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 ladder 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 <NAME OF DBMS> 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

<Identify the product whose software requirements are specified in this document, including the revision or release number. Describe the scope of the product that is covered by this SRS, particularly if this SRS describes only part of the system or a single subsystem.>

3.2 MVC 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 MVC structure

3.2.2 MVC behaviour

3.2.3 BCE Diagrams

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.

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.

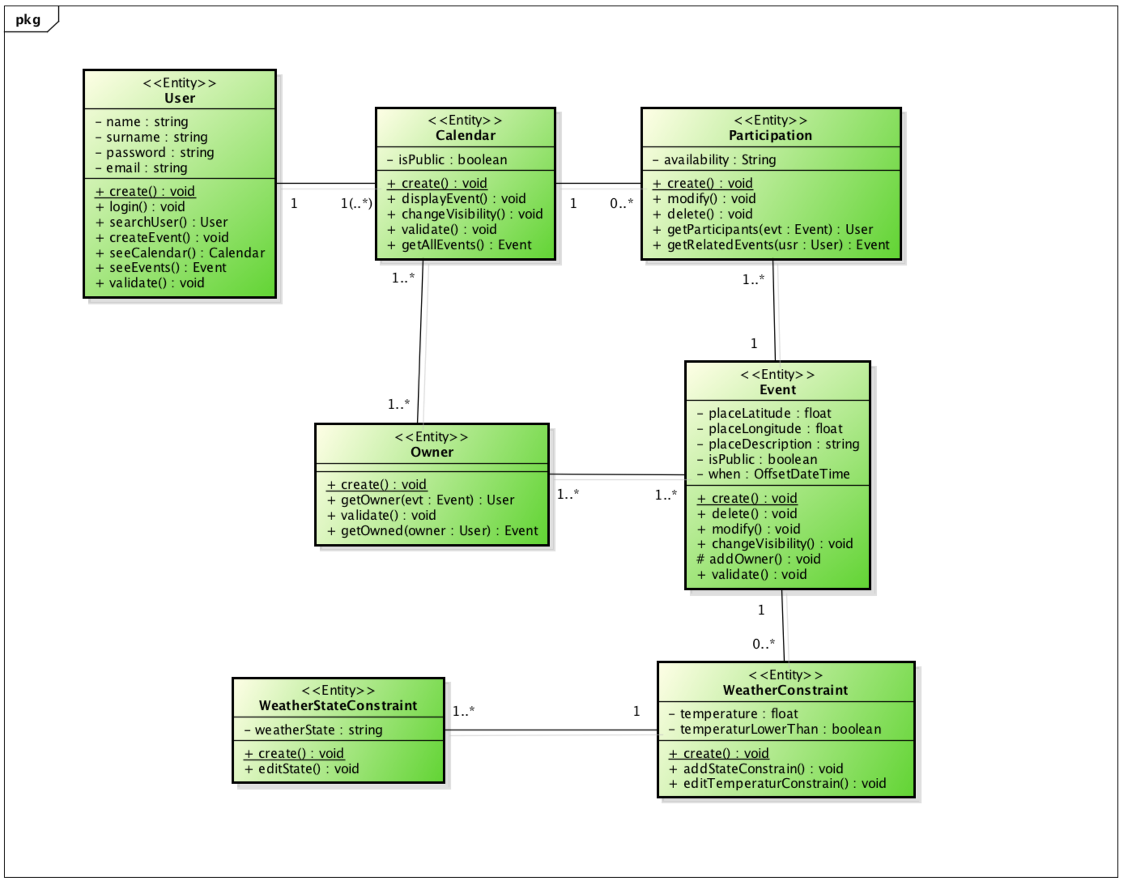

Figure 3.1: Entities involved

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.

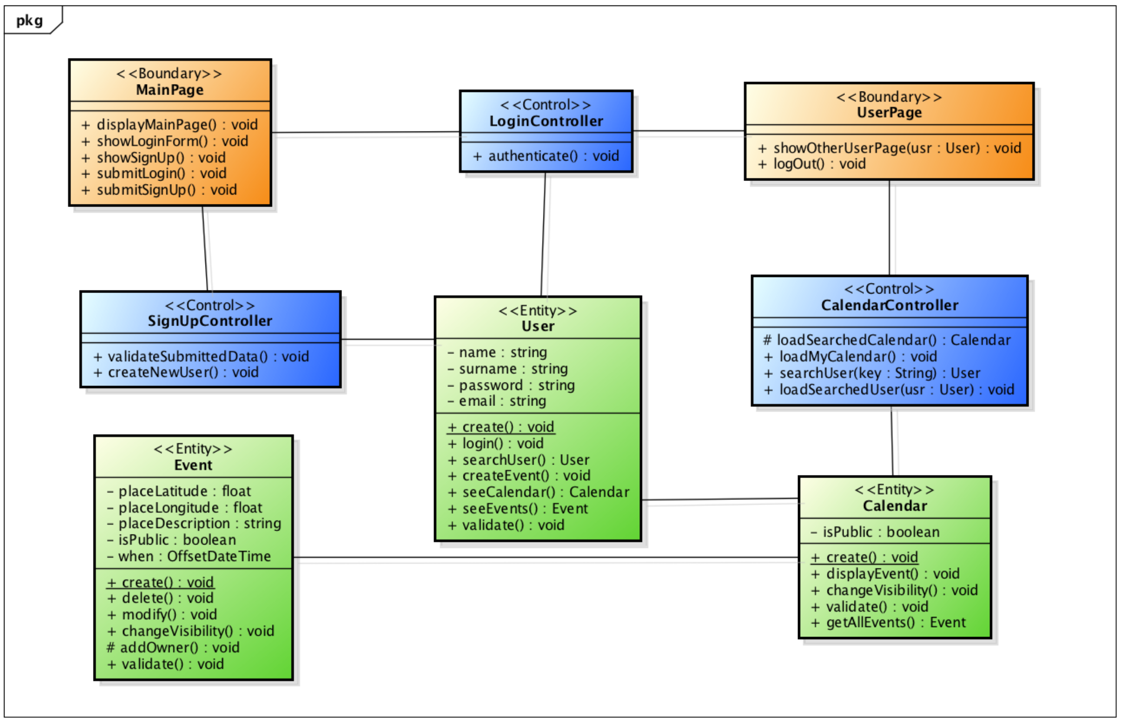

Figure 3.2: Sign Up and Log In

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.

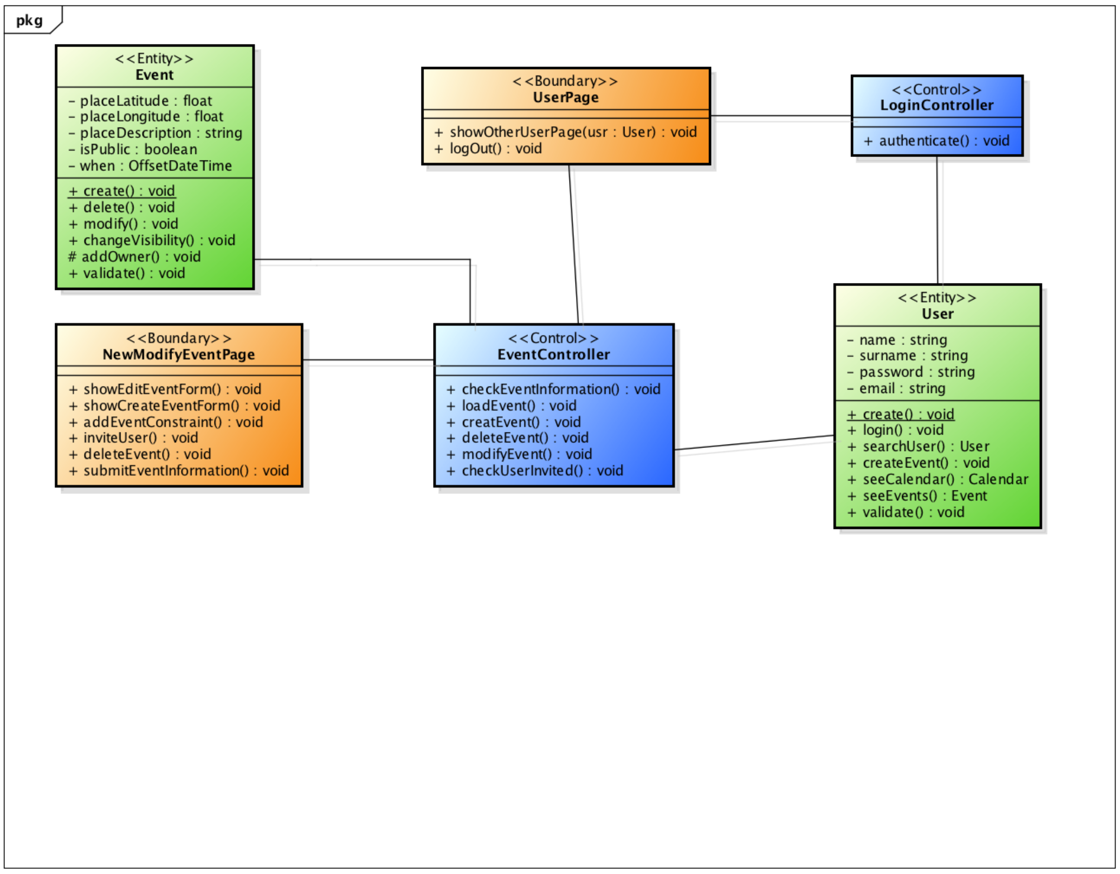

Figure 3.3: Create or modify an event

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.

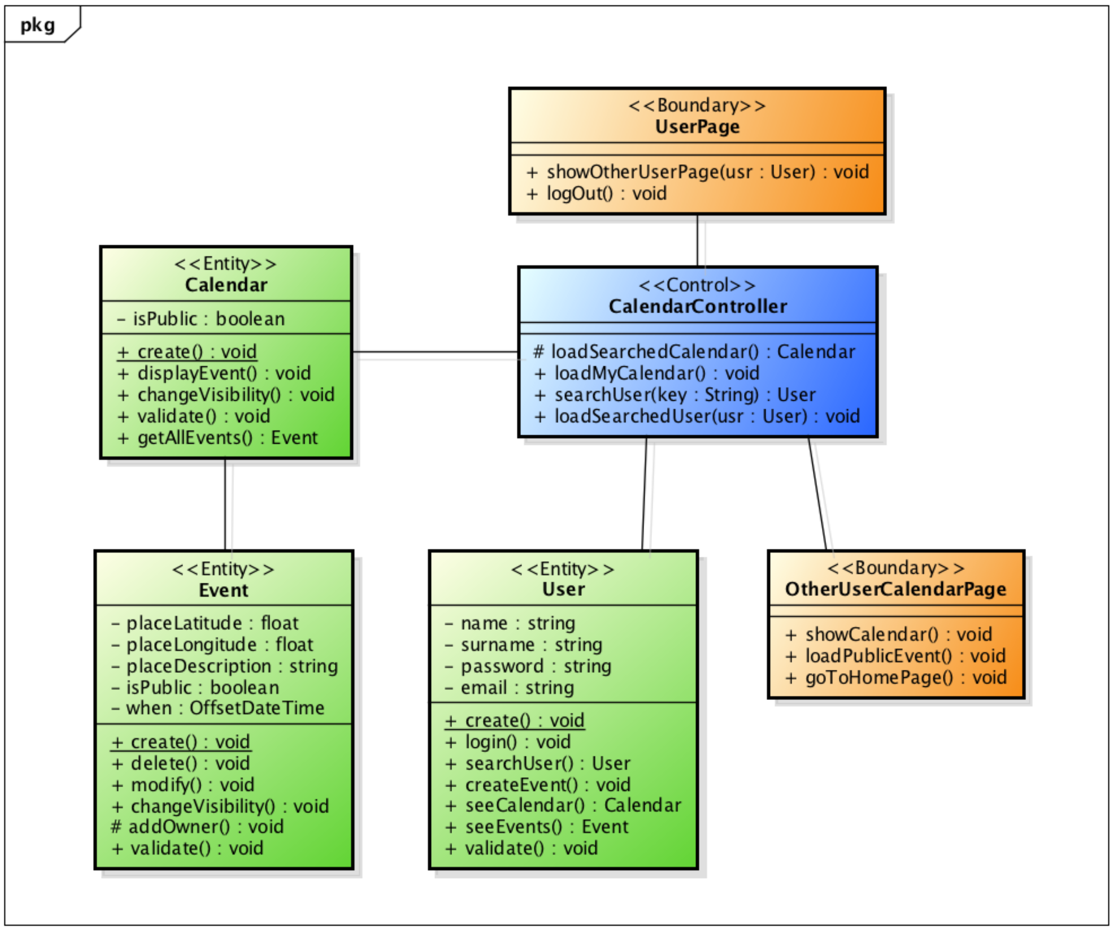

Figure 3.4: Search users

3.3 Sequence Diagram

In this section we'll show some sequence diagram associated with the BCE Diagram explained in 3.2.3 section in order to give a more comprehensive overview either of the BCE patterns and of course of our system behavior.

3.3.1 Sign Up

Figure 3.5 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.

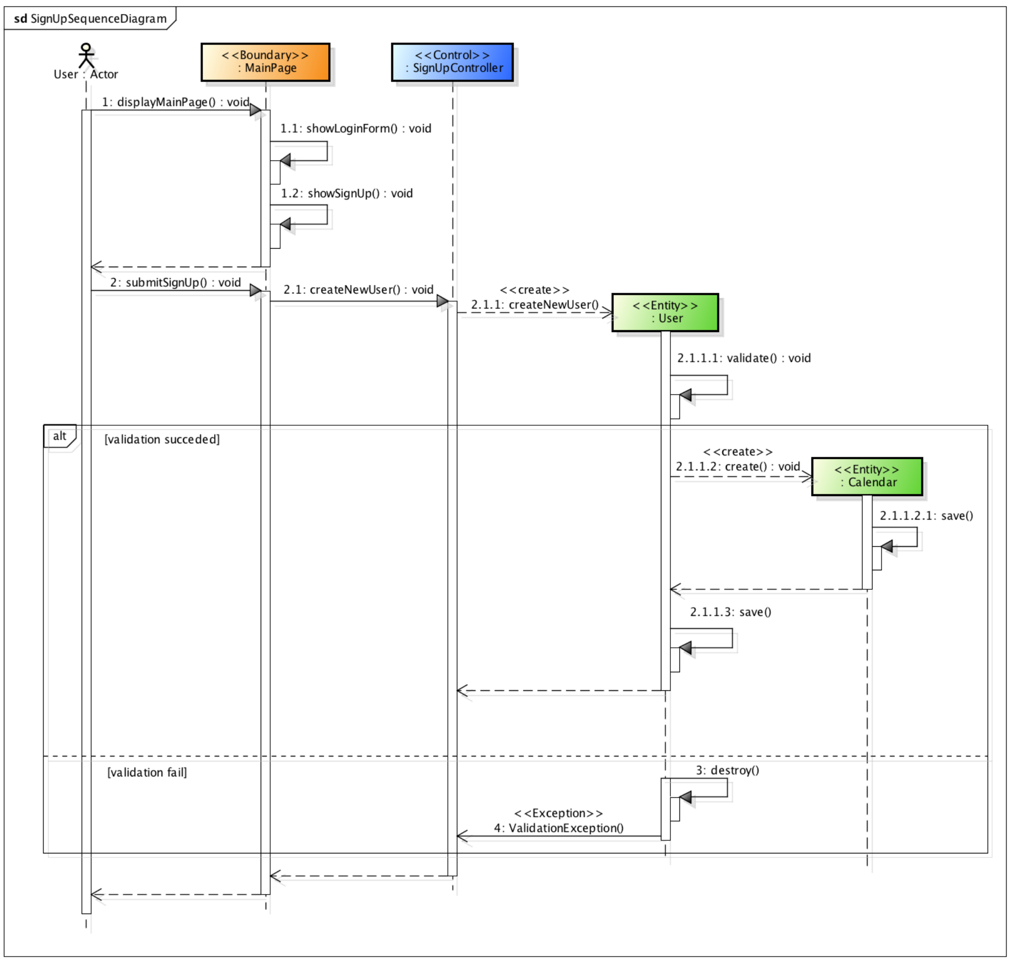
sd SignUpSequenceDiagram
User : Actor
<<Boundary>> : MainPage
<<Control>> : SignUpController
1: displayMainPage() : void
1.1: showLoginForm() : void
1.2: showSignUp() : void
2: submitSignUp() : void
2.1: createNewUser() : void
<<create>> 2.1.1: createNewUser()
<<Entity>> : User
2.1.1.1: validate() : void
alt
[validation succeded]
<<create>> 2.1.1.2: create() : void
<<Entity>> : Calendar
2.1.1.2.1: save()
2.1.1.3: save()
[validation fail]
3: destroy()
<<Exception>> 4: ValidationException()

Figure 3.5: Sign up sequence diagram

3.3.2 Log In

The diagrams represented in figure 3.6 and 3.7 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. 3.6 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.

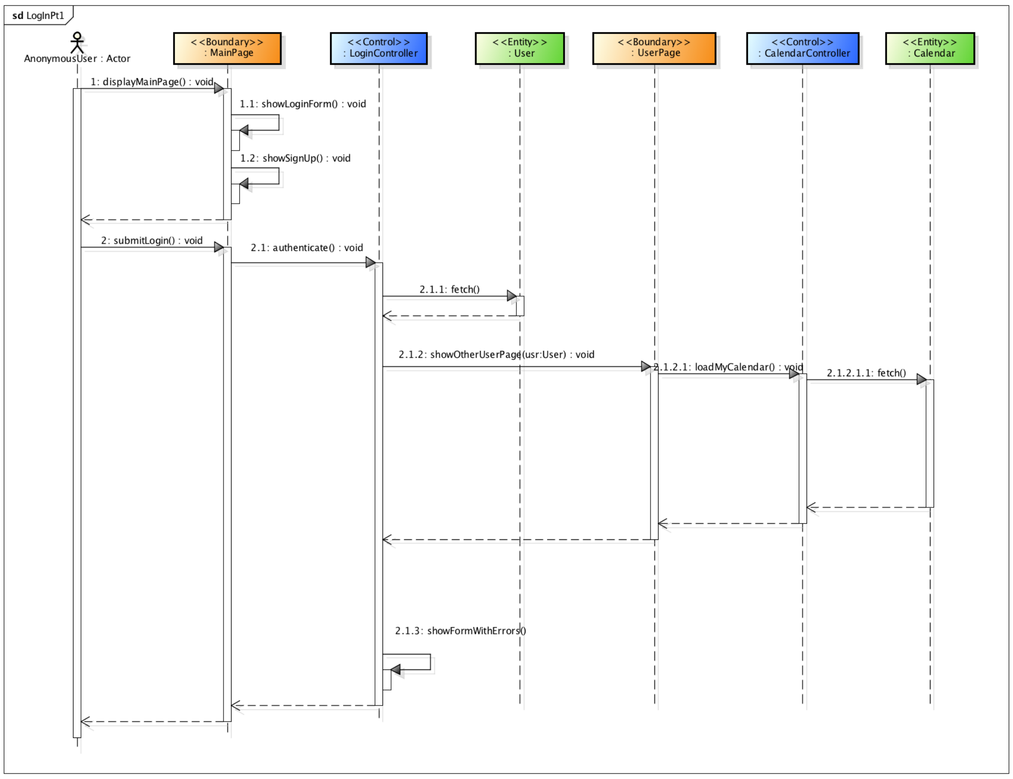
sd LogInPt1
AnonymousUser : Actor
<<Boundary>> : MainPage
<<Control>> : LoginController
<<Entity>> : User
<<Boundary>> : UserPage
<<Control>> : CalendarController
<<Entity>> : Calendar
1: displayMainPage() : void
1.1: showLoginForm() : void
1.2: showSignUp() : void
2: submitLogin() : void
2.1: authenticate() : void
2.1.1: fetch()
2.1.2: showOtherUserPage(usr:User) : void
2.1.2.1: loadMyCalendar() : void
2.1.2.1.1: fetch()
2.1.3: showFormWithErrors()

Figure 3.6: Login part 1

So if the diagram 3.6 represents the authentication process of an user then the diagram 3.7 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.

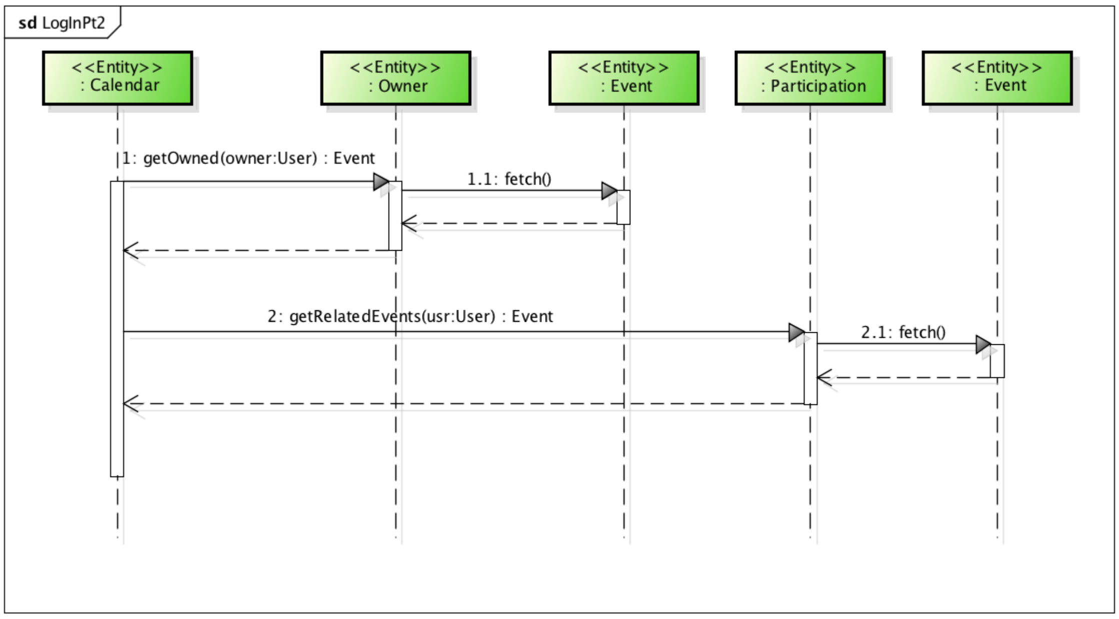

Figure 3.7: Login part 2

3.3.3 New Event

The diagram represented in figure 3.8 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.

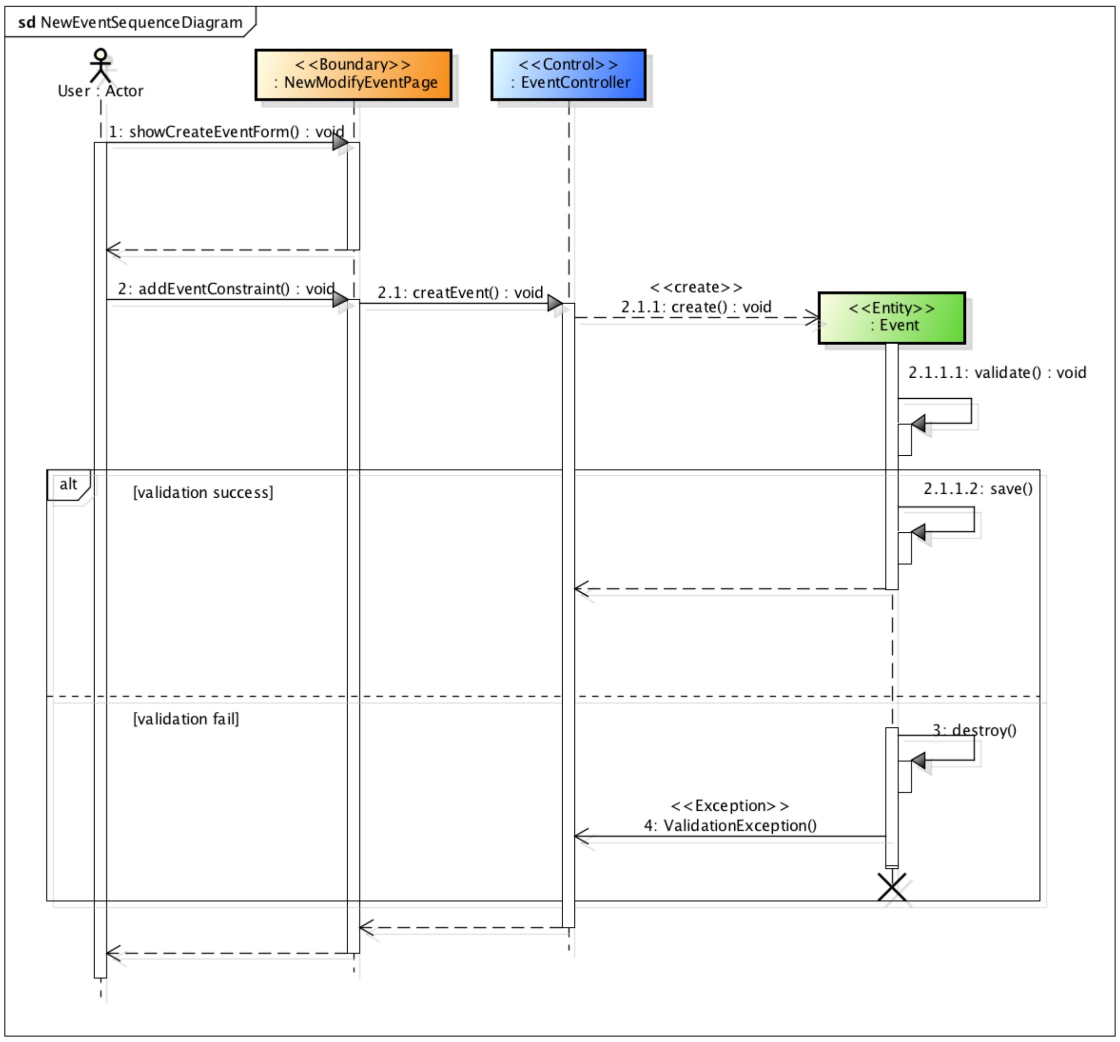
sd NewEventSequenceDiagram
User : Actor
<<Boundary>> : NewModifyEventPage
<<Control>> : EventController
1: showCreateEventForm() : void
2: addEventConstraint() : void
2.1: creatEvent() : void
<<create>> 2.1.1: create() : void
<<Entity>> : Event
2.1.1.1: validate() : void
alt
[validation success]
2.1.1.2: save()
[validation fail]
3: destroy()
<<Exception>> 4: ValidationException()

Figure 3.8: New event sequence Diagram

3.3.4 Search Users

The diagram in figure 3.9 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.

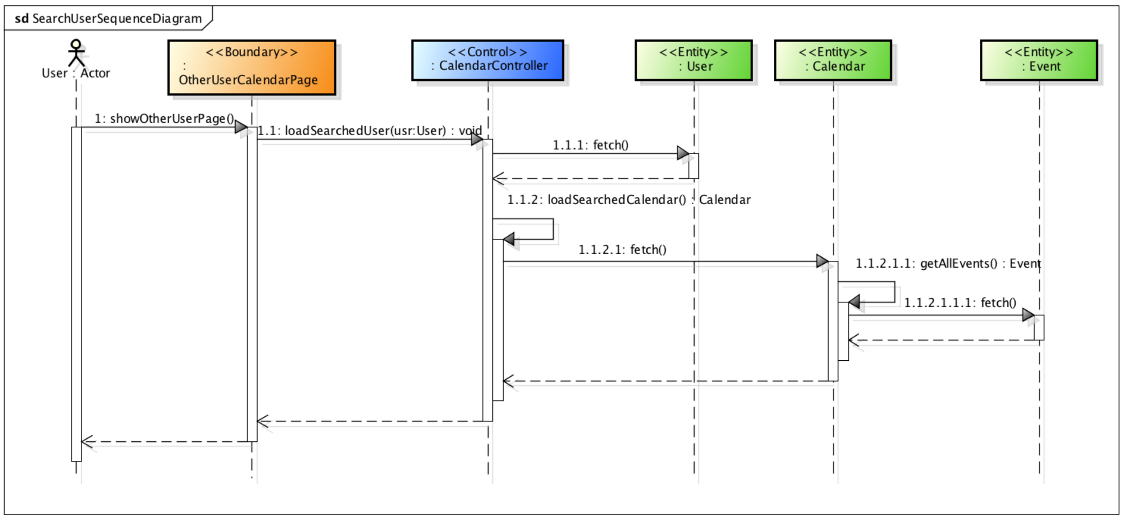
sd SearchUserSequenceDiagram
User : Actor
<<Boundary>> : OtherUserCalendarPage
<<Control>> : CalendarController
<<Entity>> : User
<<Entity>> : Calendar
<<Entity>> : Event
1: showOtherUserPage()
1.1: loadSearchedUser(usr:User) : void
1.1.1: fetch()
1.1.2: loadSearchedCalendar() : Calendar
1.1.2.1: fetch()
1.1.2.1.1: getAllEvents() : Event
1.1.2.1.1.1: fetch()

Figure 3.9: Search users sequence diagram

Time Reporting

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

List of Figures

Listings