POLITECNICO
MILANO 1863

MSc Computer Science
and Engineering

Software Engineering 2
ACADEMIC YEAR 2017-2018

Design Document

894135
Franklin Onwu
franklinchinedu.onwu@mail.polimi.it

899318
Ivan Sanzeni
ivan.sanzeni@mail.polimi.it

884021
Matteo Vantadori
matteo.vantadori@mail.polimi.it

Related professor:
Prof. Matteo Giovanni Rossi

Release Date: November 26th 2017
Version 2.0

Contents

1 Introduction

1.1 Purpose

Section [G.1] treats all the goals related to the creation and customization of a new event:

G.1.1 The user can schedule a new event adding name, time slot, location, type and description.

G.1.2 The user can modify the name of the event.

G.1.3 The user can modify the location of the event.

G.1.4 The user can modify the description of the event.

G.1.5 The user can modify the starting time of the event.

G.1.6 The user can modify the ending time of the event.

G.1.7 The user can modify the event type, from a work event to a personal event or vice versa.

G.1.8 The user can modify the description.

G.1.9 The user can choose how many minutes earlier arrive to the event location.

G.1.10 The user can delete an already existing event.

G.1.11 The user can see all the events he/she has already scheduled.

Section [G.2] treats all the goals related to the customization of the user preferences:

G.2.1 The user can find the quickest way to reach an event.

G.2.2 The user can find the cheapest way to reach an event.

G.2.3 The user can find the most ecological way to reach an event.

G.2.4 In adverse weather conditions, the user can find the best way to reach an event only using means that keep him/her protected.

G.2.5 The user can find the best way to reach an event imposing constraints to the time slots designated to any mean.

G.2.6 The user can find the best way to reach an event imposing constraints to the maximum distance covered by any mean.

G.2.7 The user can find the best way to reach an event with a maximum chosen budget.

G.2.8 The user can find the best way to reach an event only using chosen means.

Section [G.3] treats all the goals related to the customization of the user settings:

G.3.1 A user with disabilities can find the best way to reach an event according to his/her needs.

Section [G.4] treats all goals related to the purchase of *non-shared transports*:

G.4.1 The user can book a taxi.

G.4.2 The user can book a limousine.

Section [G.5] treats all the goals related to the purchase of *public transports*:

G.5.1 The user can buy a metro ticket.

G.5.2 The user can buy a bus ticket.

G.5.3 The user can buy a trolleybus ticket.

G.5.4 The user can buy a tram ticket.

G.5.5 user can buy a train ticket.

Section [G.6] treats all the goals related to the purchase of *shared transports*:

G.6.1 The user can take a bike from a bike sharing service.

G.6.2 The user can take a car from a car sharing service.

1.2 Scope

Travlendar+ is a calendar-based application designed to schelude any kind of event, supporting the user in reaching the location of the events all across Milan, combining different sort of means in relation to the user preferences.

The application is designed to match the user needs to personalize each event in every respect. So the user can easly customize each event assigning it a category and distinguishing it between work or personal reasons, deciding means and constraints to reach it and buying tickets or booking means in-app, if necessary.

The main application goal is to lead the user to handle each kind of event with *Travlendar+*: from a lunch with friends to a job interview, from an interesting expo to an out of town meeting.

1.3 Definitions, acronyms, abbreviations

1.3.1 Definitions

Cheap = with this preference the application chooses the cheapest way to reach the location.

Eco = with this preference the application chooses the most ecological way to reach the location.

Flexible event = kind of event that provides calendar, reminder and street direction supports and can be overlapped with activities as long as exists a minimum amount of time fixed by the user.

Lasting event = kind of event that provides calendar, reminder and street direction supports and can be overlapped with activities.

Non-shared transports = limousine, taxi.

Not wet = with this preferences the application chooses only means that keeps the user out of adverse weather conditions to reach the location.

Personal event = the user specifies that the event has personal purposes.

Public transports = bus, metro, train, tram, trolleybus.

Quick = with this preference the application chooses the quickest way to reach the location.

Shared transports = bike sharing, car sharing.

Standard event = kind of event that provides calendar, reminder and street direction supports and cannot be overlapped with other activities.

Transfer event = kind of event that provides calendar and reminder supports and cannot be overlapped with other activities. It is used for events that take place outside Milan.

Travlendar+ = the name of the application.

Travlender = a registered and logged user of Travlendar+.

Work event = the user specifies that the event has work purposes.

1.3.2 Acronyms

API = Application Programming Interface.

DD = Design Document.

GPS = Global Positioning System.

MMS = Mapping Managing System.

MVC = Model-View-Controller.

TMS = Transporting Managing System.

1.3.3 Abbreviations

G.n.m = Goal number m in section n.

R.n.m = Requirement number m in section n.

1.4 Revision history

26th November 2017
Version 1.0 - Document delivery.

28th November 2017
Version 1.1 - Added the document structure, the component interfaces diagram and a complete integration plan, fixed some typing errors.

5th January 2018
Version 2.0 - Turned the JSP server into a JEE server in the deployment diagram, changed the responses tag in the event diagrams, added the dispatcher algorithm.

1.5 Reference documents

https://standards.ieee.org/findstds/standard
IEEE standard for requirements documents.

https://developers.google.com/maps
Reference point for the third-party *MMS* considered in this project.

https://citymapper.com/milano
Reference point for the third-party *TMS* considered in this project.

https://developers.facebook.com/docs/facebook-login/android
Reference point for the Facebook login.

https://developers.google.com/identity/sign-in/android
Reference point for the Google+ login.

DD Sample from A.Y. 2015-2016.pdf
DD document example from Software Engineering 2 directory, on BEEP.

1.6 Document structure

In the following sections we will introduce the application design. The document is subdivided in other six parts, besides the introduction:

Architectural design
A detailed description of *Travlendar+*, that includes a list of the application components, a model of the physical application deployment, the sequence diagrams of the most significant cases and a list of the used patterns.

Algorithm design
A Java-code example of the most important *Travlendar+* algorithms.

User interface design
A further development of the user interface showed in the *Requirements Analysis and Specification Document*.

Requirements Traceability
A list of all the components necessary to achieve each requirement described in the *Requirements Analysis and Specification Document*.

Implementation, integration and test plan
The proposal for a possible implementation plan for all the application components, dealing with their integration and test pashes.

Effort spent
A complete table of all the hours spent by the team during the project.

2 | Architectural design

2.1 Overview

Travlendar+ is designed to be exclusively a smartphone application, this implies that the user can have access to the application server only via his/her device.

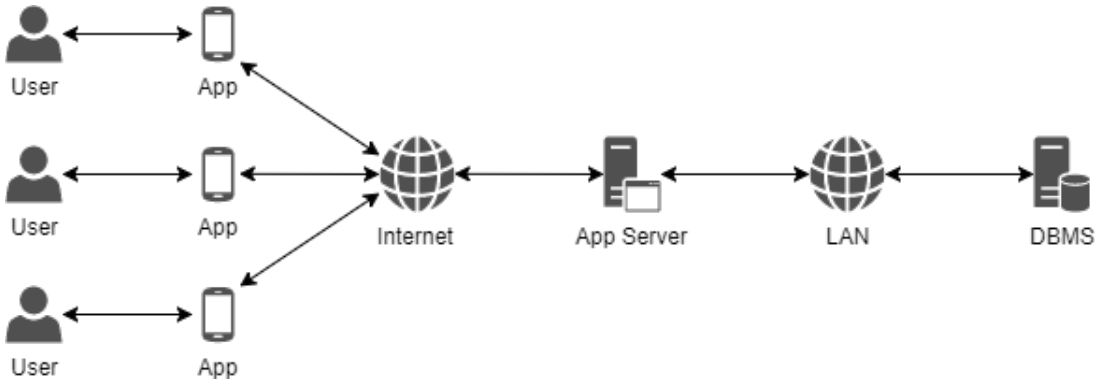

Figure 2.1: High-level architecture

For this reason, *Travlendar+* is meant to be a three-tier application, divided as follows:

Presentation tier includes the application itself on the user smartphone, it represents the means by which the user interacts with the remote system.

Application tier contains the application model and logic, it's the layer that makes decisions and evaluations on the best way to travel, following the user preferences.

Data tier stores all the data neccessary to the system.

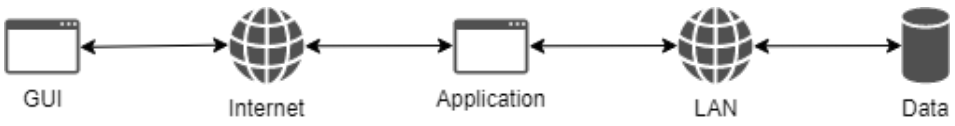

Figure 2.2: Three-tier architecture

2.2 Component view

The following diagram shows in deatil the cooperation between each component of the *Device*, the *Application Server* and the *Database Server*. The diagram also highlights the adopted patterns and the way that the application server interacts wtih the *Transporting Managing System* and the *Mapping Managing System*.

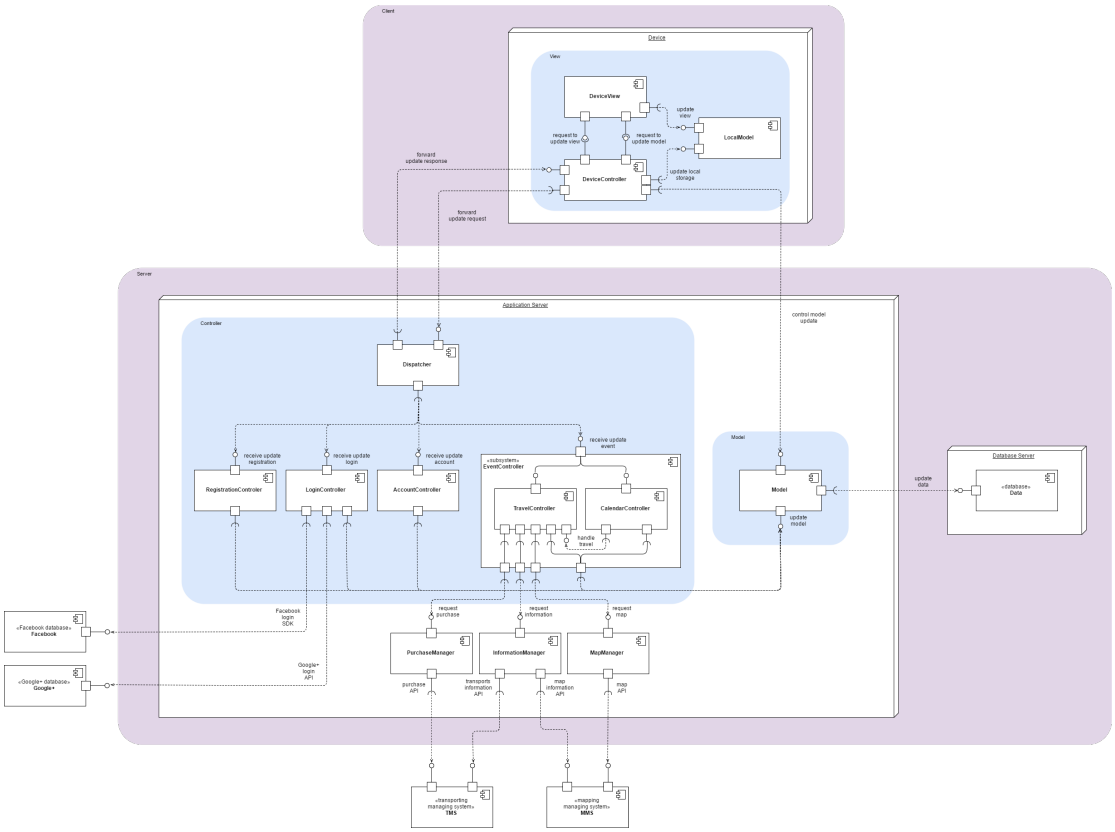

Figure 2.3: Component diagram

2.2.1 Device

The *Device* node has three components that simulate a local *MVC* pattern:

DeviceView represents the input/output application entity, it includes all the things the user can see and interact with.

DeviceController is the bridge connecting the view to the *Model*, it plays two important roles:

1. it forwards any request to update the *Model* from the *DeviceView* to the *Application Server*;
2. once the *Model* has been updated on the *Application Server*, it updates the *LocalModel* in the same way.

LocalModel is a local copy of the *Model*, its implementation allows the user to check the schedule and add a new event also if he/she is offline (it should be noticed that any update will be effective only once the user will be back online).

2.2.2 Application Server

The *Application Server* node has nine components (one of them divided into two subcomponents), it comunicates with the *Device*, the *Database Server* and the third-party applications:

Dispatcher is the *Application Server* component in charge of dispatching each request coming from the device to the relevant component.

RegistrationController handles any registration request.

LoginController handles any login request, it is also in charge of communicating with Facebook and Google+ to allow the user to log by means of them.

AccountController handles any request about user preferences (travel priority, not wet mode), constraints (maximum range of time, maximum distance, maximum amount of money, selected means, disabilities) and owned means.

EventController handles any request about the creation of a new event or the update of an existing one, it is designed as the closed cooperation between two subcomponents:

- **TravelController** is the spatial *EventController* subcomponent in charge of evaluating the best way to travel on the bases of the user preferences, constraints and owned means.
- **CalendarController** is the temporal *EventController* subcomponent in charge of evaluating if the event can be located in the time slot indicated by the user, taking account not only of the event duration, but also of the travel duration calculated by the *TravelController*.

PurchaseManager controls the non-shared (taxi, limousine) and shared booking (bike sharing, car sharing) and the public transports (bus, metro, train, tram, trolleybus) tickets purchase, thanks to its interaction with the *Transporting Managing System*.

InformationManager collects any information required to evaluate the best way to travel for each event, thanks to the combined interaction between:

1. the *Transporting Managing System* to know the public transports tickets cost, the fuel price, the taxi and limousine fares, the bike sharing and car sharing systems coordinates and the average cars fuel consumption.
2. the *Mapping Managing System* to know the weather conditions, the road traffic and to locate the bike sharing and car sharing systems coordinates provided by the *Transporting Managing System*.

MapManager provides the map that requests to the *Mapping Managing System*.

Model stores all the application data updated by the *Controller* components.

2.2.3 Database Server

The *Database Server* has only a single component:

Data stores all the user data updated by the *Controller* components, passing through the *Model*.

2.2.4 External Services

Transporting Managing System provides the public transports tickets cost, the fuel price, the taxi and limousine fares, the bike sharing and car sharing systems coordinates and the average cars fuel consumption.

Mapping Managing System provides the weather conditions, the road traffic and the bike sharing and car sharing systems location.

Facebook allows the user to sign in to *Travlendar+* using the Facebook login.

Google+ allows the user to sign in to *Travlendar+* using the Google+ login.

2.3 Deployment view

The following diagram shows in detail the physical deployment of the *Device*, the *Application Server* and the *Database Server*. The diagram also highlights the adopted patterns and the way that the application server interacts with the *Transporting Managing System* and the *Mapping Managing System*. The diagram also highlights the three-tier system architecture.

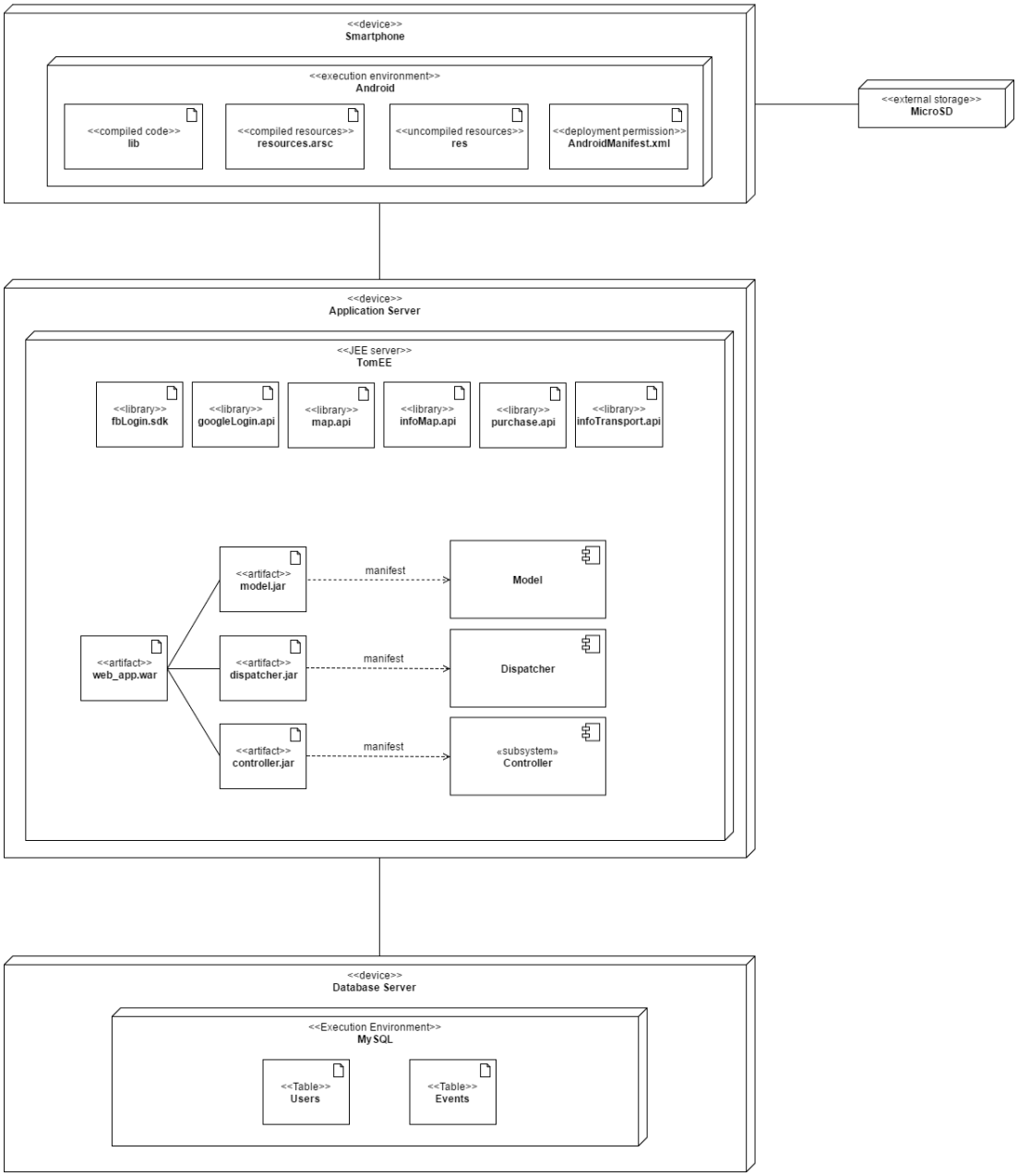

Figure 2.4: Deployment diagram

2.3.1 Device

The *Device* execution environment is *Android*, so the application has to be an *APK file*. The block shows the most significant files or directories:

lib contains the compiled code.

resourches.arsc contains the precompiled resources.

res contains the resources not compiled into *resources.arsc*.

AndroidManifest.xml describes the name, the version, the access rights and the referenced library files for the application.

2.3.2 Application Server

The *Application Server* JEE server is *TomEE*. The block shows the most important libraries and archives:

fbLogin.sdk allows the Facebook login.

googleLogin.api allows the Google+ login.

map.api provides the map.

infoMap.api provides the weather conditions, the road traffic and the bike sharing and car sharing services location.

purchase.api provides the purchase services.

infoTransport.api provides the public transports tickets cost, the fuel price, the taxi and limousine fares, the bike sharing and car sharing services coordinates and the average cars fuel consumption.

web app.war is a collection of *JAR file* that contains:

model.jar is the artifact that manifests the *Model* component.

dispatcher.jar is the artifact that manifests the *Dispatcher* component.

controller.jar is the artifact that manifests all the *Controller* components.

2.3.3 Database Server

The *Database Server* execution environment is *MySQL*. The block shows the most important tables:

Users in which are gathered the user preferences, constraints and owned means.

Events in which are gathered all the user events information.

2.4 Runtime view

2.4.1 Creation of a standard event or a flexible event

The sequence starts when the *Travlender* taps the + button to create a new event, choosing a standard event or a flexible one. In the Device, the *DeviceView* sends a request to the *DeviceController*, passing to it the event information. Then, the *DeviceController* dispatches the request to the Application Server *Dispatcher*, that requests a new event creation to the *EventController*, that in turn requests an event list to the *Model*. Now, the *EventController* takes the first test to check whether there is at least an overlap between the new event and an old one. If the test comes back negative, the *EventController* requests the map to the *MapManager* and a set of necessary information to the *InformationManager*, then it takes the second test to check whether there is at least an overlap between the travel time evaluated by the algorithm and the previous saved event. Worst-case scenario, if the algorithm doesn't find any acceptable travel, the application warns the *Travlender* and suggests him/her to modify the event time slot. On the contrary, if the test comes back negative, the *EventController* updates the *Model*, that also forwards the update to the *Data*. Now, the *DeviceController* can finally update the *LocalModel* and send to the *DeviceView* the request to update the view.

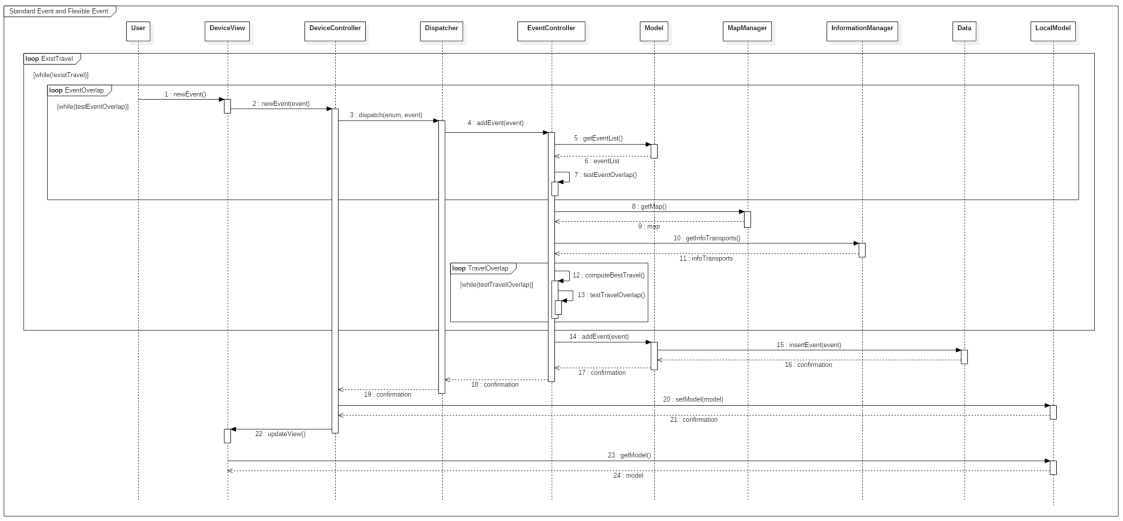

Figure 2.5: Standard event or flexible event

2.4.2 Creation of a lasting event

In this case too, the sequence starts when the *Travlender* taps the + button to create a new event, choosing this time to create a lasting event. In the Device, the *DeviceView* sends a request to the *DeviceController*, passing to it the event information. Then, the *DeviceController* dispatches the request to the Application Server *Dispatcher*, that requests a new event creation to the *EventController*, that in turn requests an event list to the *Model*, as in the previous case. However, this time the *EventController* doesn't take any test on the event time slot, because a lasting event allows its creation regardless of the overlap with other events. So, the *EventController* requests immediately the map to the *MapManager* and a set of necessary information to the *InformationManager* and updates the *Model*, that forwards the update to the *Data*. As before, the *DeviceController* can finally update the *LocalModel* and send to the *DeviceView* the request to update the view.

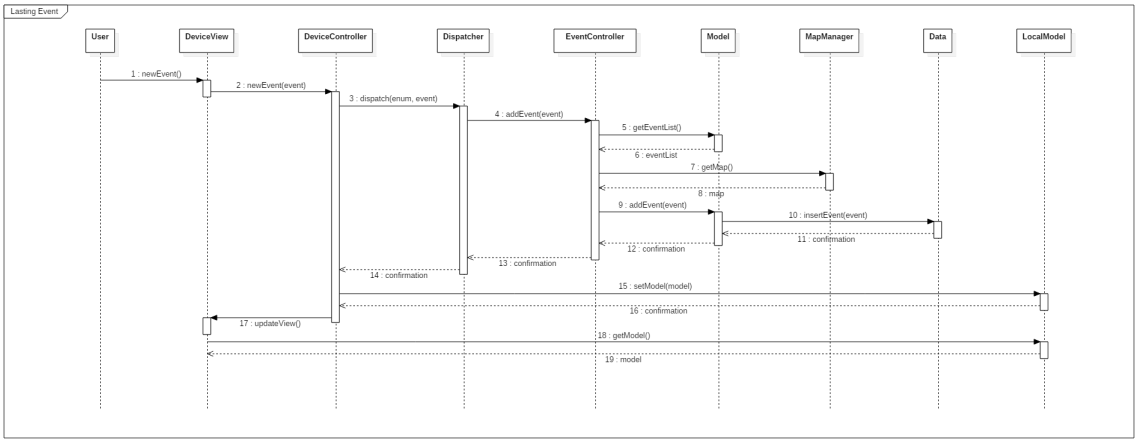

Figure 2.6: Lasting event

2.4.3 Creation of a transfer event

As the two previous cases, the sequence starts when the *Travlender* taps the + button to create a new event, choosing to create a transfer event. In the Device, the *DeviceView* sends a request to the *DeviceController*, passing to it the event information. Then, the *DeviceController* dispatches the request to the Application Server *Dispatcher*, that requests a new event creation to the *EventController*, that in turn requests an event list to the *Model*. Since the transfer event is designed only for events outside Milan, the *EventController* takes only one test, to check whether there is at least an overlap between the new transfer event and an old transfer, standard or flexible event (in the third case, it's important to be careful about its constraints). If the test comes back negative, the *EventController* requests the map to the *MapManager* and a set of necessary information to the *InformationManager*. Then, the *EventController* updates the *Model*, that also forwards the update to the *Data*. Now, the *DeviceController* can finally update the *LocalModel* and send to the *DeviceView* the request to update the view.

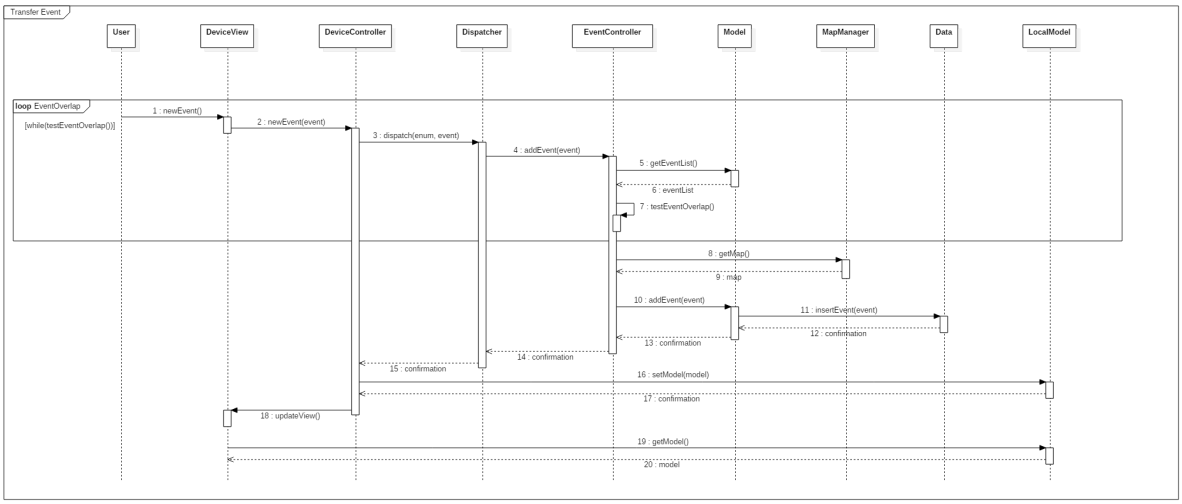

Figure 2.7: Transfer event

2.4.4 Start a travel and purchase a ticket

The sequence starts when the *Travlender* taps the *Go* button to reach an existing event. In the Device, the *DeviceView* sends a request to the *DeviceController*, passing the information on the upcoming event that the *Travlender* wants to reach. The *DeviceController* dispatches the request to the Application Server *Dispatcher*, that requests to start the travel to the *EventController*, that in turn requests the map to the *MapManager*, a set of necessary information to the *InformationManager* and all the user preferences, owned vehicles and constraints to the *Model*, that forwards the request to the *Data*. Now, the *EventController* evaluates the best way to reach the event, taking account of all the acquired information, and checks whether the travel time overlaps part of the event (on the off chance that, for any reason, a way to reach in time the event doesn't exist anymore, the application will warn the user of the delay time). Then, the *EventController* updates the *Model*, that also forwards the update to the *Data*. Now, the *DeviceController* can update the *LocalModel* and send to the *DeviceView* the request to update the view.
Once the *Travlender* starts the travel, he/she can decide to buy in-app a public transport ticket, if required. In the event that the *Travlender* decides to do that, the *DeviceView* sends a request to the *DeviceController*, passing all the user purchase information. The *DeviceController* dispatches the request to the Application Server *Dispatcher*, that requests to purchase a ticket to the*EventController*. The *EventController* forwards the purchase request to the *PurchaseManager*, that sends on success a confirmation message. So, the *EventController* can update the *Model*, that also forwards the update to the *Data*. In this case too, the *DeviceController* updates the *LocalModel* and sends to the *DeviceView* the request to update the view.

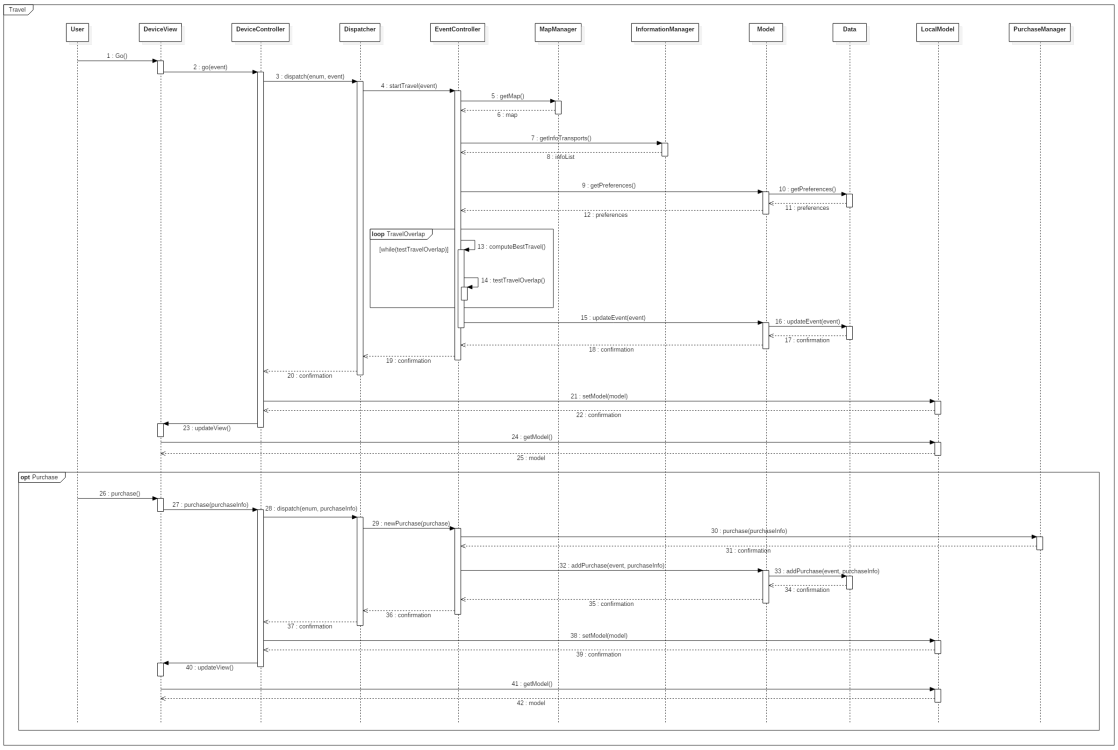

Figure 2.8: Travel start and ticket purchase

2.5 Component interfaces

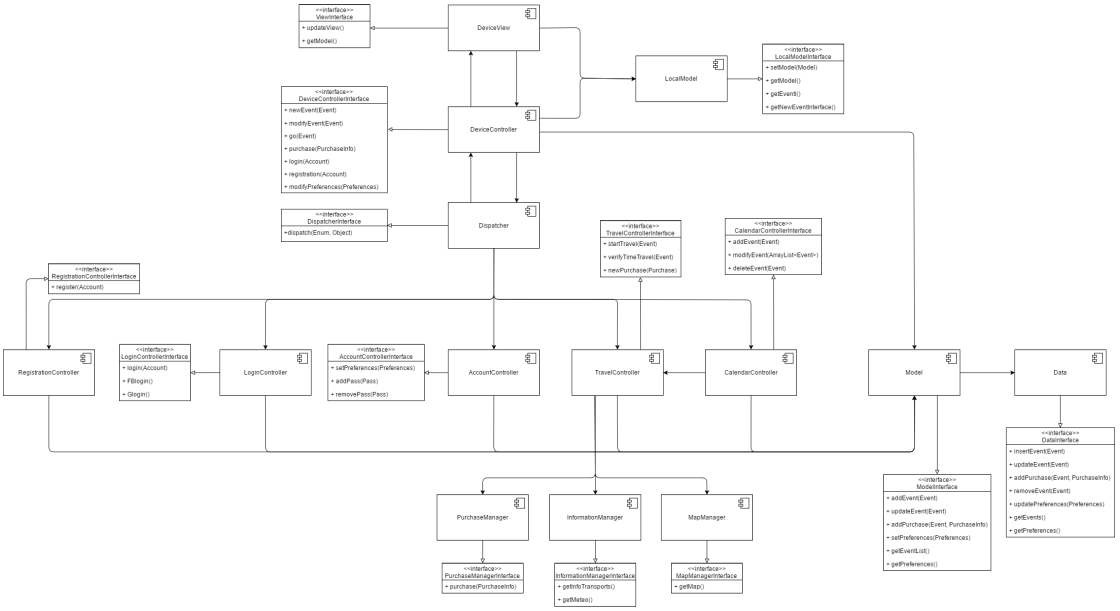

Figure 2.9: Component interfaces diagram

2.6 Selected architectural styles and patterns

2.6.1 Architectural styles

Travlendar+ is designed to use one architectural style:

Client-Server
As highlighted in purple in the component diagram [Figure 2.3], the applitication relies on a close communication beetwen two different hardwares:

1. the Device on which *Travlendar+* is installed (the *client*);
2. the Application Server and the Database Server, that provide the *Travlendar+* services (the *server*).

2.6.2 Patterns

Travlendar+ also is designed to use three different patterns:

Model-View-Controller
As highlighted in cyan in the component diagram [Figure 2.3], the application is divided into three sections:

1. a *view*, that corresponds to the device and sends all the update requests to the *Dispatcher* and displays the *Model*;
2. a *controller*, that groups the *Dispatcher*, the *Registration Controller*, the *LoginController*, the *AccountController* and the *EventController*, recieves the *DeviceController* requests and manipulates the *Model*;
3. a *model*, represented by the namesake component.

Local Model-View-Controller

The device is also divided into three components, that simulate a local *MVC* pattern, through which the *Travlender* can use some application services also if he/she is offline (obviously, any update will be effective only once he/she will be back online):

1. a *local view*, represented by the *DeviceView*, that displays the *LocalModel*.
2. a *local controller*, represented by the *DeviceController*, that recieves the requests from the *DeviceView* and manipulates the *LocalModel* after the *Model* and *Data* update.
3. a *local model*, represented by the namesake component, that is a local storage of the Application Server *Model*.

Dispatcher

The addition of the *Dispatcher* component permits to carefully monitor all the controllers. The *Dispatcher* is responsible for the request dispatching to the concerned controllers. Obviously, the *Dispatcher* needs a list of all of them.

3 | Algorithm design

3.1 Cheapest travel algorithm

This Java-code method convey a sense of how the cheapest travel algorithm will be implemented in the application:

Input: a set of travels.

Output: the cheapest travel from the set.

```
//cheapest way method
public Travel minimizeCost (Arraylist<Travel> travels){
   travels[0] = min;
   for(i = 1; i <= travels.size(); i++){
      if(travels[i].totalCost < min.totalCost){
         min = travels.get(i);
      }
   }
   return min;
}
```

3.2 Most ecological travel algorithm

This Java-code method convey a sense of how the most ecological travel algorithm will be implemented in the application:

Input: a set of travels.

Output: the most ecological travel from the set.

```
//most ecological way method
public Travel minimizeCFP (Arraylist<Travel> travels){
   travels[0] = min;
   for(i = 1; i <= travels.size(); i++){
      if(travels[i].totalCFP < min.totalCFP){
         min = travels.get(i);
      }
   }
   return min;
}
```

3.3 Quickest travel algorithm

This Java-code method convey a sense of how the quickest travel algorithm will be implemented in the application:

Input: a set of travels.

Output: the quickest travel from the set.

```
//quickest way method
public Travel minimizeTime (Arraylist<Travel> travels){
   travels[0] = min;
   for(i = 1; i <= travels.size(); i++){
      if(travels[i].totalDistance < min.totalDistance){
         min = travels.get(i);
      }
   }
   return min;
}
```

3.4 Dispatcher algorithm

This Java-code method convey a sense of how the dispatcher algorithm will be implemented in the application:

Input: the action number and an object.

Output: none.

```
//dispatcher
public void dispatch(EnumActionType action, Object parameter){
   switch (action){
      case DELETE_EVENT:
         calendarController.deleteEvent((Event) parameter);
      break;
      case ADD_EVENT:
         calendarController.addEvent((Event) parameter);
      break;
      case MODIFY_EVENT:
         calendarController.modifyEvent((ArraList<Event>) parameter);
      break;
      case LOGIN:
         loginController.login((Account) parameter);
      break;
      case REGISTRATION:
         registrationController.register((Account) paramenter);
      break;
      case MODIFY_PREFERENCES:
         accountController.setPreferences((Preferences) parameter);
      break;
      case START_TRAVEL:
         travelController.startTravel((Event) parameter);
      break;
   }
}
```

4 | User interface design

4.1 User interface

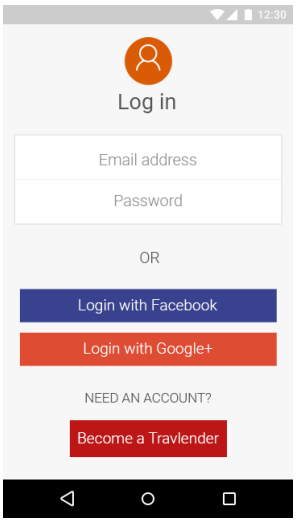

Figure 4.1: Login

The unregistered or unlogged user can log in to his/her account (if he/she has already got one) or create a new one. The application allows to log in with his/her own Facebook or Google+ account.

Tapping *Become a Travlender* an unregistered user can sign in to *Travlendar+*. A registration form appears and the user can fill out the form and tap *Send* to complete his/her registration.

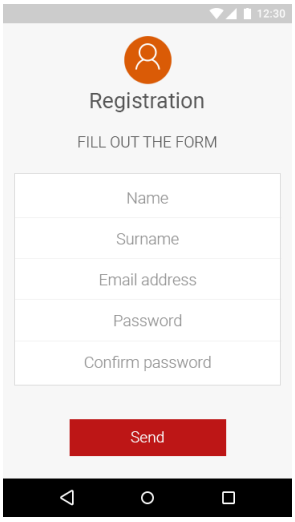

Figure 4.2: Registration

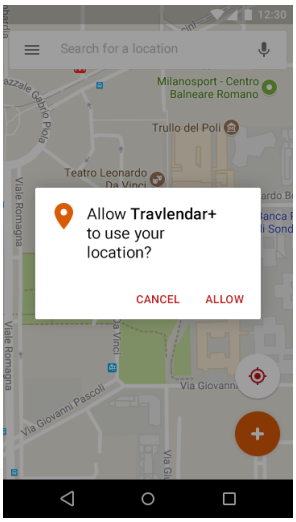

Figure 4.3: Alert

The first time the *Travlender* starts the application a pop-up appears, in which *Travlendar+* asks for the permission to use the user location. The user can tap on *allow* to give it, or *cancel* to refuse. In the second case, many application functions won't be accessible.

If the *Travlender* decides to use the map view, he can see two buttons on the right. The first one centers the map on his/her position, the second one allows him/her to schedule a new event.

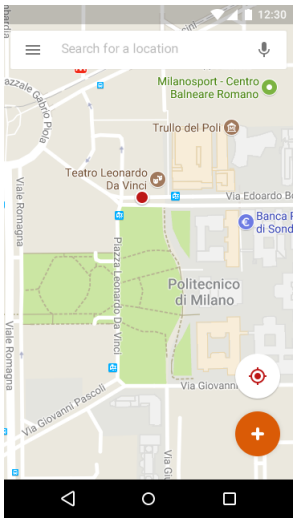

Figure 4.4: Map

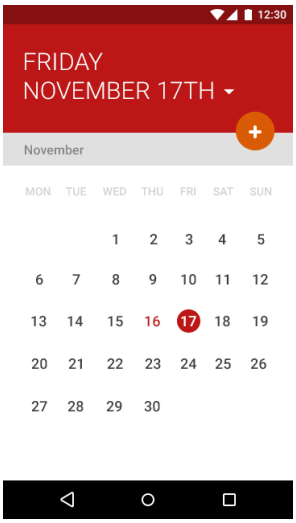

Figure 4.5: Calendar

If the *Travlender* decides to use the calendar view instead, he can see only one button on the right that allows him/her to schedule a new event. The *Travlender* can go to the previous or next month swiping on the left or on the right, respectively.

When the *Travlender* decides to create a new event, he/she can tap the + button on the map and a pop-up appears, in which *Travlendar+* asks for the event category.

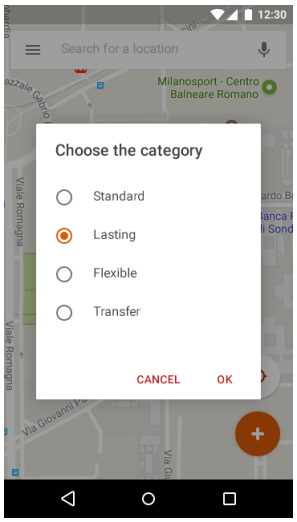

Figure 4.6: Choise from the map

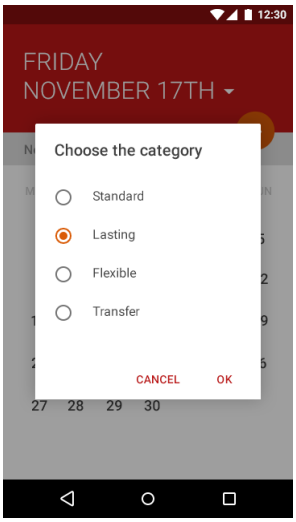

Figure 4.7: Choise from the calendar

The *Travlender* can also tap the same button from the calendar. Also in this case the same pop-up appears.

Now a new screen appears, in which the user can insert the location of the event, its name, a description, the starting and ending date and time and can modify the default settings, tapping the four buttons on the bottom.

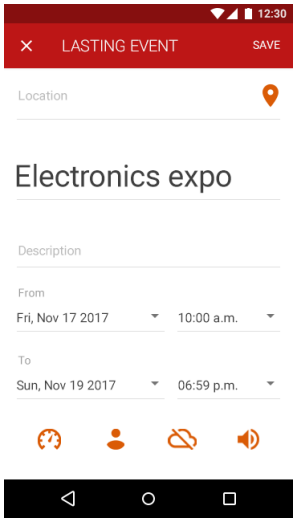

Figure 4.8: Creation of a new event

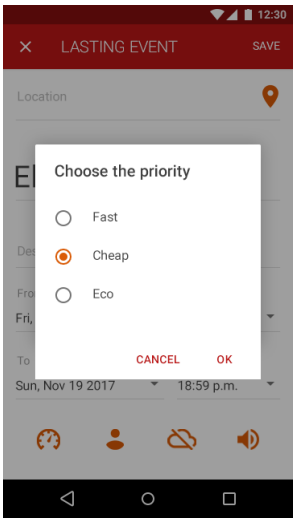

Figure 4.9: Priority choise

The first one permits to change the travel preference from *quick* to *cheap*, or *eco*.

The second one permits to change the event type from *work* to *personal*.

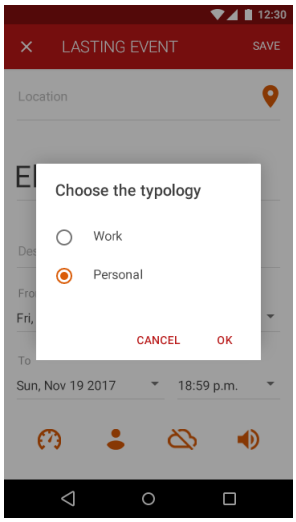

Figure 4.10: Typology choise

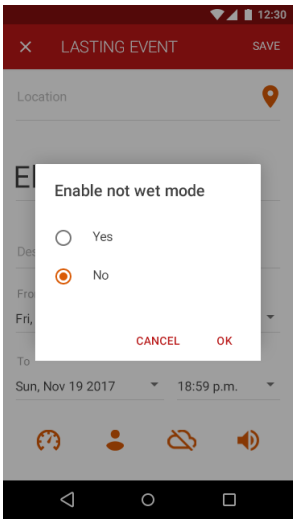

Figure 4.11: Not wet mode status

The third one enables (or disables) the *not wet* travel.

The last one enables (or disables) *notifications*.

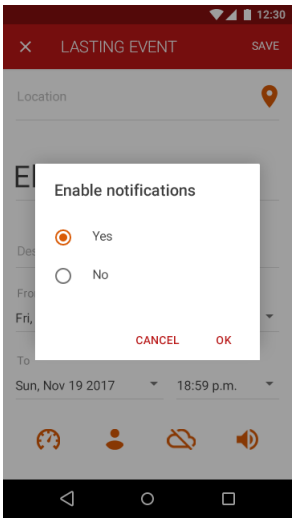

Figure 4.12: Notifications status

Figure 4.13: Location edit

Tapping the top-right button, the map appears and the *Travlender* can choose the location straight from it.

With the left swipe, the *Travlender* can access the scheduled events view, from the menu.

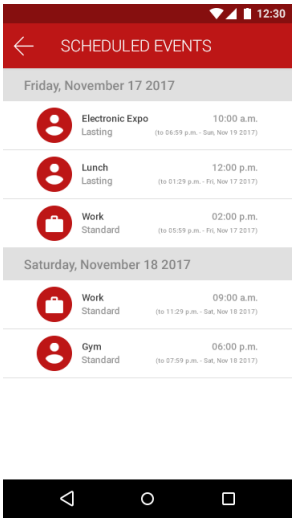

Figure 4.14: List of scheduled events

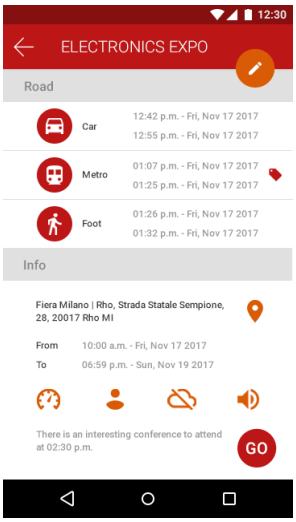

Figure 4.15: Summary of the event

The summary permits to modify the event information with the top-right button. The button on the right permits to change the location. Also, the *Travlender* can modify the default settings, tapping on the four buttons on the bottom. The first one is the *quick/cheap/eco* button. The second one is the *work/personal* button. The last but one is the *not wet* travel button. The last one is the *notifications* button. The bottom-right button is the *Go* button, tapping that the travel starts.

Once the *Travlender* taps on the *Go* button, he/she can't modify the travel settings anymore. From now, the *Travlender* can buy in-app all the required tickets, if he/she wants.

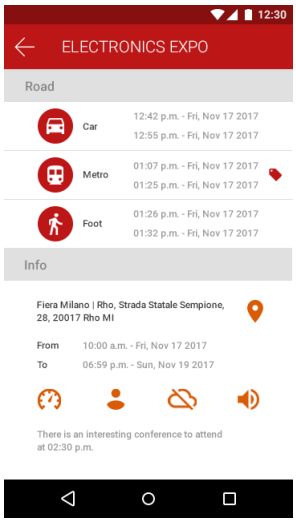

Figure 4.16: Started event

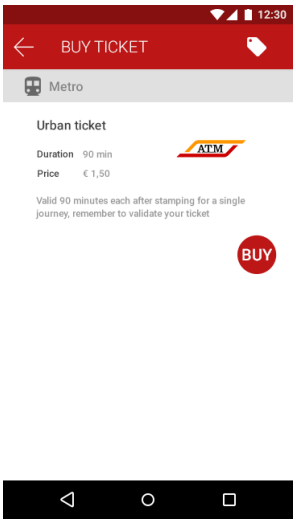

Figure 4.17: Ticket to be purchased

Tapping the ticket icon, the *Travlender* can see all the ticket details. The *Buy* button permits the user to buy the ticket.

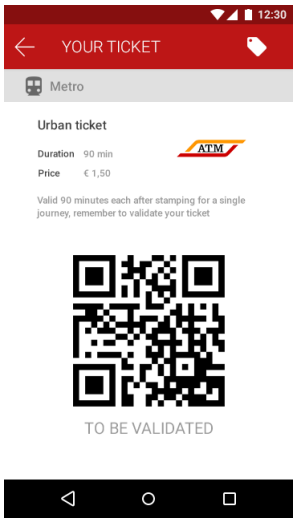

Figure 4.18: Purchased ticket

Tapping on the ticket icon once the *Travlender* bought the ticket, he/she can see a summary of the ticket details.

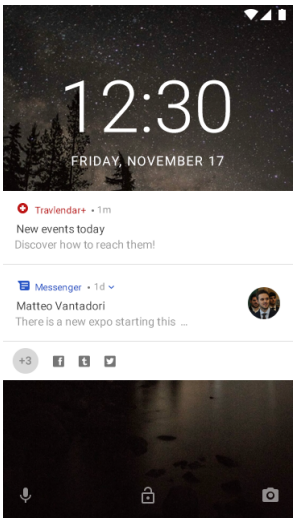

Figure 4.19: Lock screen

If there is at least a lasting event, a low-priority notification appears on the lock screen once a day.

4.2 User interface diagram

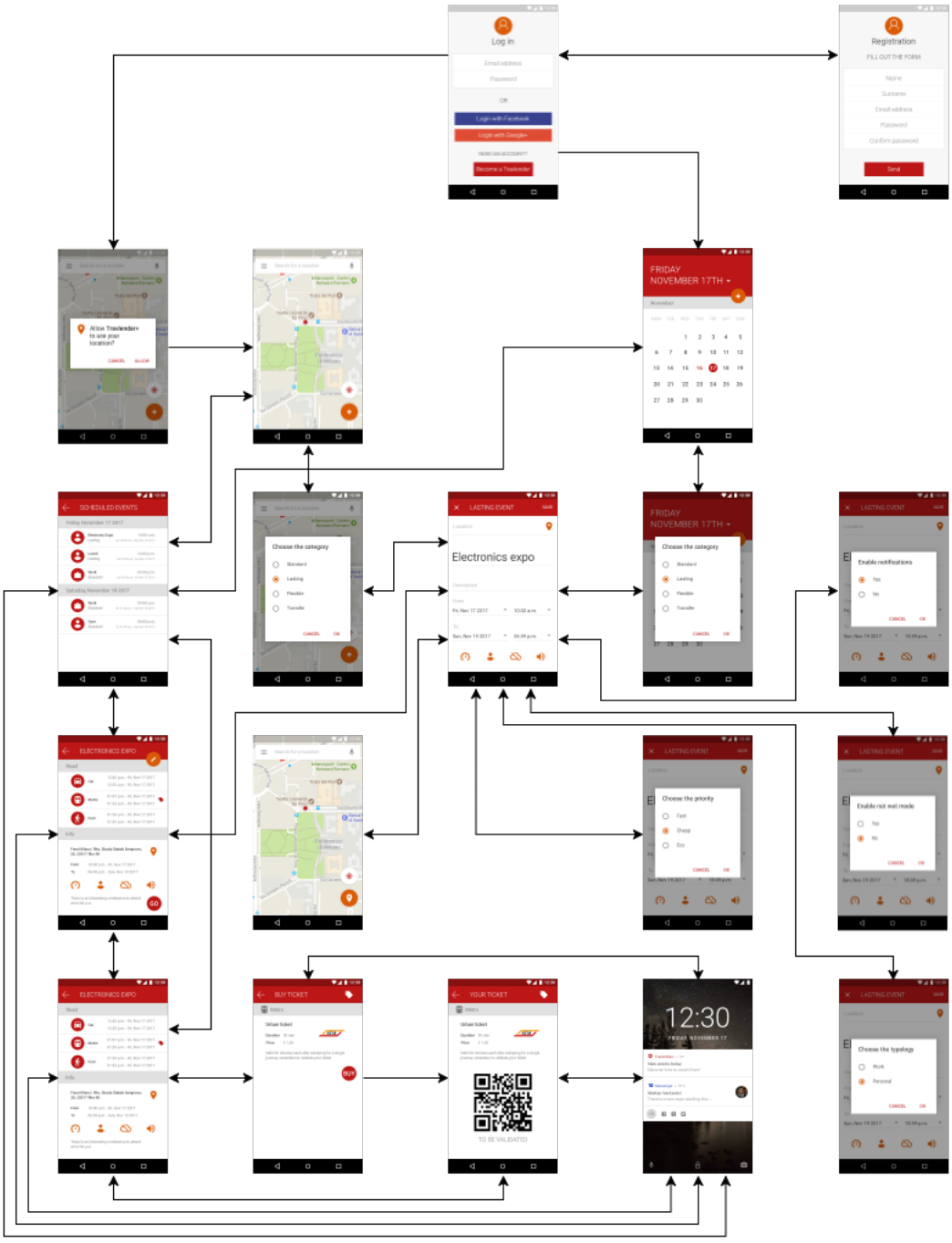

Figure 4.20: User interface diagram

5 | Requirements Traceability

Section [R.1] treats the requirements related to the event scheduling:

R.1.1 The system must keep track of the user commitments.
Data

Section [R.2] treats all the requirements related to the creation or modification of an event:

R.2.1 The system must verify that the *Travlender* is free during all the event time slot.
CalendarController, TravelController, InformationManager, MapManager, Model, Data

R.2.2 The system must verify that the *Travlender* is free during all the travel time slot.
CalendarController, TravelController, InformationManager, MapManager, Model, Data

R.2.3 The system must guarantee that the new event doesn't overlap the previous event time slot.
CalendarController, TravelController, InformationManager, MapManager, Model, Data

R.2.4 The system must avoid that the new event compromizes the reachability of the upcoming event.
CalendarController, TravelController, InformationManager, MapManager, Model, Data

R.2.5 The system must inform the *Travlender* when it's not possible to schedule the new event in the chosen time slot.
DeviceView, DeviceController, LocalModel, Dispatcher, CalendarController, TravelController, InformationManager, MapManager, Model, Data

R.2.6 The system must insert the new event after having checked whether it's possible to do it.
DeviceView, DeviceController, LocalModel, Dispatcher, TravelController, CalendarController, InformationManager, MapManager, Model, Data

R.2.7 The system must suggest possible solutions to arrange any overlap, either postponing the starting time of the upcoming event or anticipating the ending time of the previous one.
DeviceView, DeviceController, LocalModel, Dispatcher, TravelController, CalendarController, InformationManager, MapManager, Model, Data

Section [R.3] treats all the requirements related to the public, shared and non-shared transports:

R.3.1 The system must locate all the bike sharing system in Milan on the map.
DeviceView, DeviceController, LocalModel, InformationManager, MapManager, Model

R.3.2 The system must locate all the car sharing system in Milan on the map.
DeviceView, DeviceController, LocalModel, InformationManager, MapManager, Model

R.3.3 The system must locate all the public transports stops in Milan on the map.
DeviceView, DeviceController, LocalModel, InformationManager, MapManager, Model

R.3.4 The system must unlock a shared car when required.
AccountController, TravelController, PurchaseManager, Model, Data

R.3.5 The system must unlock a shared bike when required.
AccountController, TravelController, PurchaseManager, Model, Data

R.3.6 The system must let the *Travlender* book a taxi via-app.
AccountController, TravelController, PurchaseManager, Model, Data

R.3.7 The system must let the *Travlender* book a limousine via-app.
AccountController, TravelController, PurchaseManager, Model, Data

Section [R.4] treats all the requirements related to the travel:

R.4.1 The system must control if the (eventually) required tickets are already owned by the *Travlender*.
InformationManager, Model, Data

R.4.2 The system must evaluate the best way to travel for the *Travlender*, according to his/her preferences, constraints and owned means.
TravelController, CalendarController, MapManager, InformationManager, Model, Data

R.4.3 The system must not suggest the *Travlender* to use the owned car and bike for a travel starting from a location where they aren't placed.
TravelController, MapManager, Model, Data

R.4.4 The system must allow the *Travlender* to purchase tickets via-app.
TravelController, PurchaseManager

Section [R.5] treats all the requirements related to the special events:

R.5.1 The system must allow the *Travlender* to create an overlappable event.
DeviceView, DeviceController, LocalModel, Dispatcher, TravelController, CalendarController, InformationManager, MapManager, Model, Data

R.5.2 The system must allow the *Travlender* to create an event with a specified reservation time slot.
DeviceView, DeviceController, LocalModel, Dispatcher, TravelController, CalendarController, InformationManager, MapManager, Model, Data

Section [R.6] treats all the requirements related to the *Travlender* preferences:

R.6.1 The system must keep track of the *Travlender* public transports passes and tickets.
Data

R.6.2 If asked, the system must notify the user if a scheduled event takes place on an adverse weather conditions day.
DeviceView, DeviceController, LocalModel, Dispatcher, CalendarEvent, InformationManager, Model, Data

R.6.3 The system must allow a *Travlender* with disabilities to reach the event evaluating a way according to his/her needs.
TravelController, CalendarController, InformationManager, MapManager, Model, Data

R.6.4 The system must keep track of the means owned by the *Travlender*.
Data

R.6.5 The system must keep track of the means selected by the *Travlender*.
Data

R.6.6 The system must be able to evaluate the fastest way to reach the event, according to the *Travlender* preferences, constraints, owned means, tickets and passes.
TravelController, CalendarController, InformationManager, MapManager, Model, Data

R.6.7 The system must be able to evaluate the cheapest way to reach the event, according to the *Travlender* preferences, constraints, owned means, tickets and passes. **TravelController, CalendarController, InformationManager, MapManager, Model, Data**

R.6.8 The system must be able to evaluate the most ecological way to reach the event, according to the *Travlender* preferences, constraints, owned means, tickets and passes. **TravelController, CalendarController, InformationManager, MapManager, Model, Data**

6 | Implementation, integration and test plan

6.1 Preconditions

As specified in the *Domain Assumtpion* chapter of the *Requirements Analysis and Specification Document*, all the third-party services *Travlendar+* relies on are always available and correct functioning. This results in a correctly functioning of the *fbLogin.sdk*, *googleLogin.api*, *map.api*, *infoMap.api*, *purchase.api* and *infoTransport.api*. On this basis, the *LoginController*, *PurchaseManager*, *InformationManager* and *MapManager* can be fully tested, since they communicate directly with the different third-party services.

6.2 Proposed plan

6.2.1 White-box testing

First of all, each component will be tested using the white-box method, focusing on all the functions that don't provide any interaction with the other components. This makes sure that any future problem will depend only on the compontents interaction itself.

6.2.2 Local Model-View-Control pattern testing

The first step will be to test separately the *DeviceView* both with the *DeviceController* (for example, checking the *view* returned by the *controller*) and the *LocalModel* (for example, checking the *model* returned by the *view*).

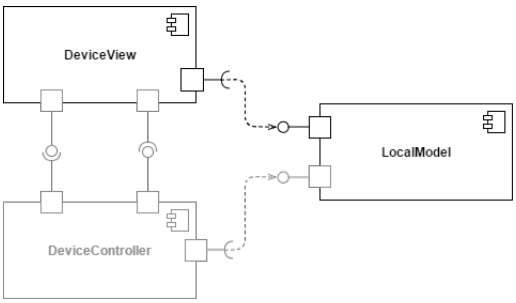

Figure 6.1: DeviceView with LocalModel

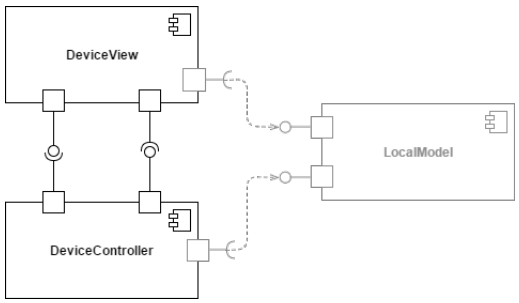

Figure 6.2: DeviceView with DeviceController

The second step will be to join the three components and test them together.

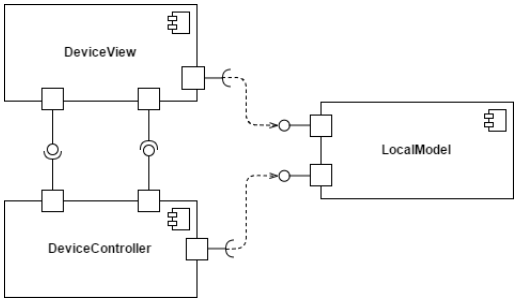

Figure 6.3: DeviceView with both LocalModel and DeviceController

6.2.3 Dispatcher pattern testing

To ensure the proper *Dispatcher* functioning, it will be tested with each controller to which it forwards the requests. Testing the *Dispatcher* and the *RegistrationController* together will make sure that each registration request will be dispatched to that controller.

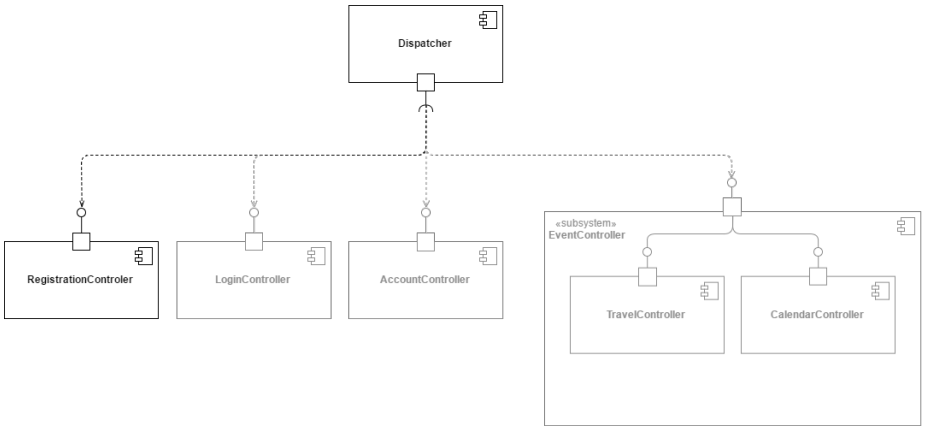

Figure 6.4: Dispatcher with RegistrationController

Testing the *Dispatcher* and the *LoginController* together will make sure that each login request will be dispatched to that controller.

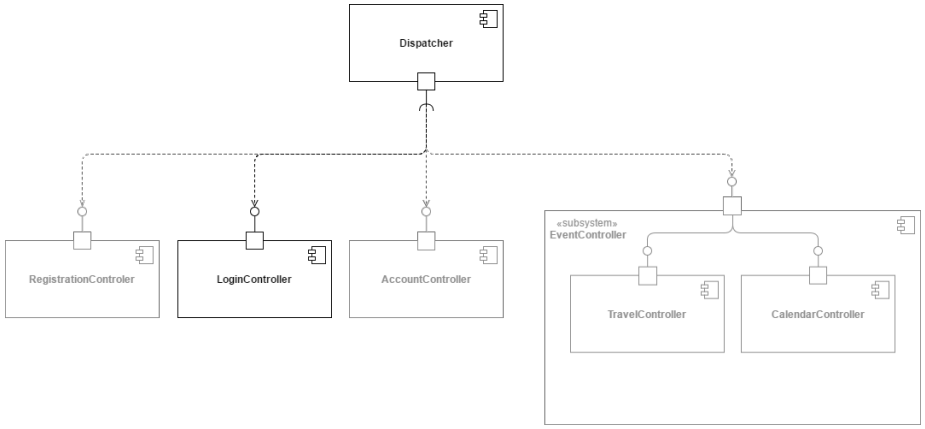

Figure 6.5: Dispatcher with LoginController

Testing the *Dispatcher* and the *AccountController* together will make sure that each account request will be dispatched to that controller.

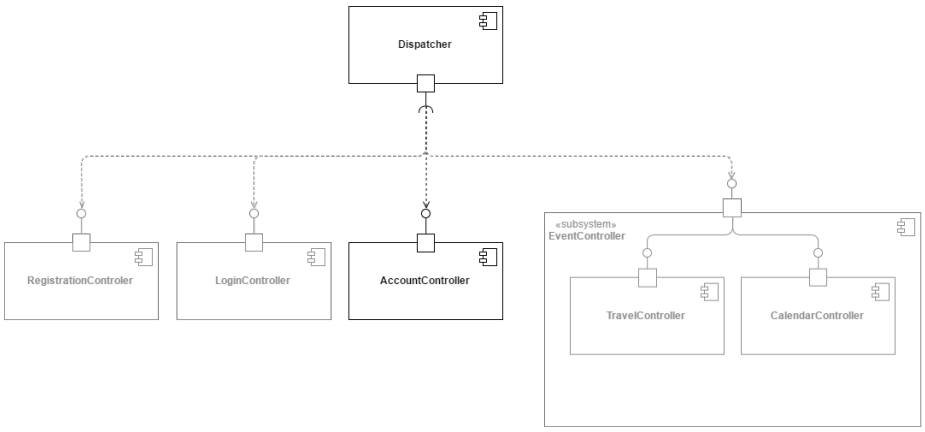

Figure 6.6: Dispatcher with AccountController

Testing the *Dispatcher* and the *TravelController* together will make sure that each event request that involve the *TravelController* will be dispatched to that controller.

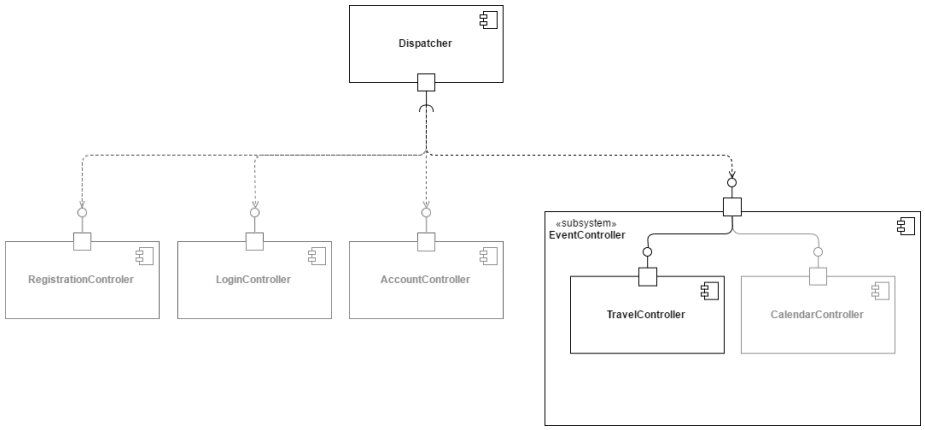

Figure 6.7: Dispatcher with TravelController

Testing the *Dispatcher* and the *CalendarController* together will make sure that each event request that involve the *CalendarController* will be dispatched to that controller.

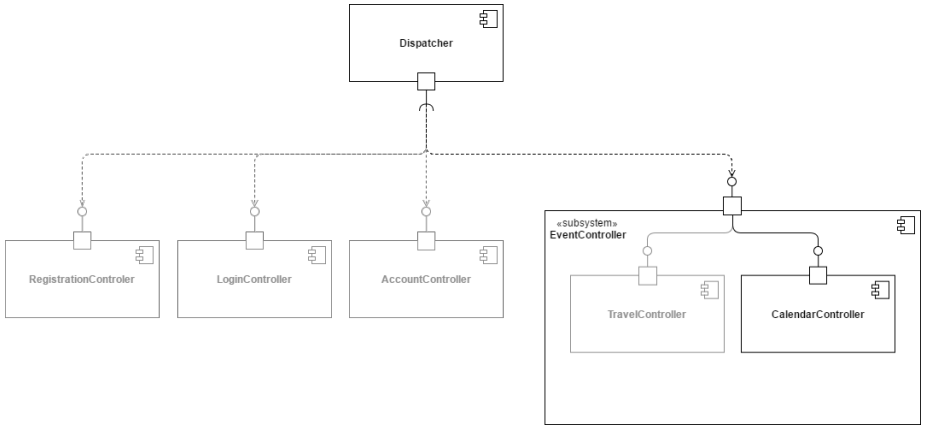

Figure 6.8: Dispatcher with CalendarController

6.2.4 Model and Data testing

First of all, the interaction between *Model* and *Data* will be tested to check the proper data storage.

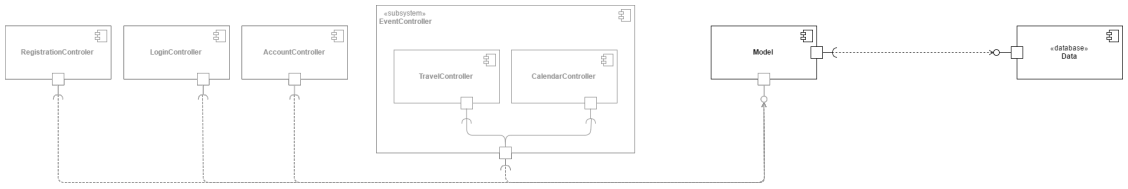

Figure 6.9: Model with Data

Then, a series of tests beetwen the *Model* and each component will be take. Testing the *Model* and the *RegistrationController* together will make sure that the controller will proper store data on the *Model*.

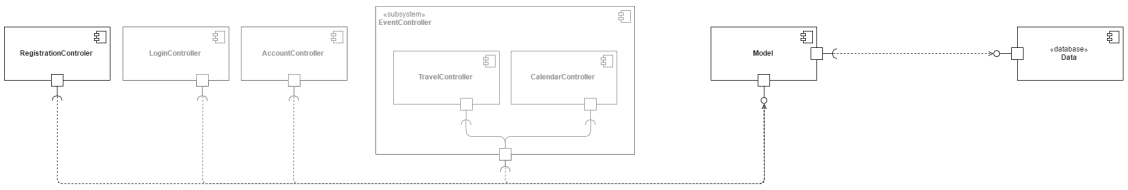

Figure 6.10: RegistrationController with Model and Data

Testing the *Model* and the *LoginController* together will make sure that the controller will proper store data on the *Model*.

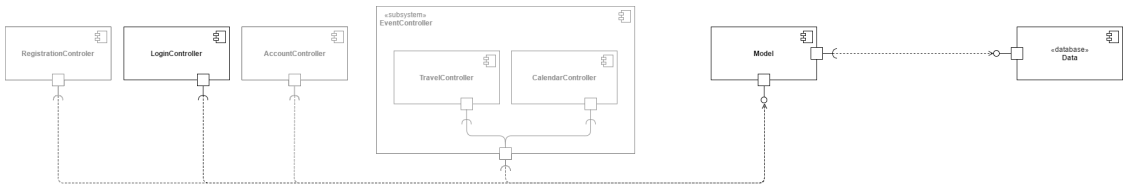

Figure 6.11: LoginController with Model and Data

Testing the *Model* and the *AccountController* together will make sure that the controller will proper store data on the *Model*.

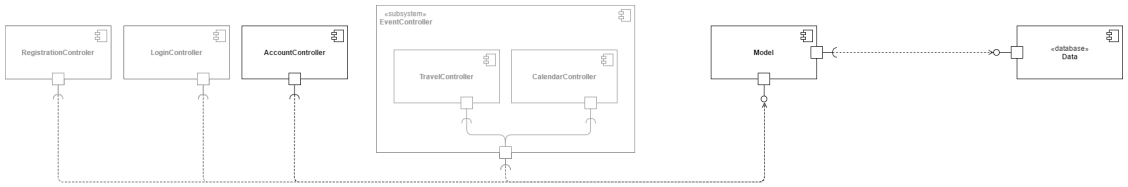

Figure 6.12: AccountController with Model and Data

Testing the *Model* and the *TravelController* together will make sure that the controller will proper store data on the *Model*.

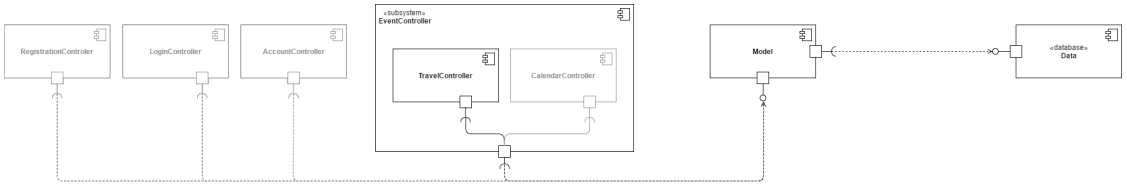

Figure 6.13: TravelController with Model and Data

Testing the *Model* and the *CalendarController* together will make sure that the controller will proper store data on the *Model*.

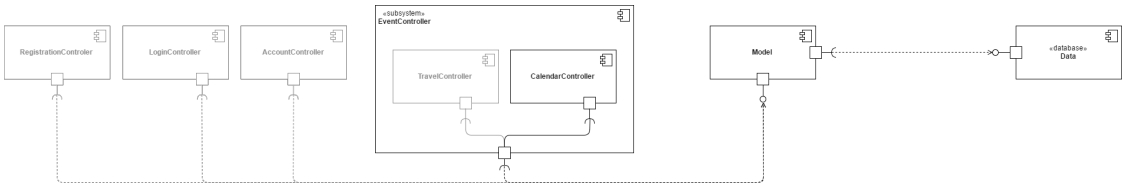

Figure 6.14: CalendarController with Model and Data

6.2.5 EventController testing

The *EventController* proper functioning is one of the most critical integration parts, because of the close cooperation beetwen its two subcomponents and the combined interaction beetwen the *TravelController* and the three managers. The first step will be to test the *TravelController* with the *InformationManager*, because it provides many information on which the *TravelController* relies to make requests to the *PurchaseManager* and the *MapManager*.

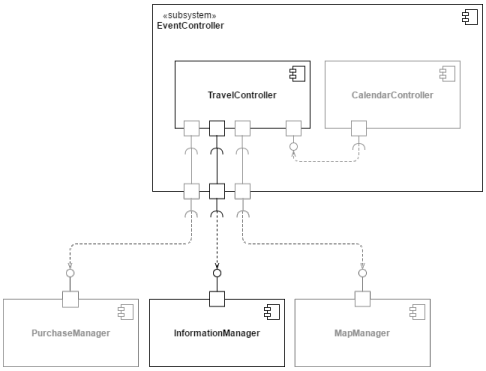

Figure 6.15: TravelController with InformationManager

The second step will be to test separately the *TravelController* both with the *PurchaseManager* and the *MapManager*.

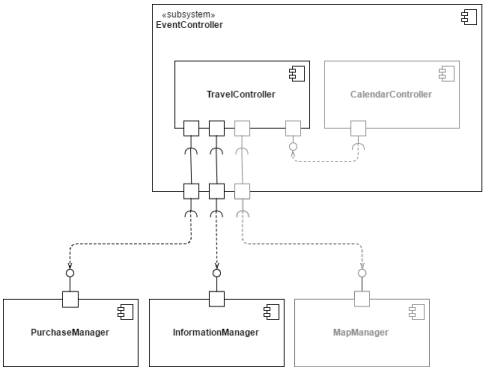

Figure 6.16: TravelController and InformationManager with PurchaseManager

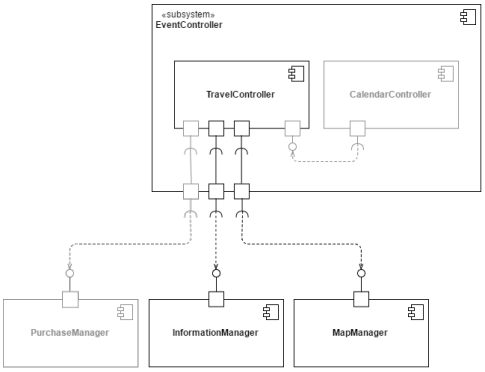

Figure 6.17: TravelController and InformationManager with MapManager

The third and final step will be to test the *TravelController* with the *CalendarController*.

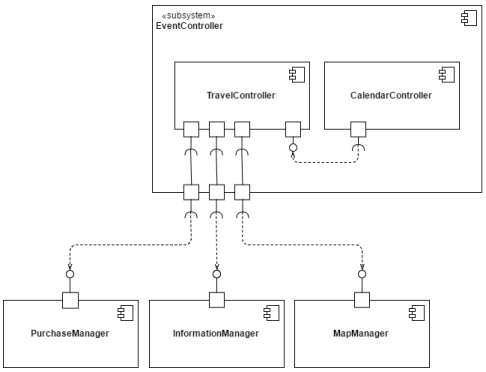

Figure 6.18: CalendarController with TravelController and the three managers

6.2.6 Model-View-Control pattern testing

The final two steps relate on the *view* interaction between the *controller* and the *model*. First, it will be tested the interaction between the *view* and the *controller*.

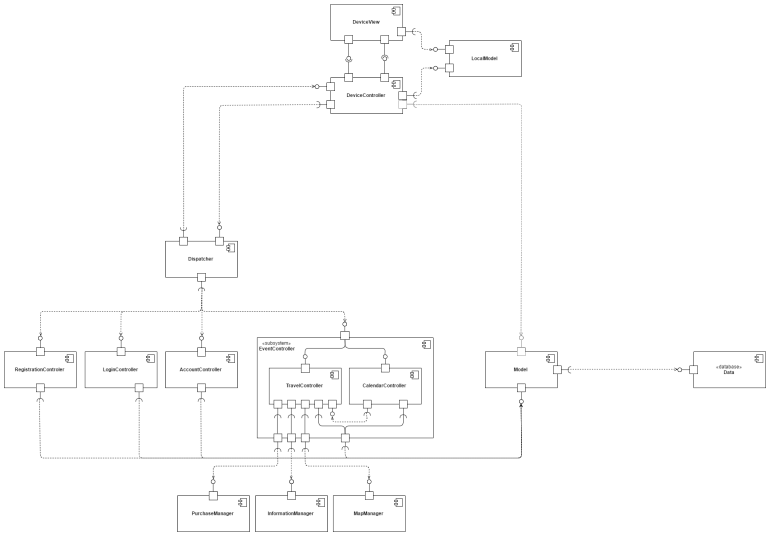

Figure 6.19: View with controller

Then, it will be tested the remaining interaction, between the *view* and the *model*.

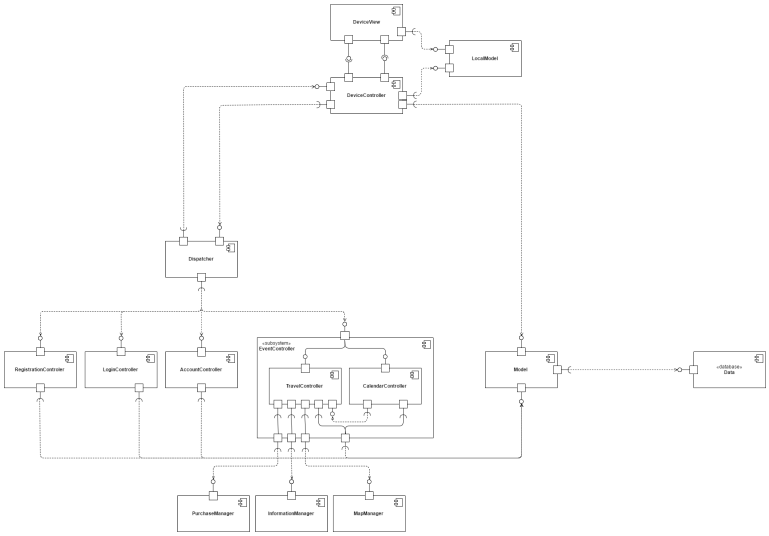

Figure 6.20: View and Controller with Model

7 | Effort spent

09-nov	21:00-00:00	3 hours
10-nov	10:00-12:00 15:00-20:00	7 hours
11-nov	16:00-19:00 21:00-00:00	6 hours
12-nov	14:00-18:00	4 hours
13-nov	21:00-01:00	4 hours
14-nov	16:00-18:00 22:00-01:00	5 hours
15-nov	17:00-19:00 22:00-00:00	4 hours
16-nov	15:00-19:00 21:00-01:00	7 hours
17-nov	15:00-19:00	4 hours
18-nov	10:00-13:00	3 hours
19-nov	15:00-19:00	4 hours
20-nov	21:00-02:00	5 hours
21-nov	16:00-18:00 22:00-01:00	5 hours
22-nov	15:00-18:00 21:00-01:00	8 hours
23-nov	21:00-03:00	6 hours
24-nov	10:00-12:00 15:00-20:00	7 hours
25-nov	10:00-12:00 14:00-20:00 22:00-02:00	12 hours
26-nov	10:00-13:00 15:00-23:00	11 hours