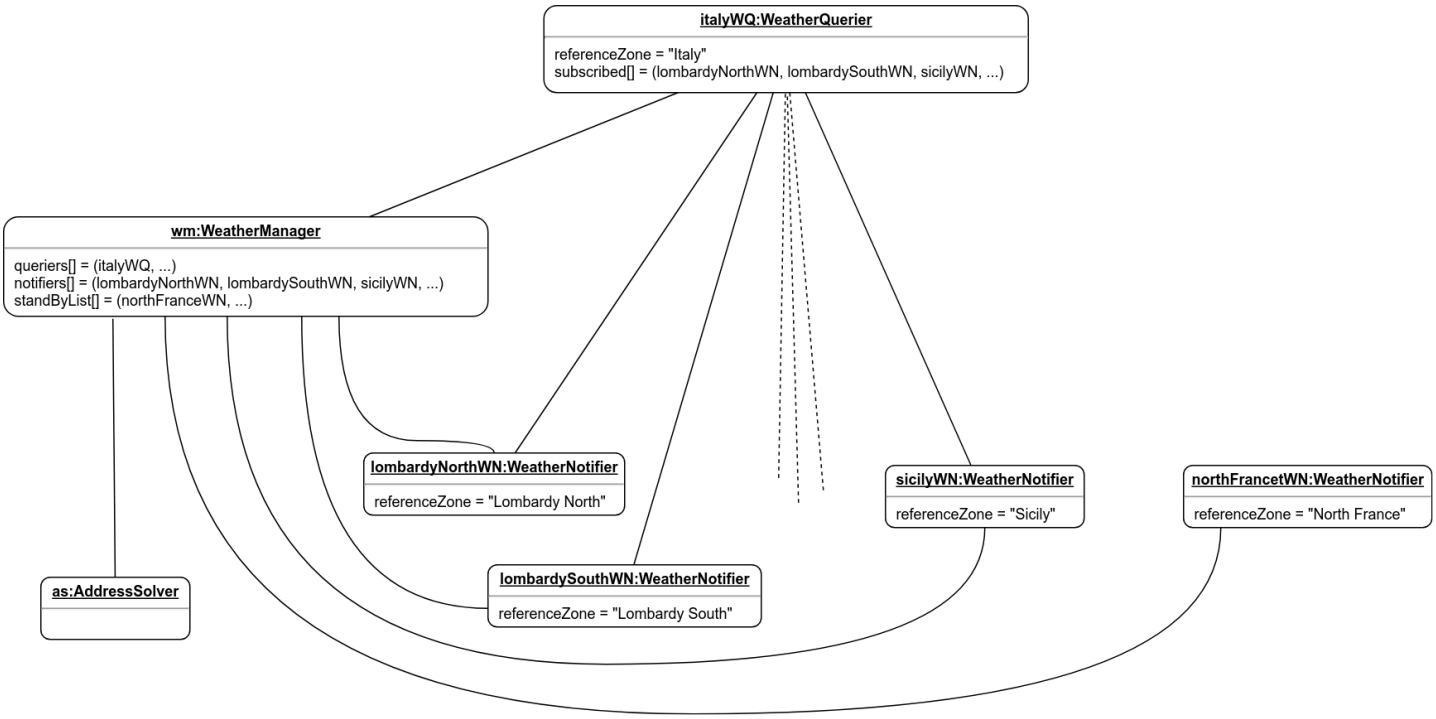

Figure 13: Object Diagram - Weather Module After Balancing and Reconfiguration

2.4 Component interfaces

We describe here most of the interfaces used and provided by the components defined in this section. The interfaces we do not describe here are considered as trivial and as so, assumed to not need any description. An example of interfaces not described are the one used in the Weather/Traffic modules, where they are mainly the standard interfaces of any publish subscribe system.
The User who asks for the service is never passed as parameter because it is meant to be retrieved as a session parameter.

Appointment Manager

AppointmentEditorInterface

- `void newAppointment(String name, String description, Date, Time, Location)`
- `void deleteAppointment(AppointmentId)`
- `void updateName(AppointmentId, String name)`
- `void updateDescription(AppointmentId, String description)`
- `void updateDate(AppointmentId, Date)`
- `void updateTime(AppointmentId, Time)`
- `void updateLocation(AppointmentId, Location)`
- `void setTravelPlan(AppointmentId, TravelPlan)`

AppointmentViewerInterface

- `Appointment[] getList(Date from, Date to)`
- `Appointment getAppointment(AppointmentId)`

AppointmentStorerInterface

- `void update(AppointmentId, Appointment)`
- `void deleteAppointment(AppointmentId)`
- `void newAppointment(AppointmentId)`

IncomingAppointmentSchedulerInterface

- `void schedule(Appointment)`

Notification Manager
This module makes use of the Data Interface, `Notificable`, that is an interface extended by all the objects that can be sent as Notification.

NotificationManagerInterface

- `void instantNotification(User, Notificable)`
- `void schedule(User, Notificable, time, date)`
- `void unschedule(NotificationId)`
- `void reschedule(NotificationId, time, date)`

NotificationDispatcherInt

- `void notify(User, Notificable)`

Invitation Manager

InvitationGeneratorInterface

- `void invite(Appointment, emails[])`
- `void removeInvitation(Appointment, emails)`
- `emails[] retrieveInvitations(Appointment)`

InvitationHandlerInterface

- `void accept(AppointmentId)`
- `void reject(AppointmentId)`
- `Invitation[] retrieveInvitations()`

Ticket Manager

PaymentHandlerInterface

- `void pay(Ticket)`

PurchaseManagerInternalInterface

- `Tickets[] retrieveTickets(Ride)`

PurchaseManagerExternalInterface

- `void storeTicket(Ticket)`

RideControllerInterface

- `Map(Ride,Tickets[]) getTickets(TravelPlan)`

2.5 Selected architectural styles and patterns

The following architectural styles have been used:

2.5.1 Client-server

The mobile is a client communicating directly with the application server. The browser supporting the web app is a client communicating with the web server. The application server behaves as a client querying the database server.

2.5.2 Service-Oriented Architecture (SOA)

The way clients interact with the application server is thought to be service-oriented. The single components are analyzed from a high-level point of view depending on the service they offer. SOA allows to easily extend the system by building and adding independent modules to the core.

2.5.3 Model-View-Controller (MVC)

MVC pattern is followed throughout the whole system design. The clients are front-end components *(views)* interacting with logic components *(controllers)* which drive the information flow and the information retrieval from the database *(model)*.

2.6 Other design decisions

2.6.1 Password Storage:

Users' passwords are not stored in plain text, but they are hashed and salted with strong cryptographic hash functions.

3 Algorithm Design

3.1 Weather and Traffic modules - Dynamic configuration

As we previously described the Weather and Traffic modules are subject to load balancing and (↑)*dynamic configuration* by the System Administrator. These two mechanisms cause four events to happen. How they are managed is described below:

– *a Notifier is deleted*: this is the only activity that is not managed by the Manager because there is no reason for doing it. If the Notifier is simply being closed, it detaches itself from the Querier. If the Notifier crashes or it is suddenly closed, the Querier will notice this when trying to notify it and it will detach the dead Notifier. No other actions are needed.

– *a Notifier is created*: the new Notifier communicates its zone to the Manager. The Manager will return to the Notifier a reference to the appropriate Querier using the Address Solver to interpret the zone of the Notifier. Thus, the Notifier can subscribe itself to the Querier.

– *Quierier is deleted*: all the Notifiers previously attached to the Querier have to be analyzed. If a less specific Querier exists *(see below)*, they are attached to it, otherwise they are put in a standby list (they will not receive any information about weather, or traffic)

– *a new Querier is instantiated*: the standby list is scanned searching for Notifier that can be attached to the Querier (matching the two zone through the Address Solver). If a less specific Querier exists *(see below)*, all the Notifier subscribed to it are analyzed and are eventually moved to the new Querier.

Meaning of specificity of a Querier
Let's assume, for instance, that there are four Querier: `ItalyQ`, `MilanQ`, `LazioQ`, `ParisQ`, respectively related to zones: Italy, Milan, Lazio, Paris. `MilanQ` and `LazioQ` are more specific with respect to `ItalyQ` because Milan and Lazio are inside the region Italy. On the other hand, `ItalyQ` is less specific with respect to `MilanQ` and `LazioQ`. `ParisQ` has no relation of specificity with all the others.
Hence:

- `LazioQ` crashes → all the Notifiers subscribed to `LazioQ` are now moved to `ItalyQ`
- `RomeQ` Querier is created → `ItalyQ` is less specific than `RomeQ`, so `ItalyQ` is scanned searching for Notifiers associated to region Rome: if there are such Notifiers, they are moved to `RomeQ`. Notice: the Notifiers that were associated to `LazioQ` but are actually outside the region of Rome would remain associated to `ItalyQ`.
- `ParisQ` crashes → all the Notifiers associated to `ParisQ` are moved to the standby list, because no 'less specific' Queriers are available.

Notice that Queriers are meant to be wide regions and not single cities, this was just an example.

3.2 Solution Calculator - How it works

Here it is illustrated the procedure by which the S2B provides the user with the travel solutions for the selected appointment. As we previously described the Solution Calculator takes care of handling the three-step articulated process: recovery of feasible travelPlanSolutions, filtering and labeling.

Recovery of feasible travelPlanSolutions:

- Solution Calculator, through Appointment Manager, retrieves appointment information (departure location, destination, time, presence of passengers, baggage, ..);
- based on this information, Solution Calculator interfaces with Google, which provides it with a list of possible travel solutions - travelPlanSolutions *(each TravelPlan solution consists of more Rides each with its own TravelMean, distance, time).*

Filtering

- Solution Calculator, through Account Manager, retrieves user's TravelMeanConstraints and Eco-Friendly preference (each TravelMean constraint refers to a particular Travel Mean specifying the distance beyond which it is not to be used or a time band to be avoided);
- Solution Calculator, through Weather Manager and Traffic Manager, retrieves information about weather and traffic conditions (this takes into account the reliability of this information so they are only considered if the appointment is an (↑)*Incoming Appointment*);
- for each TravelPlan in travelPlanSolutions, Solution Calculutor checks if there is a TravelPlan that does not meet the requirements, in particular:
  - if the user has specified the presence of passengers and the TravelMean of the Ride is not authorized for the carriage of passengers, TravelPlan is removed by travelPlanSolutions;
  - if the user has specified baggage and the Ride's TravelMean is not suitable for baggage's trasportation, TravelPlan is removed from travelPlanSolutions;
  - if Ride violates a TravelMeanConstraint (for example, the Ride has a TravelMean for which there is a TravelMeanConstraint that says TravelMean should not be used for a distance > x and Ride has a distance > x), TravelPlan is removed by travelPlanSolutions.

Labeling
After filtering solutions in travelPlanSolutions, Solution Calculator:

- calculates the cheapest TravelPlan: the one for which the sum of all the estimated prices of individual Rides in TravelPlan results to be lowest if compared with the other TravelPlan in travelPlanSolutions and sets a tag = "cheapest";
- calculates the fastest TravelPlan: the one for which the sum of all estimated Ride times of each Ride is the lowest if compared with the other TravelPlan in travelPlanSolutions and sets a tag = "fastest";
- calculates the most ecological TravelPlan: the one for which the sum of all EcoScores of each Ride is the highest if compared with the other TravelPlan in travelPlanSolutions and sets a tag = "Eco-Friendlest" (for this operation a function calculateEcoScore (TravelPlan) , detailed below, is used);
- for each TravelPlan where there is a Ride with a TravelMean is not recommended in case of adverse weather conditions (eg rain, snow, ..), if weather forecasts are bad, set a tag = "warning: Weather";
- for each TravelPlan where there is a Ride crossing an area where there is intense traffic, set a tag="warning: traffic".

Finally Solution Calculator sends TravelPlanSolutions to the user.

Calculate EcoScore:
Each TravelMean has associated a travelMeanEcoPenalty, more the TravelMean is 'Eco-Friendly', less the penalty is. 'The algorithm attributes to each TravelPlan an EcoScore considering all the Rides that belong to it. The maximum possible EcoScore is 0;

PseudoCode:

```
1       calculateEcoScore(TravelPlan tp){
2               score = 0;
3               for each Ride r in tp{
4                       score = score - r.travelMean.penalty * r.distance;
5               }
6               tp.Ecoscore = score;
7       }
```

4 User Interface Design

4.1 UX Diagrams

In this diagram we represent how the app and the web App should interact with the User. We chose to expand only the main Screens and to leave some methods unexplored, to avoid overcomplicating the diagram unnecessarily. The methods not explored are preceded by **.

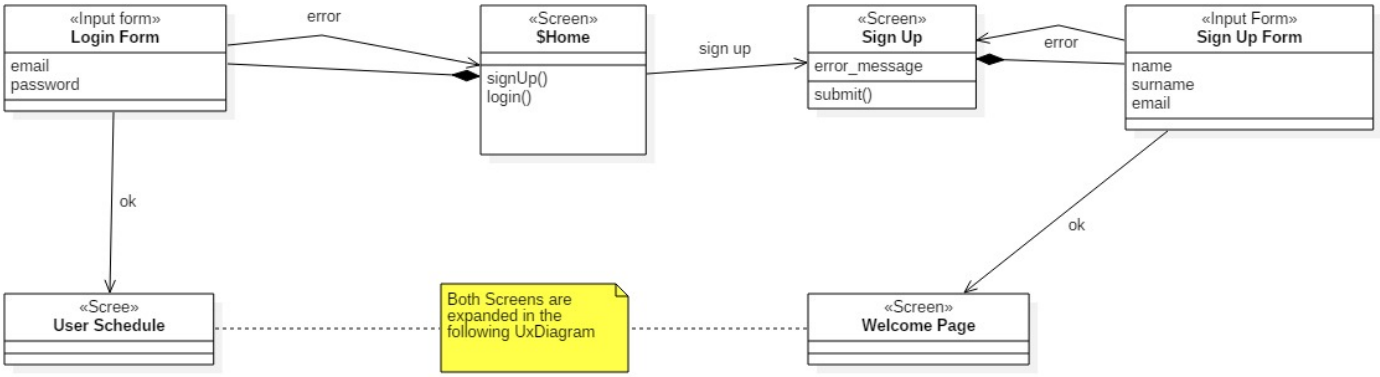

Figure 14: UX Diagram - Person UX

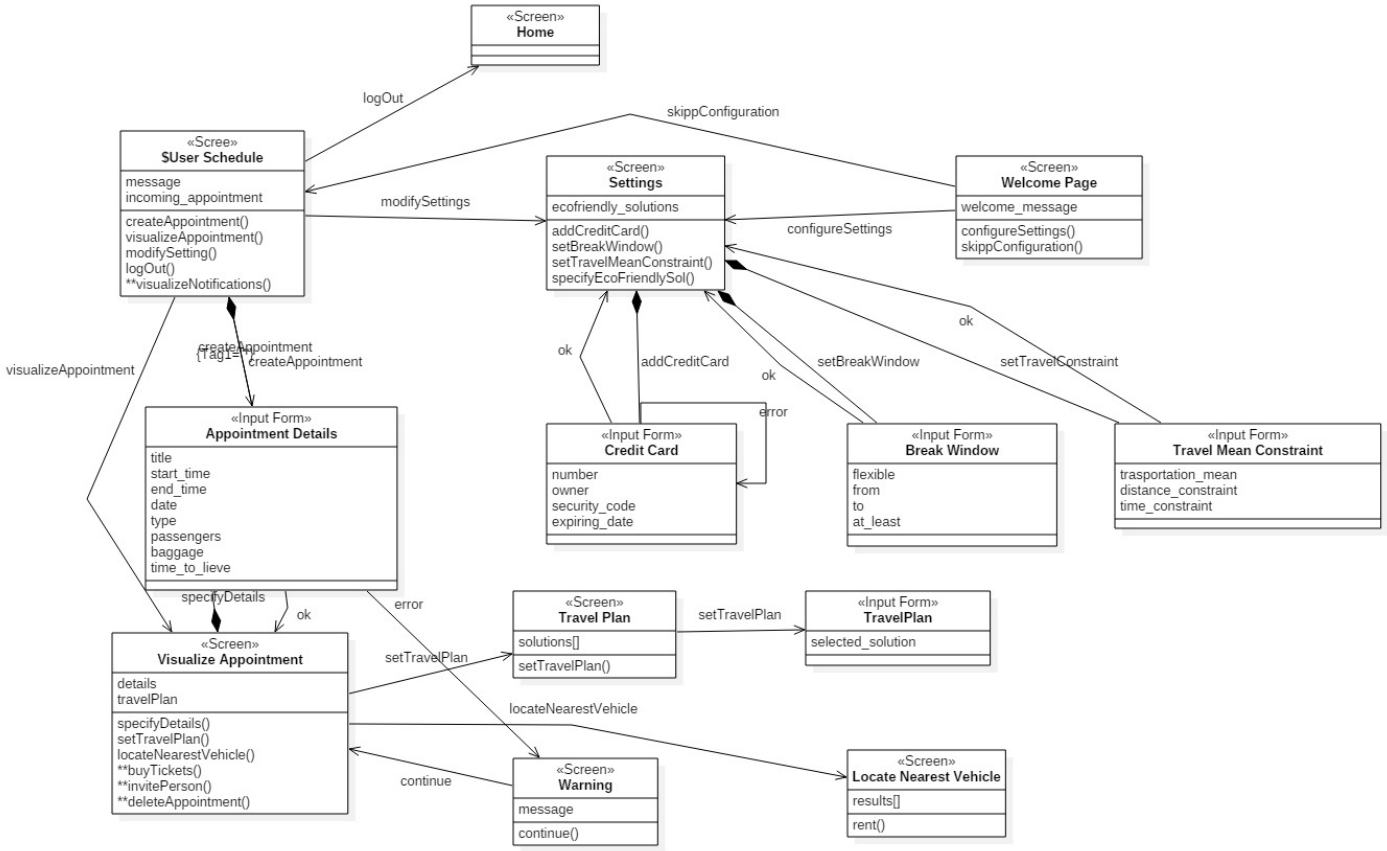

Figure 15: UX Diagram - User UX

4.2 App Mockups

Here we provide several mockups to highlight what we described in the UX Diagrams above. To logging in, a user has to insert his credentials in the Home Page. After that, he is redirected to his Schedule from where he can access all the functionalities with no more than three clicks.

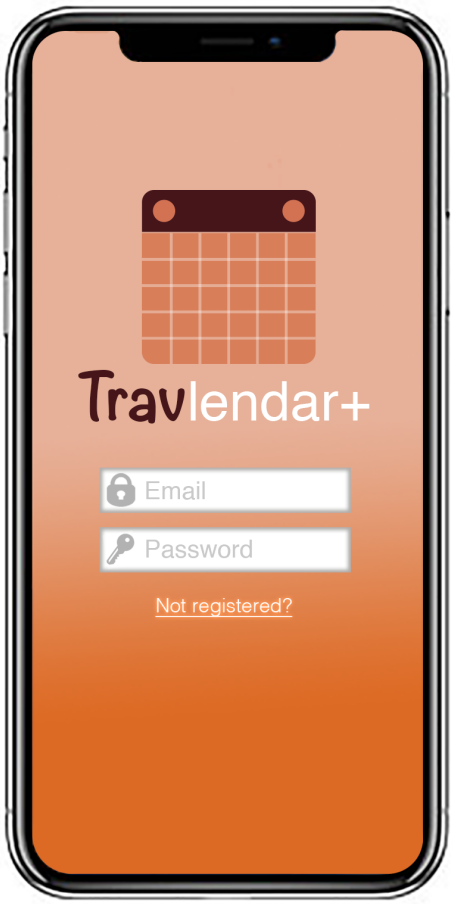

Figure 16: Mockup - Home Page

Figure 17: Mockup - User Schedule

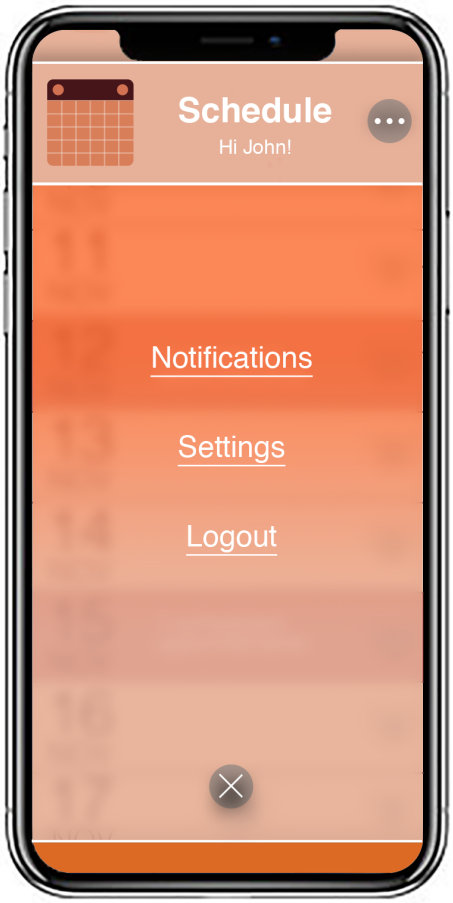

Figure 18: Mockup - Settings

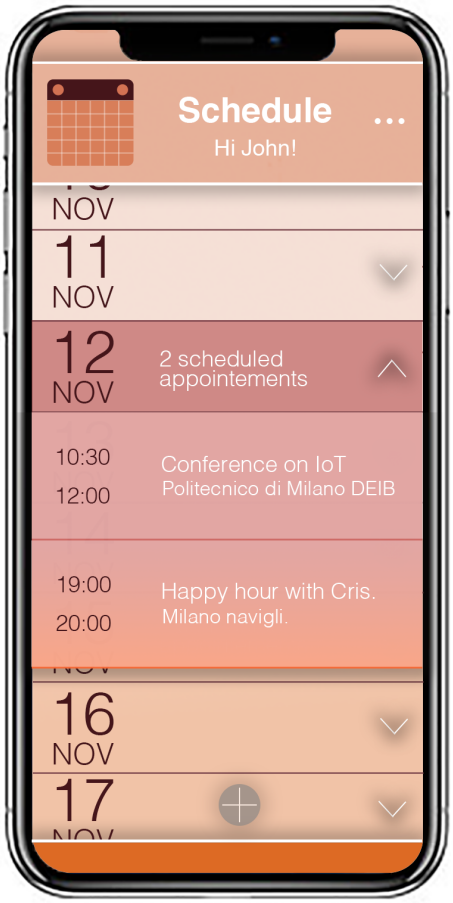

Figure 19: Mockup - Daily Schedule

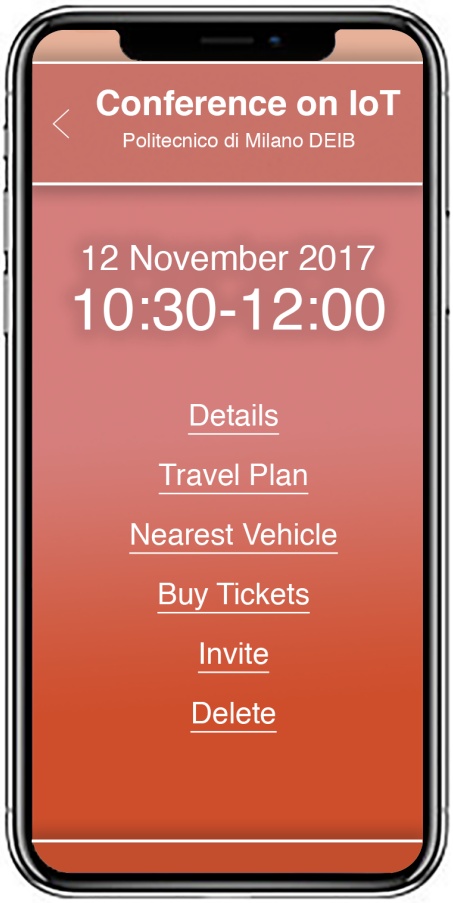

Figure 20: Mockup - Visualize Appointment

Figure 21: Mockup - Select TravelPlan

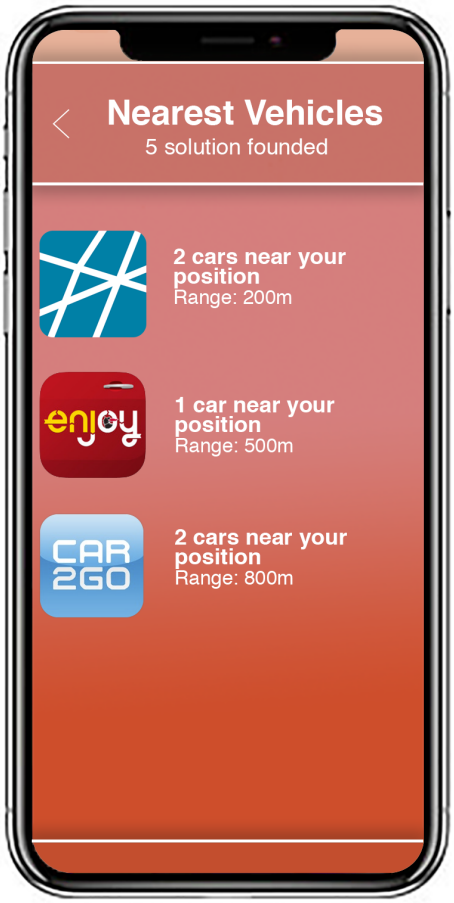

Figure 22: Mockup - Locate Nearest Vehicle

5 Requirements Traceability

Here we show the mapping between goals and requirements identified in section 3.3 of the RASD and the modules of the business logic described in the previous sections of this document. We list only the components of the actual logic, omitting the DBMS, the Orchestrator, the Mobile App and the Web App because they are omnipresent.

Goal G1: a Person should be able to have his/her own Travlendar+ agenda

→ **Requirements R1÷R6**

Components:

- Account Manager: as it can be seen in the Sign Up Sequence Diagram (2.4.1), it is the component in charge of the sign up process

Goal G2: a User should be able to customize the offered service

→ **Requirements R7, R8**

Components:

- Account Manager: it lets the user set preferences, constraints, break windows
- Solution Calculator: it takes into account the settings of the User when calculating the solutions

Goal G3: a User should be able to take note of all his/her appointments and their details

→ **Requirements R9, R10**

Components:

- Appointment Manager: it lets every User add new appointments

Goal G4: a User should be able to manage his/her appointments

→ **Requirements R11÷R16**

Components:

- Appointment Manager: it lets Users visualize their schedule and modify their appointments

Goal G5: for each appointment, the User should be assisted in the choice of the travel solution

→ **Requirements R17÷R24**

Components:

- Solution Calculator: it actually provides the available solutions
- Weather Module: it provides weather information to support the Solution Calculator computation
- Traffic Module: it provides traffic information to support the Solution Calculator computation
- Notification Manager: it notifies the User when events that can influence his/her choices occurs

Goal G6: a User should be able to invite other persons to his/her appointment

→ **Requirements R25, R26**

Components:

- Invitation Manager: it lets the User generate new invitation and lets other Users accept/refuse them
- Notification Manager: it sends notification to the User

Goal G7: a User is assisted in the purchase of a ticket when it is required

→ **Requirements R27÷R31**

Components:

- Account Manager: it lets the User register his credit card(s)
- Ticket Manager: it provides ticket suggestions and let the User buy new tickets

Goal G8: a User should be able to locate the nearest vehicle of a vehicle sharing system, if that is the transportation mean of choice of an (↑)*incoming appointment*

→ **Requirements R32**

Components:

- Appointment Manager: it classifies appointment as "incoming" when necessary and it provides appointment details
- Vehicle Sharing Manager: it is in charge of the localization feature

Goal G9: a User should be able to rent a shared vehicle, if that is the transportation mean of choice of an (↑)*incoming appointment*

→ **Requirements R33, R34**

Components:

- Appointment Manager: it classifies appointment as "incoming" when necessary and it provides appointment details
- Vehicle Sharing Manager: it is in charge of the redirection to the Shared Vehicle Company

Goal G10: a User should always be aware of the (↑)*incoming appointments* and how to reach them

→ **Requirements R35÷R37**

Components:

- Appointment Manager: it classifies appointment as "incoming" when necessary and it provides appointment details
- Solution Calculator: it computes the Solution to be, eventually, shown to User in the Notification
- Notification Manager: it sends the Notification to the User
- Weather Module: it provides updates on the weather to know good and bad travel solutions
- Traffic Module: it provides updates on the traffic to know good and bad travel solutions

6 Implementation, Integration and Test Plan

With the goal of identifying a proper developing plan, for each feature we have analyzed:

- the importance for Users: how much the feature is perceived as fundamental, from the User point of view. Core features have been labeled with a high or medium-high importance. They have to be implemented completely during the first releases. Features with importance low or medium-low are the ones that could be implemented in successive releases as additional features that add value to Travlendar+ but are not part of the core functionalities.
- difficulty of implementation of the Back End: it defines an estimate of the effort that will be spent in the implementation of the communication issues, the logic and the data models on the server required by the feature.
- difficulty of implementation of the Front End: this indicates an estimate of the effort required to implement in the App *(Mobile App or Web App)* the graphic and the user interactions

The following table summarizes the result of our analysis.

Feature	**Importance for User**	**Back End**	**Front End**
sign up	medium-low	low	low
appointment creation and management	high	medium-low	medium-low
schedule visualization	high	low	medium-high
settings	medium-low	low	medium-low
solutions calculation	high	medium-high	medium-high
notification system	medium-high	medium-high	low
weather & traffic updates	medium-high	high	low
tickets purchase and management	low	medium-high	medium-high
localization sharing vehicle	medium-low	medium-low	medium-low
rent a sharing vehicle	low	low	low
invitations	medium-low	medium-high	medium-low
Incoming Appointments	medium-high	medium-low	low

Verification and Validation process will begin as soon as the system starts to be implemented. For each component (and for each subcomponent belonging to it) there are two phases: the first one is an analysis concerning in a manual inspection by team members, the second one is an automatized test using tools support. We chose to make a manual inspection of the product because, in this way, we have a major formal level compared to a Walkthroughs analysis; in fact, inspection is an expensive technique, but it guarantees a low level of errors in the final product.
Below there is a list ordered (time increasing) by system features and not by components because, for the integration process, it has been chosen to adopt Thread strategy. In this way we can release to the customer more than one version of the product, from the version that contains core functionalities (all of high importance for the user) to the complete version, adding step by step new features. Thread strategy allows implementing also a part of a specific component. For instance, considering Solution Creator component, at the beginning it can be implemented, analyzed and tested without taking into account the possibility to specify settings (medium-low priority for the user). Setting features can be added to the system at a later time.

Components will be implemented, integrated and tested in the following order:

1. ***Appointment creation and management***: to satisfy this functionality, part of the Appointment Manager is implemented, analyzed and tested. In particular, only two of its subcomponents are implemented, Appointment Editor and Appointment Handler.

2. ***Solution calculation***: to satisfy this functionality, part of the Solution Calculator is implemented, analyzed and tested. Solution Calculator, at this point, interfaces with the implemented part of Appointment Manager and with Google API. It doesn't interface either with Account Manager (solutions are processed without taking into account user's settings) or with Weather and Traffic modules (solutions are calculated without considering bad weather and traffic).

3. ***Sign up***: to satisfy this functionality, part of Account Manager is implemented, analyzed and tested. In particular, the part that allows Users to sign up into the system in order to be recognized during future accesses.

4. ***Schedule visualization***: to satisfy this functionality, the implementation, analysis, and test of Appointment Manager continue. In it, there is not any additional component but, in this phase, we handle the graphics interface concerning the schedule visualization *(medium-high in front-end)*.
It is possible to make a first release containing core functionalities. From now on, except for point 6. below, each feature is a single module, so it can be released step by step.

5. ***Settings***: to satisfy this functionality, the implementation of Account Manager ends. The User has the possibility to insert preferences and to set constraints. From now on, Solution Calculator processes travel solution taking into account the settings that the user sets.

6. ***Notification system***: in this phase of the process, Notification Manager and his subcomponents (Notification Scheduler, Notification Dispatcher) are implemented. Note that, at this point, Notification Manager's utilizers (Incoming Appointment Scheduler, Weather and Traffic modules, Invitation Manager) are not present, so there are no actual notifications to send.

7. ***Incoming Appointments***: to satisfy this functionality, the implementation, analysis, and test of Appointment Manager end after adding its last subcomponent, Incoming Appointment Scheduler. Now Notification Manager also notifies incoming appointments.

8. ***Wheather & traffic updates***: to satisfy this functionality, Weather and Traffic modules are implemented, analyzed and tested completely. Now Solution Calculator processes travel solutions taking into account bad weather and traffic. The user receives related notifications.

9. ***Invitations***: to satisfy this functionality, Invitation Manager and related subcomponents are implemented, analyzed and tested. Now the user receives notifications concerning invitations.

10. ***Ticket purchase and management***: to satisfy this functionality, Ticket Manager and related subcomponents are implemented, analyzed and tested.

11. ***Rent a sharing vehicle***: to satisfy this functionality, part of Vehicle Sharing Manager is implemented, analyzed and tested.

12. ***Localization sharing vehicle***: to satisfy this functionality, implementation, analysis and test of Vehicle Sharing Manager ends.

During integration test process (and before its termination), the following must subsist:

- DB must be activated, it must contain an adequate quantity of data in order to test system's performances by overloading it. In this way, it is possible to test a big part of the components interfaced with the DBMS.

- Interactions with external actors must be present, both on the contractual and on the communicative level. In particular, at least two Transportation Companies, at least one Sharing Vehicle Company must interact with the system. This must subsist before Ticket Purchase and Management functionality will be completely implemented. In this way, after their implementation, Ticket Manager and related subcomponents can be tested (with more attention on Purchase Manager) as soon as the implementation will be terminated. Google must also be interfaced with the system before completing the implementation of Weather and Traffic update functionality. In this way we can test Weather and Traffic modules.

- Payment service must be activated and it has to concern affiliated Transportation Companies. In this way, there is the possibility to test Ticket Manager and its subcomponents (e.g. Payment Handler).

We chose to implement, analyze and test at first, in a parallel way, Server and Mobile App. The Web App development will start as soon as the implementation and the test of all high and medium-high priority (for the user) features will be completed.

7 Effort Spent and Team Work

Trying to save time, in our sessions we tried to concentrate only on organizing our work, comparing our ideas, and discussing important issues and decisions that had to be taken as a team. We worked together 14 hours.
Below, for each component of the team, we provide his list of main responsibilities and his hours of work indicated as total and as spent working alone. Please note that part of the work of each person has been dedicated to cross checking the work of the others.

Cassarino Pietro produced Purpose and Scope of the Introduction section. He created ER diagram and Ticket Manager component diagram developing the related descriptions. He also produced Implementation, Integration and Test Plan section.
Total hours: 46 hours, **Hours working alone:** 32 hours

Salaris Mirko organized the work on LaTeX delineating the document structure and improving it later on. He produced most of the components diagrams and the Sequence Diagram related to the Invitation Creation. All the work related to Weather and Traffic modules has been done by him. He was even in charge of the Requirements Traceability section and of the Component Interfaces section.
Total hours: 45 hours, **Hours working alone:** 31 hours

Ventrella Piervincenzo has designed high-level architecture and client-server interaction highlighting it in several diagrams, most of which are Sequence Diagrams. He also cared for the Solution Calculator algorithm design. Other minor sections were compiled by him. Finally he produced the UX diagrams and mockups to define the interaction between the user and the mobile app.
Total hours: 36 hours, **Hours working alone:** 22 hours

8 References

8.1 Software and Tools

- LaTeX for typesetting document
- TeXstudio as LaTeX IDE
- GitHub and git for version control and team work
- StarUML for UX diagrams
- Draw.io for UML diagrams
- Microsoft Visio for ER diagram
- Adobe Photoshop for Mockups

8.2 Reference Documents

- 1016-2009 - IEEE Standard for Information Technology – Systems Design – Software Design Descriptions: http://ieeexplore.ieee.org/document/5167255/
- 42010-2011 - ISO/IEC/IEEE Systems and software engineering – Architecture description: http://ieeexplore.ieee.org/document/6129467/