DESIGN DOCUMENT

Project of Software Engineering 2

WEATHER-CAL

Authors:

PAOLO POLIDORI

MARCO EDEMANTI

Contents

Chapter 1

Introduction

1.1 Purpose

This document describes the high level design and the technology involved in the development of the WeatherCal software. The target will be accomplished by the use of a description of the architecture which comes after the analisys of the problem and the constraint, explained in chapter 2. The final design and how the application will be developed is shown in chapter 3. This document is a supplement of the RASD formerly redacted. Provide an overview of the entire document.

1.2 Scope

This document is intended for the stakeholders of the system, the developers and reviewers/testers.

1.3 References

[1] IEEE, *IEEE Std 1016-2009,IEEE Software Design Descriptions*, IEEE Computer Society 1998.

[2] Raffaela Mirandola, *Design and software architecture - slides from SE2 course*, 2014.

[3] Paolo Polidori, Marco Edemanti, *Requirement analisys and specification document for WeatherCal project*, 2014.

[4] Jesse James Garrett, *Ajax: A New Approach to Web Applications*, February 18th, 2005.

Chapter 2

System

2.1 System Description

The system will implement a calendar as a web application, splitted into client-side and server-side (motivations discussed in section 2.2 and section 2.3). The former will be used for implementing the asyncronous facilities delivered in the calendar, while the latter will be used both as an interface for the former to interact with the persistency and for providing web pages to the client.

2.2 Design Constraints

The application, first, will have some constraints on the system proposed by the client explicitly. The first one is the use of J2EE as server-side application implying the use of a storage for persisting the data (events, users, invitations, etc.). This constraint entails that the client-server architecture will be adopted in the WeatherCal system.

Client constraints even include the time for the system development, which is due

to January 25th, 2015.

Constraint imposed by the client does not include any strict restraint on the hardware and the software over which the system will need to be deployed and any further requirement will be added, giving the possibility to be platform independant. Anyway the system on which the platform will be deployed on will have an impact on the server-side application performances and both the client-side environment and the network connecting the client and the server will impact the client-side application performances. Even though both the client and the server software involved in this project have some requirements on the hardware and the software to be used, so they will make our constraints.

2.3 System Architecture

As said in section 2.2 the system will use J2EE for implementing the server-side application and thus the system will rely on a client-server achitecture. The server will also implement the MVC design pattern by means of the Java Server Faces framework, which will facilitate the development of the structure taking advantage of both the design pattern and the facilities brought.

Another choice is to implement a web client because, instead of a traditional application, it provides universal access and no need of being in possess of a dedicated client application.

The application also needs to persist data, so we decided to use PostgreSQL free RDBMS to accomplish this task.

The related client side will be developed using both the Web tier provided by JSF and Marionette.js, a Javascript framework, with its dependecies, for making the client more responsive and interactive. This framework implements MVC, so the

client will have the same design pattern of the server, with same necessity of storing data. This target will be accomplished by the Java API for RESTful Web Services integrated in J2EE, giving the opportunity of sharing and syncronizing the models. This path was chosen because traditional web application, which are synchronous, need to change the webpage everytime some data need to be exchanged with the server, giving the user a worse experience [4] and it offers more capabilities than the AJAX facilities provided by JSF.

The graphic environment of the web pages will be managed by PrimeFaces, a JSF component suite. It was chosen among other similar libraries both for its features and performances in combination with the availability of support from the client.

Chapter 3

Design

3.1 Persistance design

Our system needs to store and retrieve persistent data, to achieve this task we decide to use PostgreSQL free RDBMS. In the following paragraph **??** we'll analyze the design of our database.

3.1.1 ER Diagram

To understand in the best way possible the organization and the design of our database we'll describe and analyze it through the use of the Entity-Relationship Diagram in figure **??**. The main entity of the database are four:

- **User**: it represents the data of a registered user and contains all his information such as the email, the first name, the surname, the username and the password. It's in a relationship "one to many" with the Calendar entity because we assume that in a future our users may have one or more calendar and not only a single agenda.

- **Calendar**: it stands for the agenda of the user, its only attribute is a boolean that indicates whether is public or not. It's related with the User and the Event entity. It's in a "one to one" relationship with the User because a calendar belongs only to an unique User. While the relationship with the Event entity could be of two different type: Participation, Owner. The first one is a relationship "zero to many" because it could have either no participant or many participant to it. While the second one is an "one to many" relationship because an event needs at least one owner that had previously created it and eventually it can have more owner that can manage it.

- **Event**: it represents the information stored about an existing event such as his participants, the place, the date an so on. Because it could belong at least to his owner calendar or to his participant calendar and seeing that the participant can be zero but the owner must be at least one then the event must be related through an "one to many" relationship with the Calendar. More over is in relation "zero to one" with the WeatherConstrain because it could have a weather bond as no one.

- **WeatherContraint**: it stands for the constraint associated to an Event such as the temperature. It's in a relation "one to one" with an Event, because any WeatherConstraint belong exactly to one Event but is in a relation "1 to many" with the WeatherStateConstraint that represent the desired weather condition of an event such as sunny, cloudy , rainy and so on, in fact any user has the possibility to choose different desired condition when creating a new event.

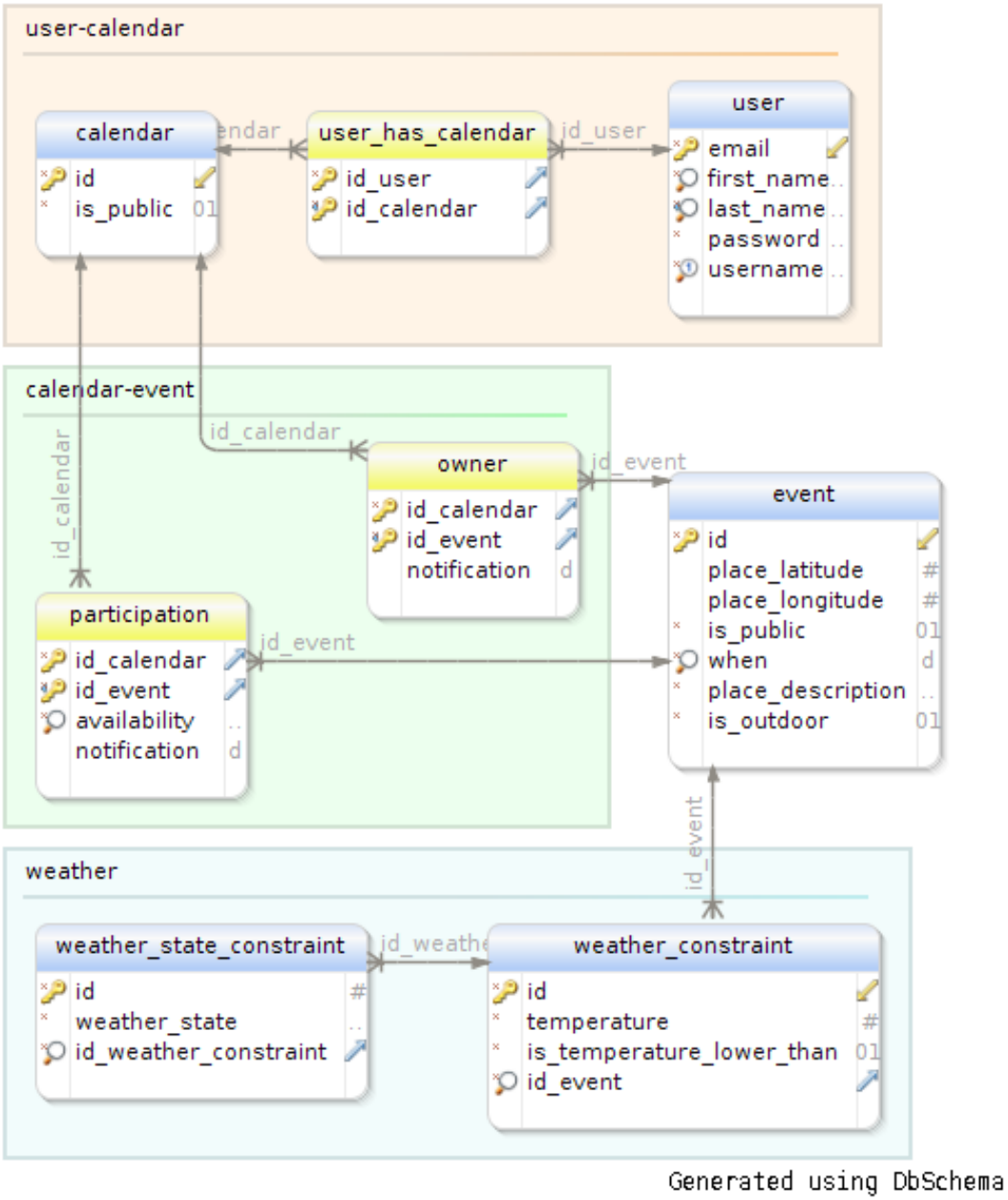

Figure 3.1: ER Diagram

3.2 Modeling

<Describe any standards or typographical conventions that were followed when writing this SRS, such as fonts or highlighting that have special significance. For example, state whether priorities for higher-level requirements are assumed to be inherited by detailed requirements, or whether every requirement statement is to have its own priority.>

3.2.1 Structure

3.2.1.1 Server-side BCE

This paragraph illustrates the BCE pattern representing the architecture that we'll implement in order to develop the WeatherCal system. When identifying the elements for some scenario of system behavior, we can align each participating element with one of three key perspectives : Boundary, Control and Entity.
This pattern is a variation of the MVC pattern indeed we can consider mapping the Boundary with the MCV's View, the Control with the MCV's Controller and the Entity with the MVC's Model. Moreover the BCE pattern is not solely appropriate for dealing with user interfaces but it gives also to the controller a slightly different role to play.
Let's take a deeper look in to the BCE pattern:

- Entity: are objects representing system data and also they perform behavior organized around some cohesive amount of data.

- Boundaries: are the objects that interface with system actors and most of the times they lay on periphery of a system. Some boundary elements will be "front-end" elements that accept input from outside the area under design

and other elements will be "back-end" managing communication to supporting elements outside the system or subsystem.

- Control: are the elements that mediate between boundaries and entities and control the flow of the interaction of the scenario. They manage the execution of commands coming from the boundary.

3.2.1.1.1 Entity overview

The diagram in 3.1 illustrates an overview of all entities involved in our system and how they are related with each other.

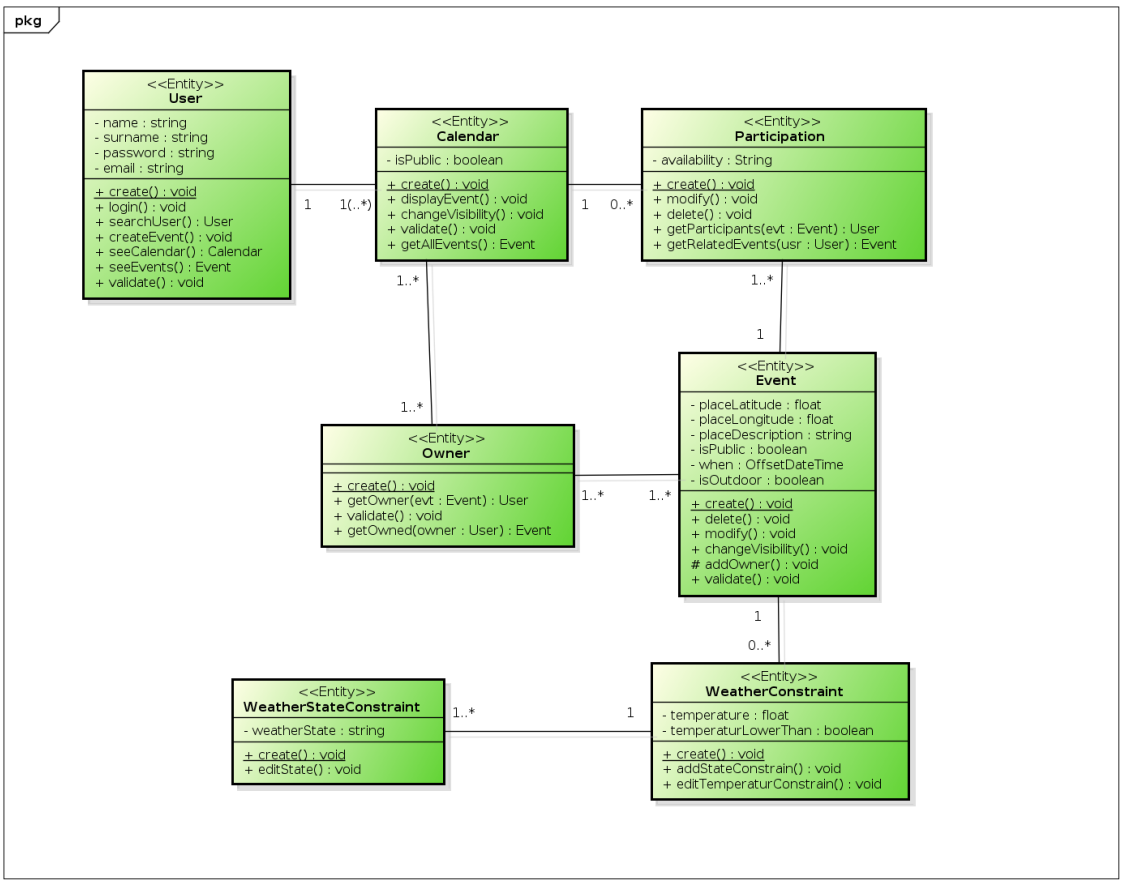

Figure 3.2: Entities involved

3.2.1.1.2 Sign Up and Log In

The diagram in 3.2 shows the flow of the system's behavior related to the registered user's login or to the anonymous user's signup. After the SignUp Controller validates the values submitted by the user, the user reaches his UserPage and then the CalendarController loads its agenda and all associated event.

In these diagram there are three entities who plays an active role:

1. **User**: it's a registered user that can log in to the platform and gets to his UserPage, or he can represents someone who is not already signed in to the system and can only reach the main page and register to the platform.

2. **Event**: represents an event in the calendar. An user can be related to it in two different way, it could be a guest or it could be its owner. Depending on this relation it can perform different action. He can modifies it or creates a new one if he's his owner or change his participation if he's a guest.

3. **Calendar**: is an element that represents the agenda of an user and contains all his scheduled event. It can be set public or private by its owner.

There are two boundaries involved in this scenario:

1. **MainPage**: it represents the index page of the system. The one in which a user can either log in or sign up to the platform.

2. **UserPage**: it stands for the page reached by the user after he logged in. It shows his calendar and all other tasks that the user can performs within it, such as searches for a user, creates or modifies an event or checks for notification.

The controls who manage the flow of this scenario are two:

1. **SignUpController**: it's the control whose role is to handle the registration's request of a new user into the system. Whenever a non registered user submits his information the SignUpController verifies the correctness of these information and if they are valid it creates a new User and redirect him to the UserPage.

2. **LoginController**: his task is to manage the log in of a registered user. It verifies that the credentials submitted by the user are the same provided in the system registration.

3. **CalendarController**: It's the responsible for the control of the calendar of an user, it loads it and all its associated events, through it an user can search other users' calendar and view them if they were set as public by their owner.

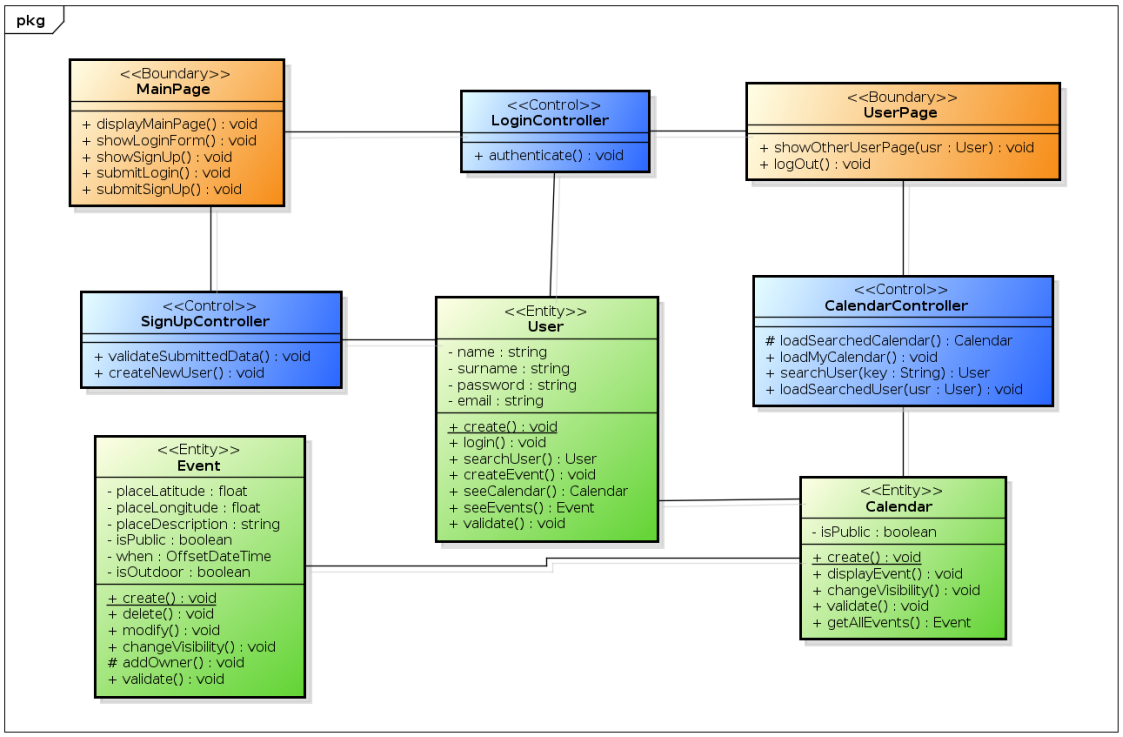

Figure 3.3: Sign Up and Log In

3.2.1.1.3 Event creation or modification

The diagram in 3.3 illustrates the flow of the creation of the modification of an event. Naturally an user can perform these action only if it's logged in to the system. The top side of the diagram it's related to the log in mechanism already seen in fig 3.2 while the bottom side represents the management of an event. From that point on through the EventController a user can either create,delete or modify an event, see the event information or change the participation to it.

In these diagram there are only two entities who plays an active role:

1. **User**: it's a registered user that can log in to the platform and gets to his UserPage, or he can represents someone who is not already signed in to the

system and can only reach the main page and register to the platform.

2. **Event**: represents an event in the calendar. An user can be related to it in two different way, it could be a guest or it could be its owner. Depending on this relation it can perform different action. Modify or create it if it's an owner or change his participation if it's a guest.

There are two boundaries involved in this scenario:

1. **UserPage**: it stands for the page reach by the user after he logged in. It shows his calendar and all other tasks that the user can performs within it such as searches for a user, creates or modifies an event or checks for notification.

2. **NewModifyEventPage**: is the page that a user uses to create or manipulate an event. It displays all the informations about the selected event such as the place, the date and the desired weather. More over gives to the owner of the event the capability to modify these data or to insert new informations in case that the user is creating a new event.

The controls who manage the flow of this scenario are two:

1. **LoginController**: his task is to manage the log in of a registered user. It verifies that the credentials submitted by the user are the same provided in the system registration.

2. **EventController**: this control has several duties many of which concerning the event creation or modification. It's able either to load the information regarding an existing event, to delete an event or to check the correctness of the submitted values for its attributes when the event is created. In addition an invited user to the event can eventually changes the participation to it.

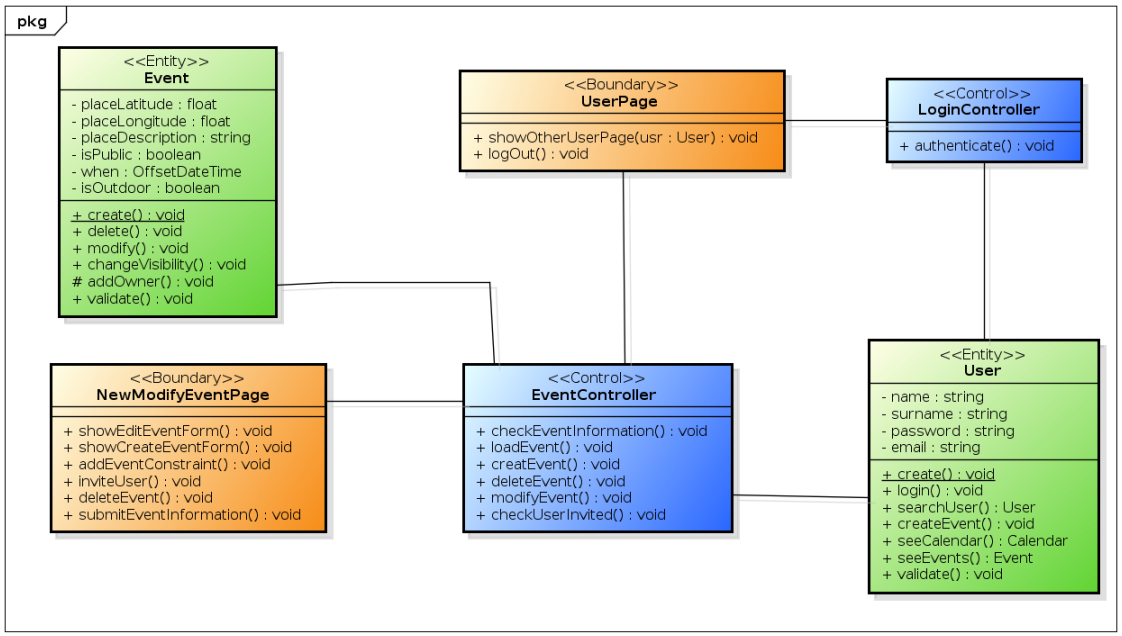

Figure 3.4: Create or modify an event

3.2.1.1.4 Search Users

The diagram in fig 3.4 display the scenario involved when a user searches for an other user profile. The flow depends also by the visibility of both the searched calendar and the events within it. This action of course can only be performed by a logged user.

In these diagram there are three entities:

1. **User**: it's a registered user that can log in to the platform and gets to his UserPage, or he can represents someone who is not already signed in to the system and can only reach the main page and register to the platform.

2. **Event**: represents an event in the calendar. An user can be related to it in two different way, it could be a guest or it could be its owner. Depending on

this relation it can perform different action. Modify or create it if it's an owner or change his participation if it's a guest.

3. **Calendar**: is an element that represents the agenda of an user and contains all his scheduled event. It can be set public or private by its owner.

There are two boundaries involved in this scenario:

1. **UserPage**: it stands for the page reach by the user after he logged in. It shows his calendar and all other tasks that the user can performs within it such as searches for a user, creates or modifies an event or checks for notification.

2. **OtherUserCalendarPage**: is the page that an user can reach after he searched for an other user, depending on the searched user's calendar's visibility it can shows either the calendar itself and its related event or a redirect to the home page

The controls who manage the flow of this scenario it's unique:

1. **CalendarController**: It's the responsible for the control of the calendar of an user, it loads it and all its associated events, through it an user can search other users' calendar and view them if they were set as public by their owner.

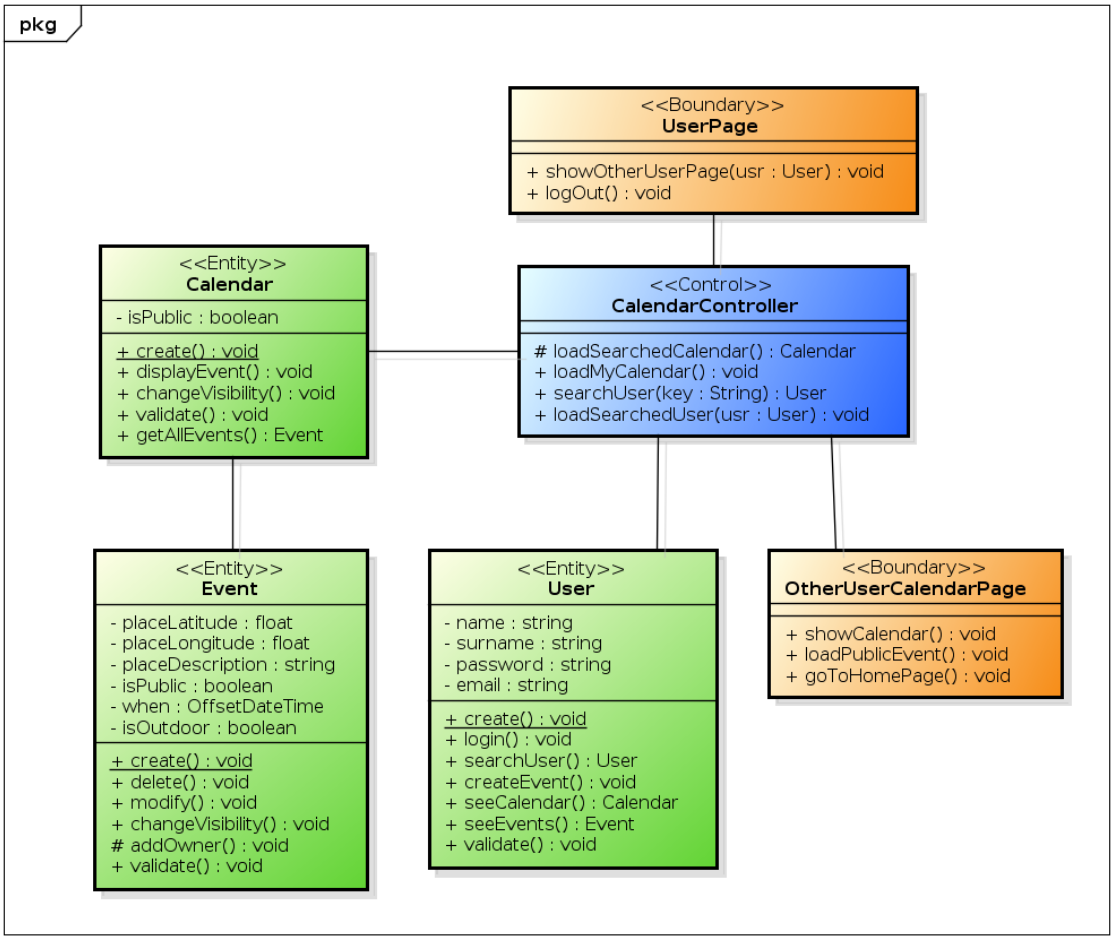

Figure 3.5: Search users

3.2.1.2 Client-side MVC

An other part of the architecture, located on the client side will rely on an MVC framework as stated in section 2.3. In this paragraph the diagrams implementing the structure of the client side application are described.

We will have three categories of system element: *Model* elements, *View* elements and *Controller* elements.

- **Model elemens** will be composed by two classes, *Collections* and *Models*, the

former implementing a set of the latters.

- **View elements** have similar differencies, even if weaker. *ItemViews* will mostly be used for displaying collections, even if they were thought for displaying single elements. Anyway, since any operation should be performed on the collection and the models contain few data, they can be used even for displaying collections. Even though, collections are usually rendered with *CollectionViews*, a sort of "set of *ItemViews*".

- **Controller elements** does not exactly exist both in the diagrams and in the framework. This is because there is no need of a real controller. Its job will be accomplished by the Application entry point, in such a way that can be associated to the Controller of the MVC pattern, but just in a coarse approximation.

3.2.1.2.1 Entity overview

The diagram in 3.5 illustrates an overview of all the models and the collections which are part of the system and how they interact among themselves.

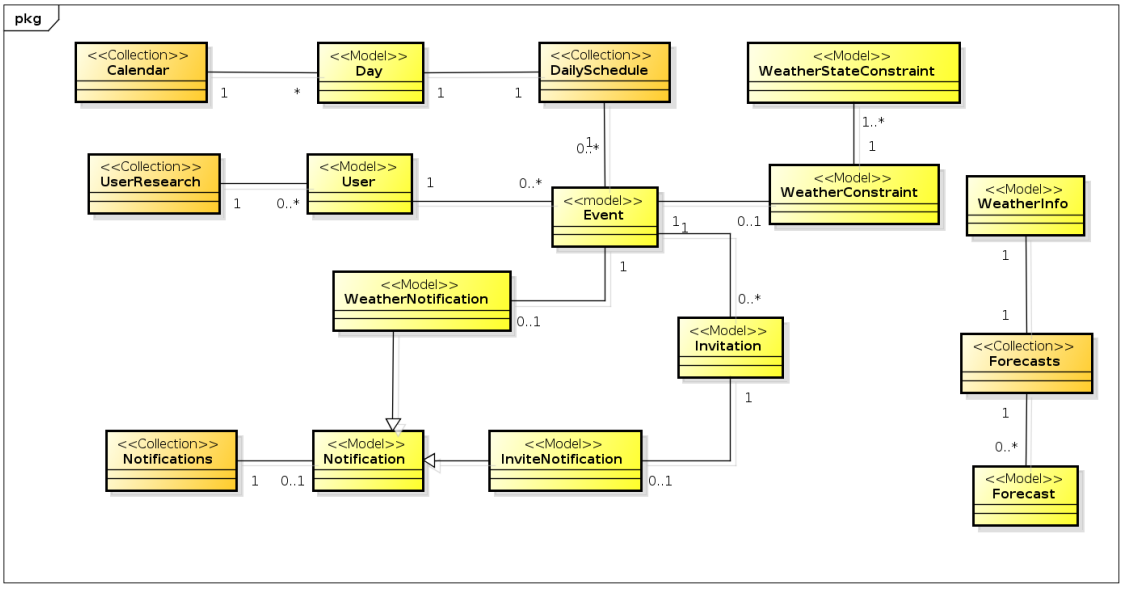

Figure 3.6: Models and Collections

3.2.1.2.2 Show notifications

In this diagram, as shown in figure 3.6 there can be two types of notification

1. **Invite notification**: it is the model representing a single notification of invite, giving the user the possibility to express his participation.

2. **Weather notification**: it is used to tell the user whether there is an adverse condition for a future event and, if owner, ask for a date modification.

Both these notification are extension of the generic **Notification**, which consists of a single element on the list of all the notifications. This model will be fetched from the server after performing some business intelligence operations over the *"Participation"* entity for retrieving both the not answered invitations and the shortcoming events having possible weather issues.

The **Notifications** will be collected inside the omonimous collection.

The collection will be displayed using an *ItemView* called *NotificationView*, which will show the list of all the notifications in the user's webpage.

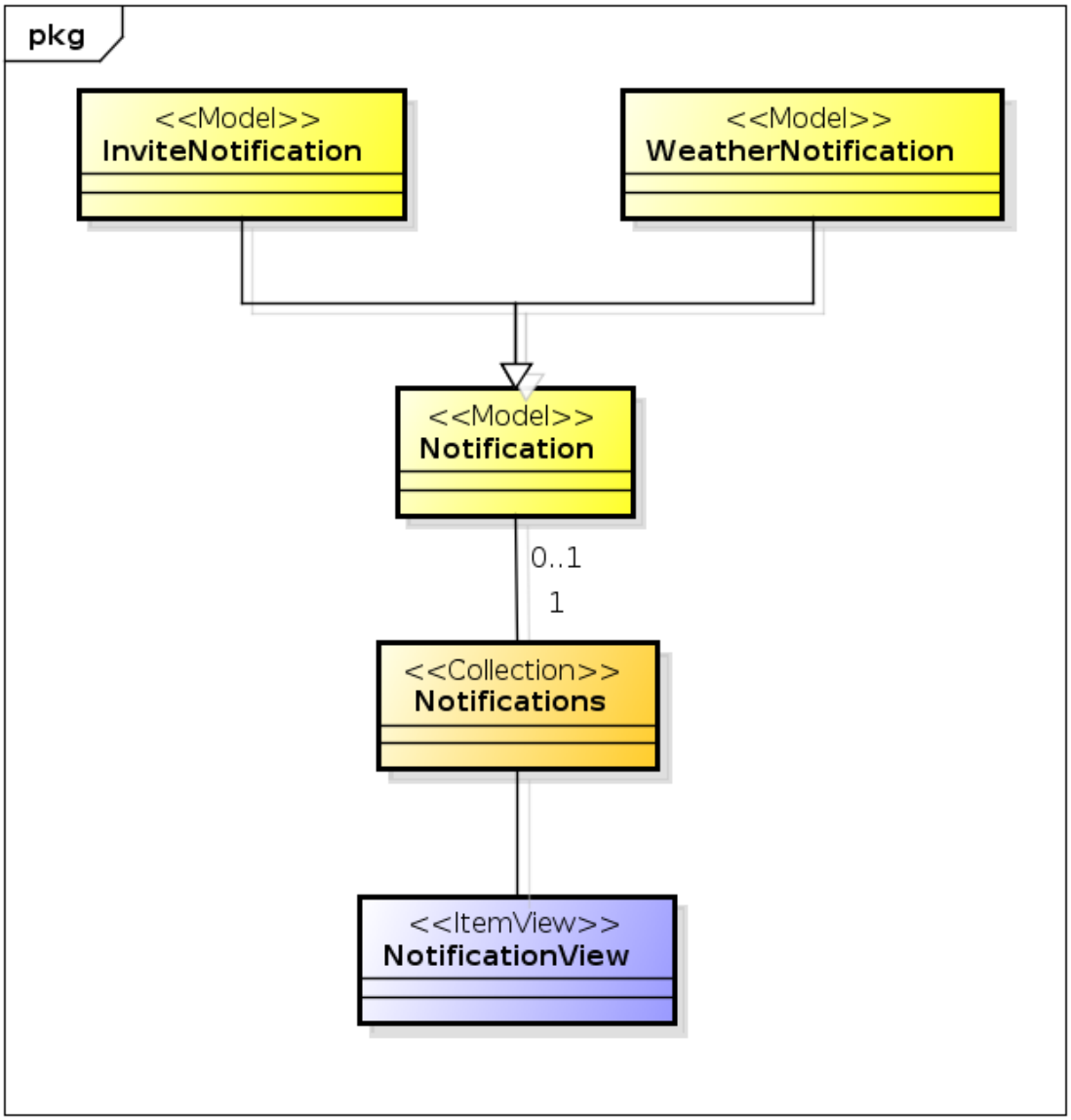

Figure 3.7: Notifications

3.2.1.2.3 Local forecast

The user will be able to see the local forecast for the next days in a dedicated area of the website. This will be accomplished using the OpenWeatherMAP RESTful API from the client. The retrived data will be modeled in figure 3.7. The first model, called **WeatherInfo** will contain the metadata of the response, including the locality details.

The **Forecasts** will be then stored inside the omonimous collection, and each one will be modeled by **Forecast**. This model will contain the forecast information about a single interval. This collection will be rendered inside the **ForecastView** *ItemView* which will be nested inside a box containing some of the data stored in the **WeatherInfo** model, managed by the **WeatherDataView** *ItemView*.

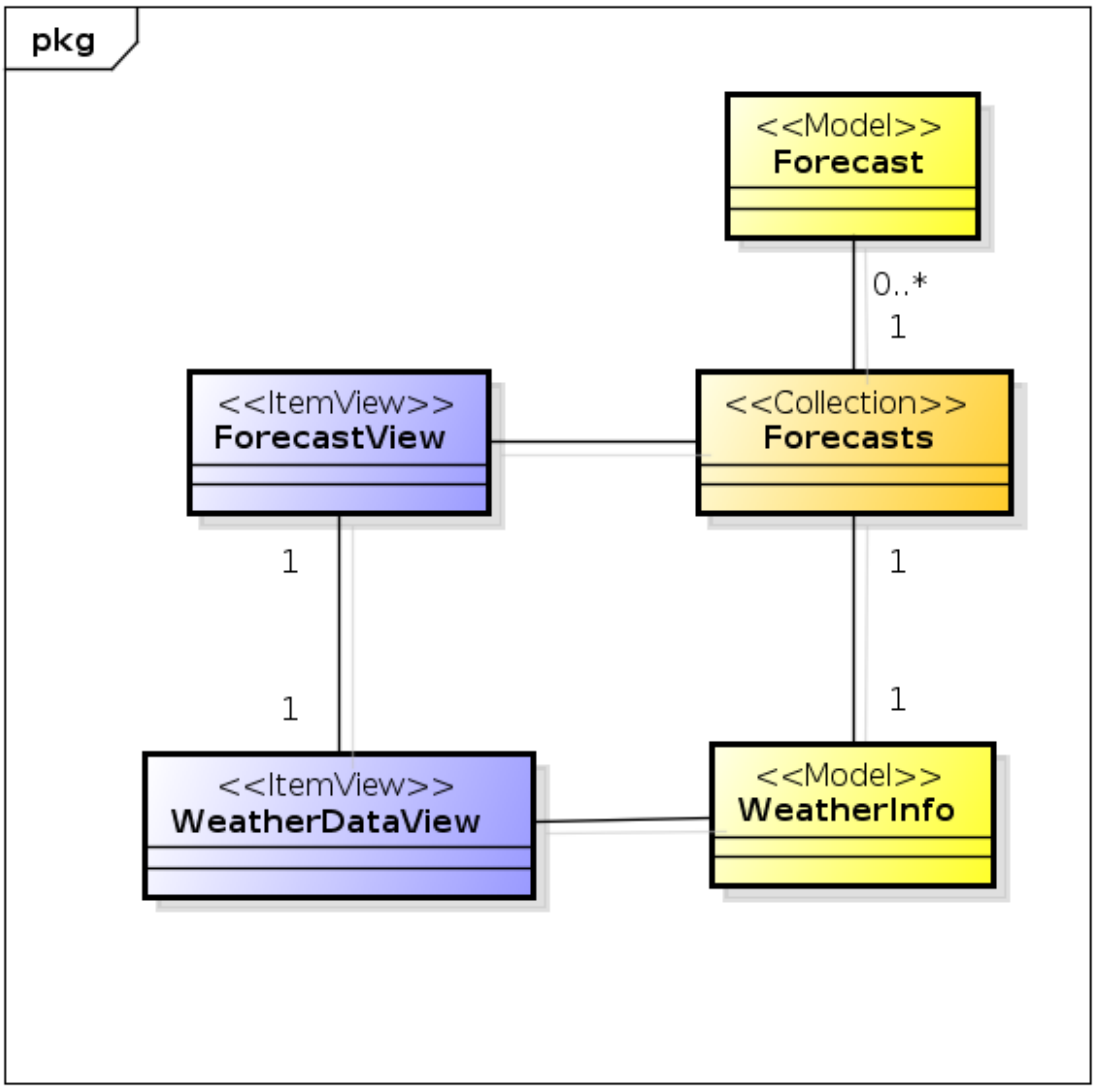

Figure 3.8: Local forecast

3.2.1.2.4 Search Users

The user will also be able to search other users, in order to find their profiles or invite them to an event. This will be realized, as illustrated in figure 3.8, from a **UserResearchView** which will list the users for a certain lookup keyword inputed by the user. As the user types the keyword, the **UserResearch** collection will be

populated by **User** models, fetching them by the server.

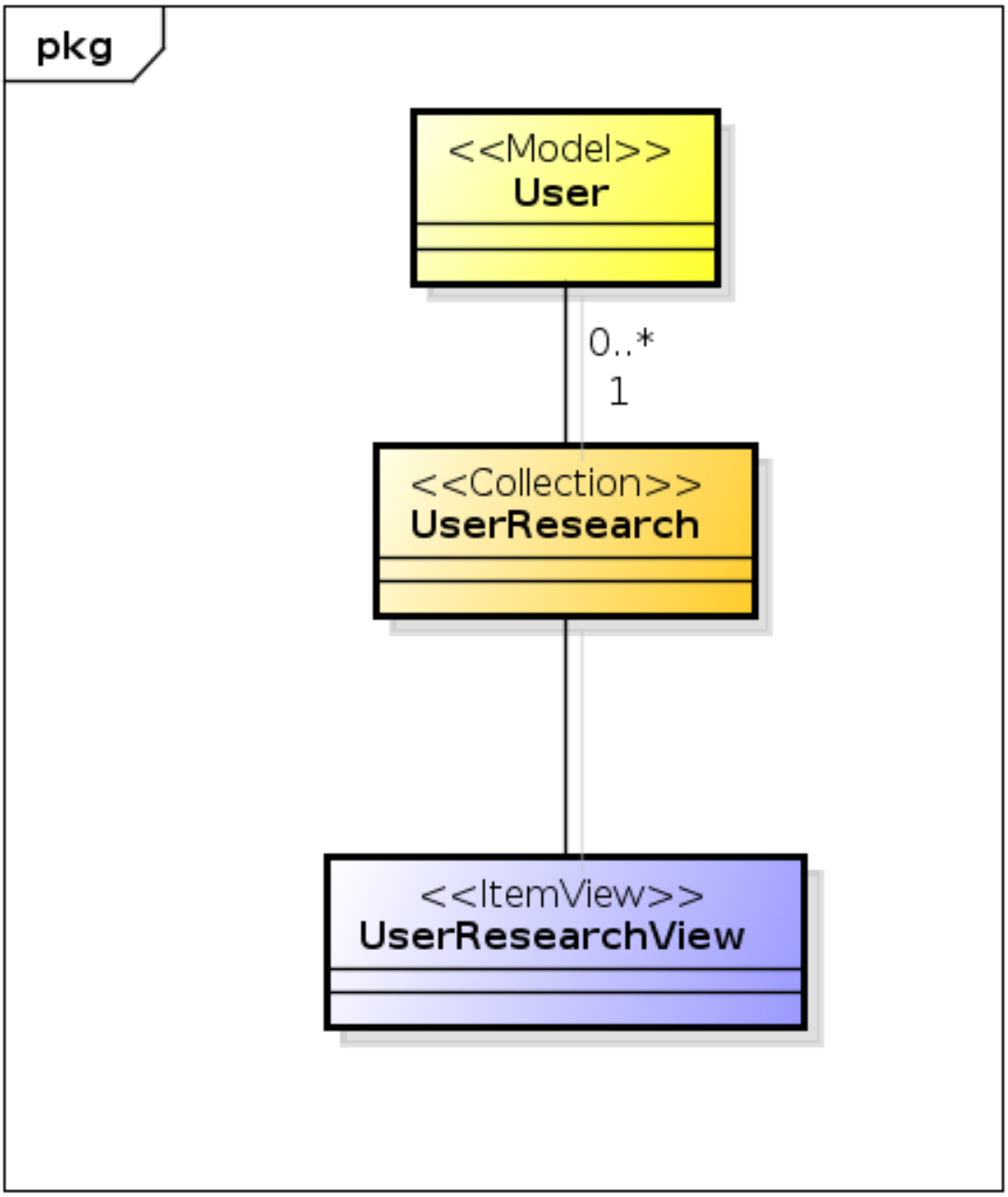

Figure 3.9: Search users

3.2.1.2.5 User's calendar

As the user will have a calendar to browse, data will be fetched asynchronously by intercepting the selection of a new period and rendering the related calendar and the events for that period. This task will be assigned to the model displayed in figure 3.9. The **Calendar** collection will contain the list of **Day** models. The former will be displayed using a *CollectionView* called **CalendarView**, which will be used for rendering every day. The rendering of the **Event** models, collected inside a **DailySchedule** collection will be accomplished by the **DayView**, which is an *ItemView* nested inside the **CalendarView**. It will display the day and iterate over the events and display them inside in an apposite area related to that day.

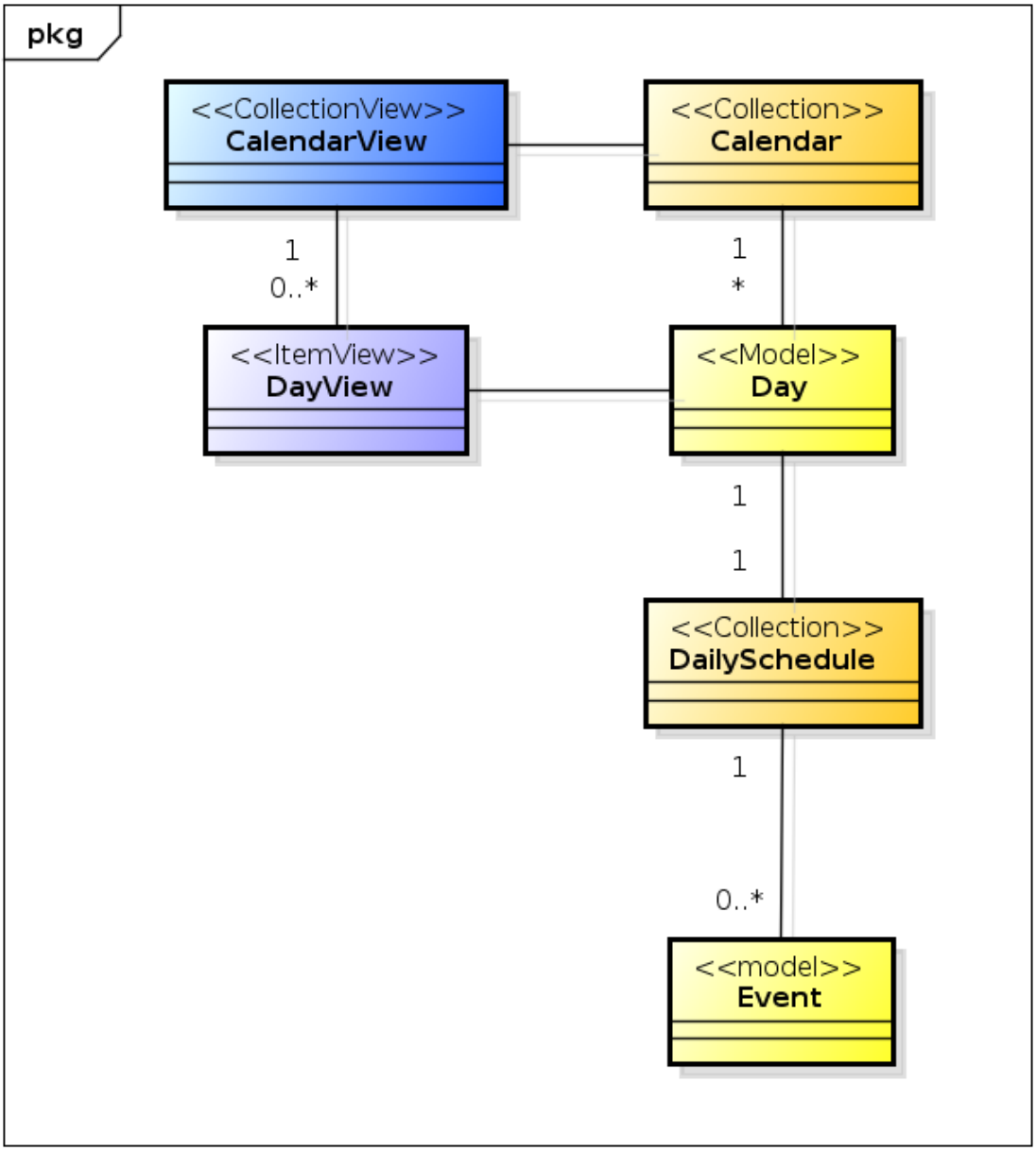

Figure 3.10: Calendar period change

3.2.2 Sequence Diagram

3.2.2.1 Server-side BCE

In this section we'll show some sequence diagram associated with the BCE Diagram explained in 3.2.1.1 section in order to give a more comprehensive overview either of the BCE patterns and of course of our system behavior.

3.2.2.1.1 Sign Up

Figure 3.10 shows the process for the registration of a new user into the system. As a user accesses the system through the MainPage, he will reach either the login or the register form to the system. If the user wants to register this is what happens. The user will submit his information through the form and the SignUpController will check that the correctness of the value. If these information are valid then a new user will be created by the SignUpController and inserted in to the system else if the information are not correct a excepetion is thrown and the registration fails.

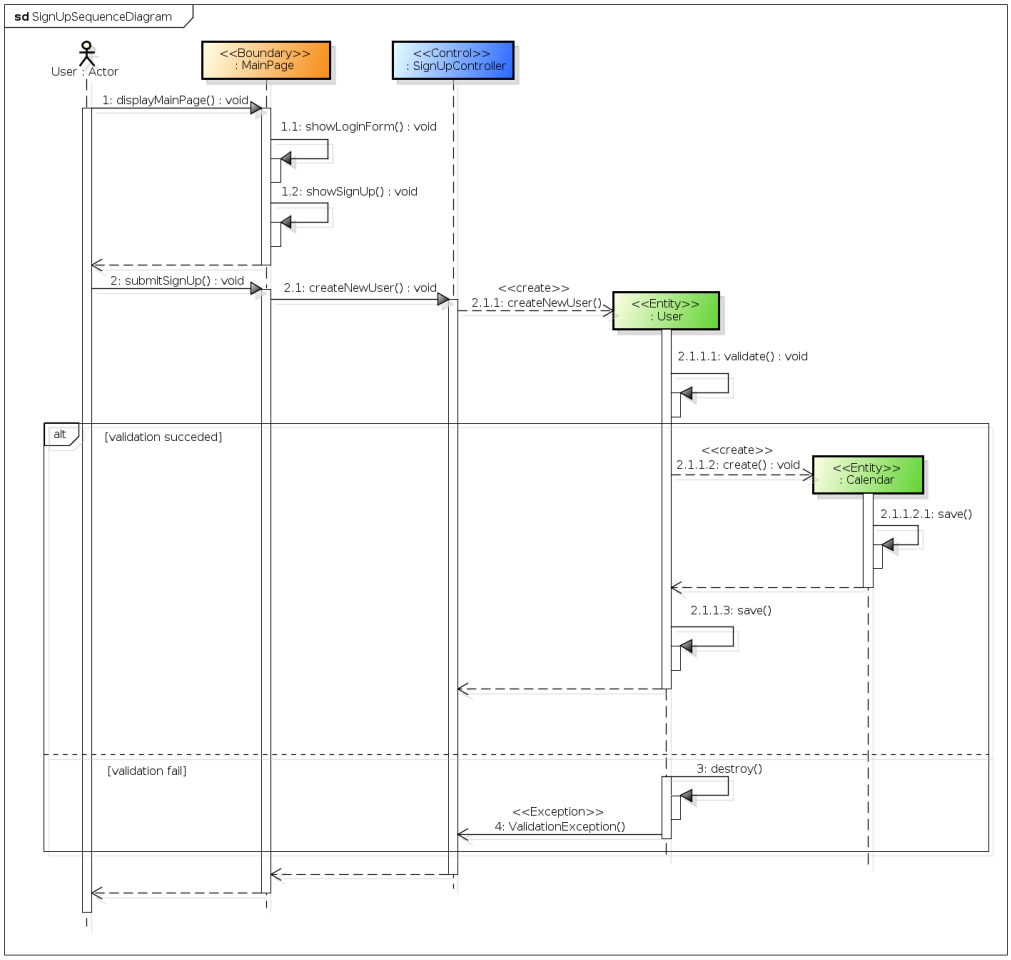

Figure 3.11: Sign up sequence diagram

3.2.2.1.2 Log In

The diagrams represented in figure 3.11 and 3.12 describe the login phase for a registered user in the system. As the user enters the platform the login form is shown to him and the validation process begins. The whole sequence diagram is divided in two part in order to give a better understanding of it.

The first diagram fig. 3.11 refers to the validation of an user. After he submitted his login value, from the log in form in the MainPage, the LoginController verify

the validity of these information, in case of an affermative response then the user is authenticated in to the system and he is redirect to the UserPage where thanks to CalendarController his calendar is loaded and shown.

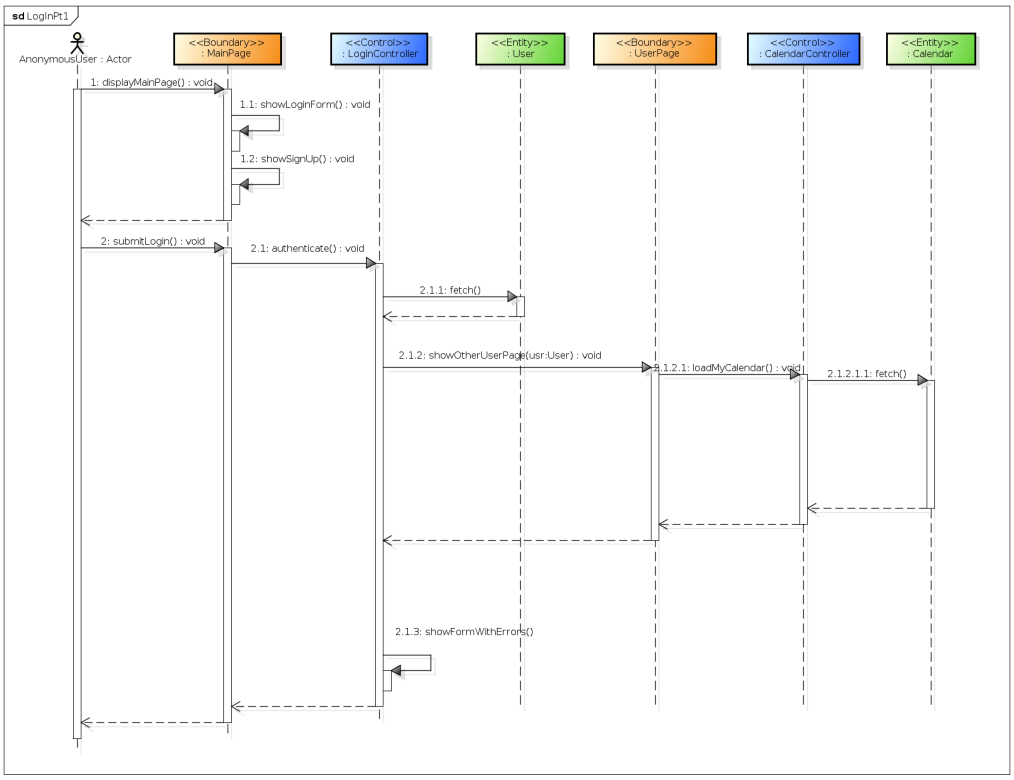

Figure 3.12: Login part 1

So if the diagram 3.11 represents the authentication process of an user then the diagram 3.12 represents the process used by the system to load the event related to the user's calendar. Once the user is logged in, he can see all the events which owns, all the events in which was invited or all the events that is going to attend.

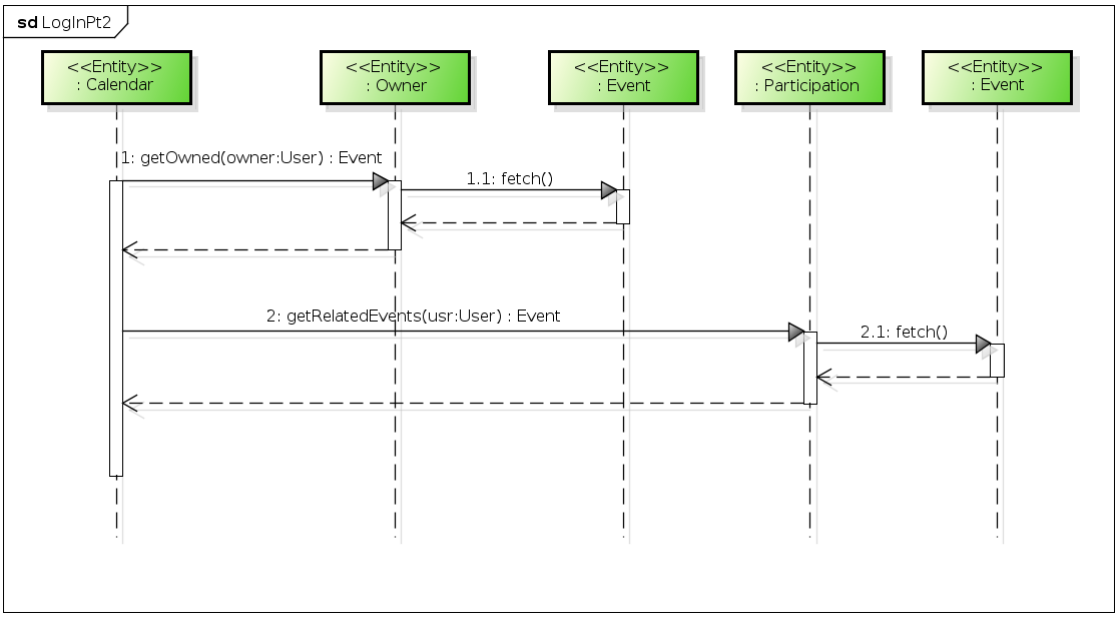

Figure 3.13: Login part 2

3.2.2.1.3 New Event

The diagram represented in figure 3.13 shows how an user will create a new event and customize it as he wants. The NewModifyEventPage will show to him an empty form to fill with the desired value for the event such as the date, the place, the invited user or the weather constraint. The same form is shown to an user that wants to see the event's details and eventually ,if he's also its owner, to modify the current value. Therefore after an user creates or modifies an event, the EventController check that all the values are correct and then either creates a new event or modify an existing one or, if the submitted preferences are not valid, it throws an exception and the creation process fails.

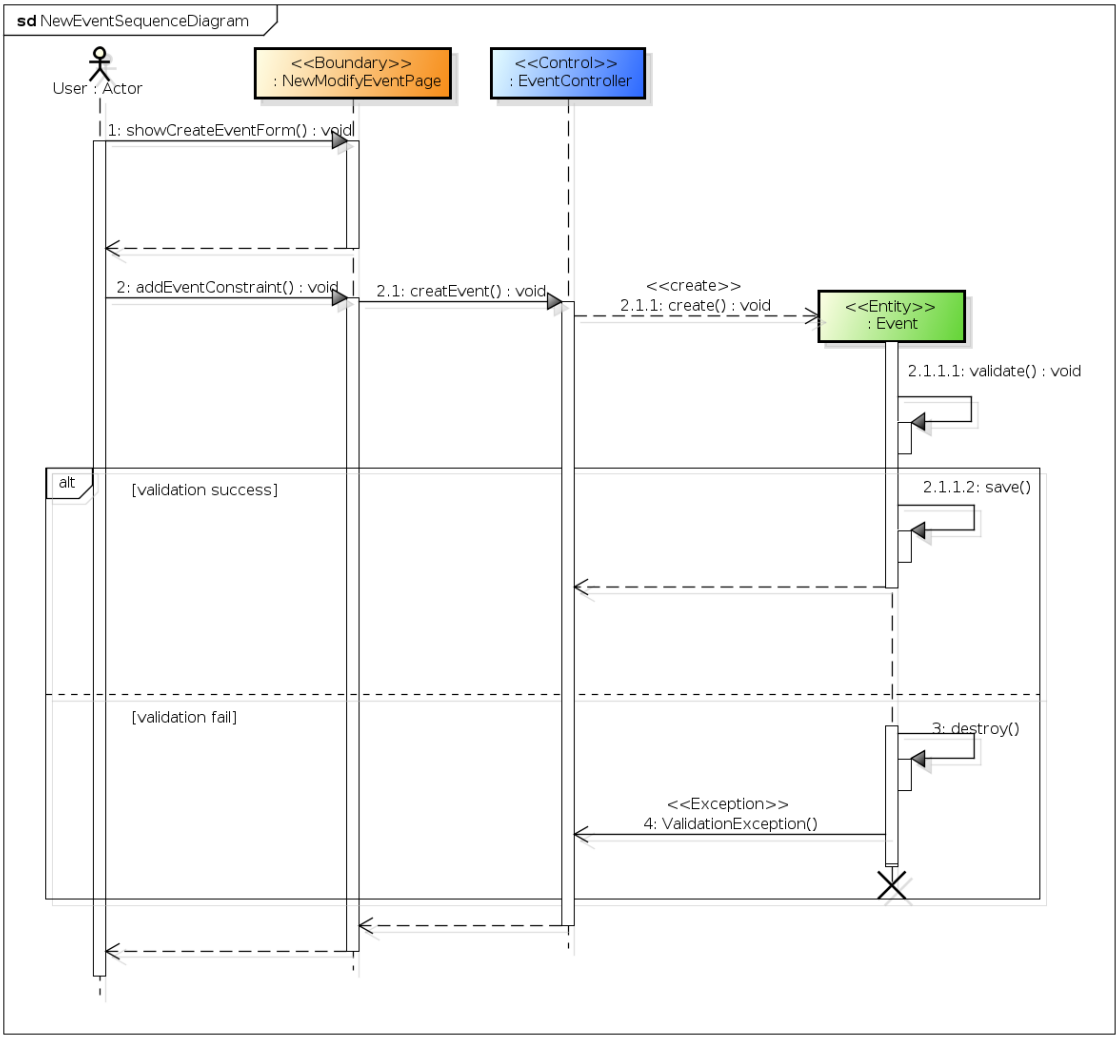

Figure 3.14: New event sequence Diagram

3.2.2.1.4 Search Users

The diagram in figure 3.14 shows the behavior of the system when an user searches for an other user's agenda. The searching and the loading process of the target calendar is accomplished by the CalendarController which once it finds the calendar and makes sure that it was marked visible to everyone by its owner, it fetches all the event related to it, taking care to filter the public event and the private one and

finally shows to the user only the public event of the desired user.

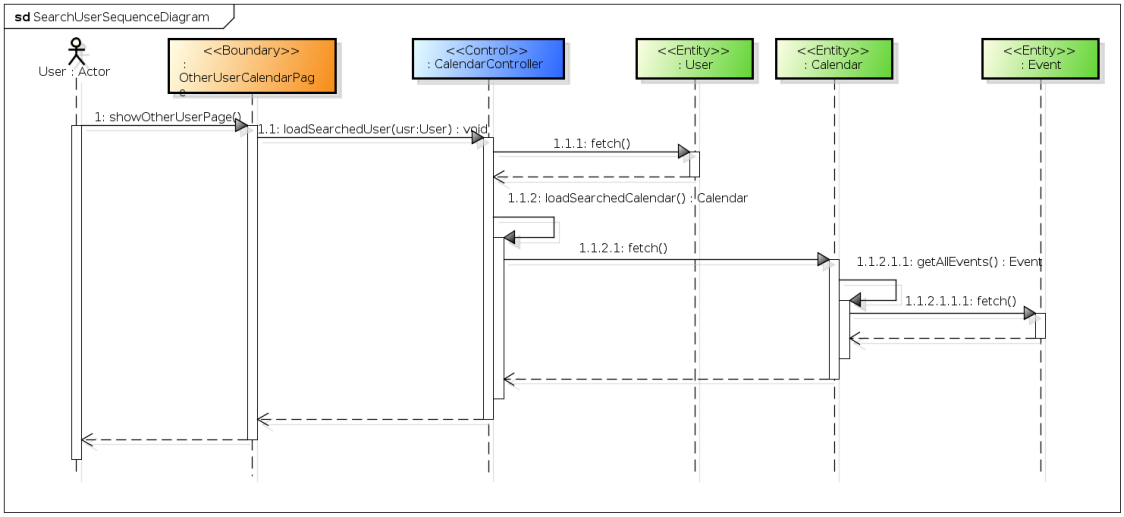

Figure 3.15: Search users sequence diagram

3.2.2.2 Client-side MVC

3.2.2.2.1 Load notifications

The diagram in figure 3.15 shows the behavior of the client application when a user logs onto the platform. As the web page loads, the application will fetch from the server the list of the notifications for that user. After that, each notification is displayed to the user.

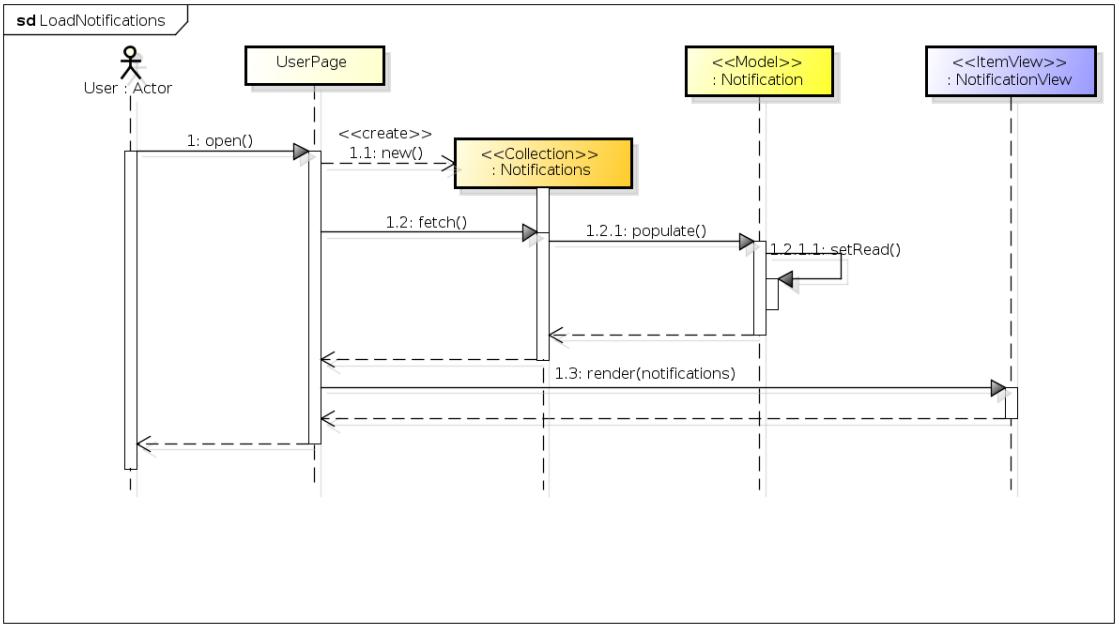

Figure 3.16: Load notifications sequence diagram

3.2.2.2.2 Local forecast

Figure 3.16 illustrates how the local forecast is loaded as the user opens his calendar. The user fetches the forecast for the current locality storing it inside a nested model and collection structure. After this process, this structure is rendered in a dedicated area. The **WeatherDataView** will be involved in displaying the metadata of the forecast, like the locality for which the forecast is shown. This view will also contain the **ForecastView**, a view for displaying the list of all the forecast periods.

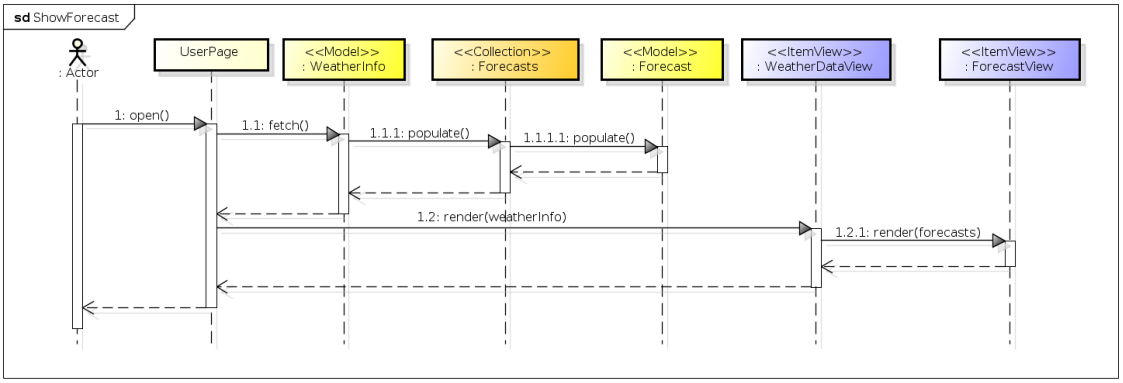

Figure 3.17: Show Forecast sequence diagram

3.2.2.2.3 Search user

3.2.2.2.4 User's calendar

Time reporting

	Paolo Polidori	**Marco Edemanti**
RASD writing	19 hours	19 hours

List of Figures