Politecnico di Milano

5th School of Engineering

Software Engineering Project

MeteoCal

Design Document

Nemanja Stolic 814842

Milos Colic 814817

Milan, 5th December, 2014

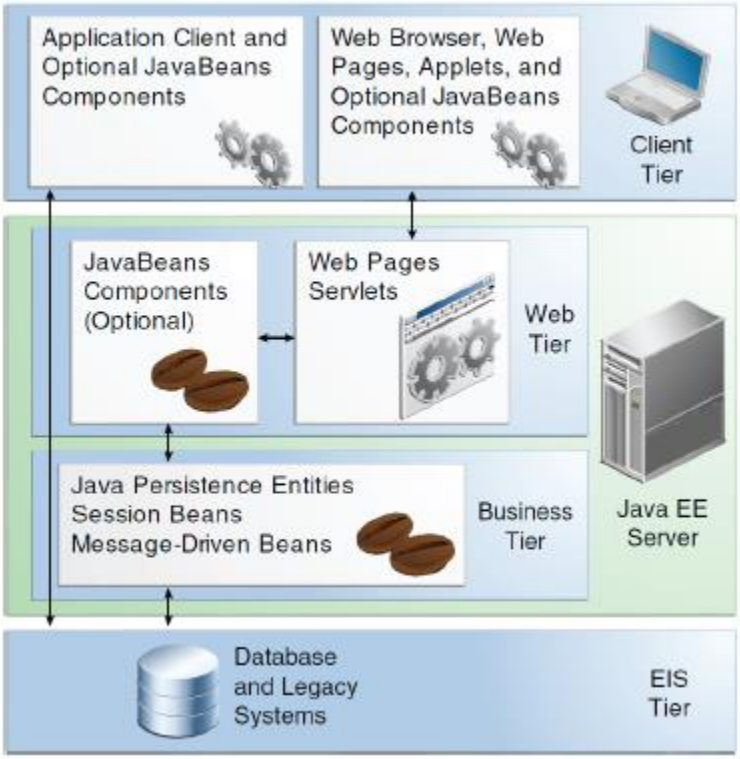
Contents

Milan, 5th December, 2014

1. ARCHITECTURE DESCRIPTION

1.1. JEE ARCHITECTURE OVERVIEW

Before starting to explain our application's architecture we want to focus on JEE Architecture.



JEE has a four tiered architecture divided as:

- **Client Tier**: it contains Application Clients and Web Browsers and it is the layer that interacts directly with the actors. As our project will be a web application the client will use a web browser to access pages;
- **Web Tier**: it contains the Servlets and Dynamic Web Pages that needs to be elaborated. This tier receives the requests from the client tier and forwards the pieces of data collected to the business tier waiting for processed data to be sent to the client tier, eventually formatted;
- **Business Tier**: it contains the Java Beans, which contain the business logic of the application, and Java Persistence Entities.
- **EIS Tier**: it contains the data source. In our case it is the database allowed to store all the relevant data and to retrieve them.

1.2. IDENTIFYING SUB-SYSTEMS

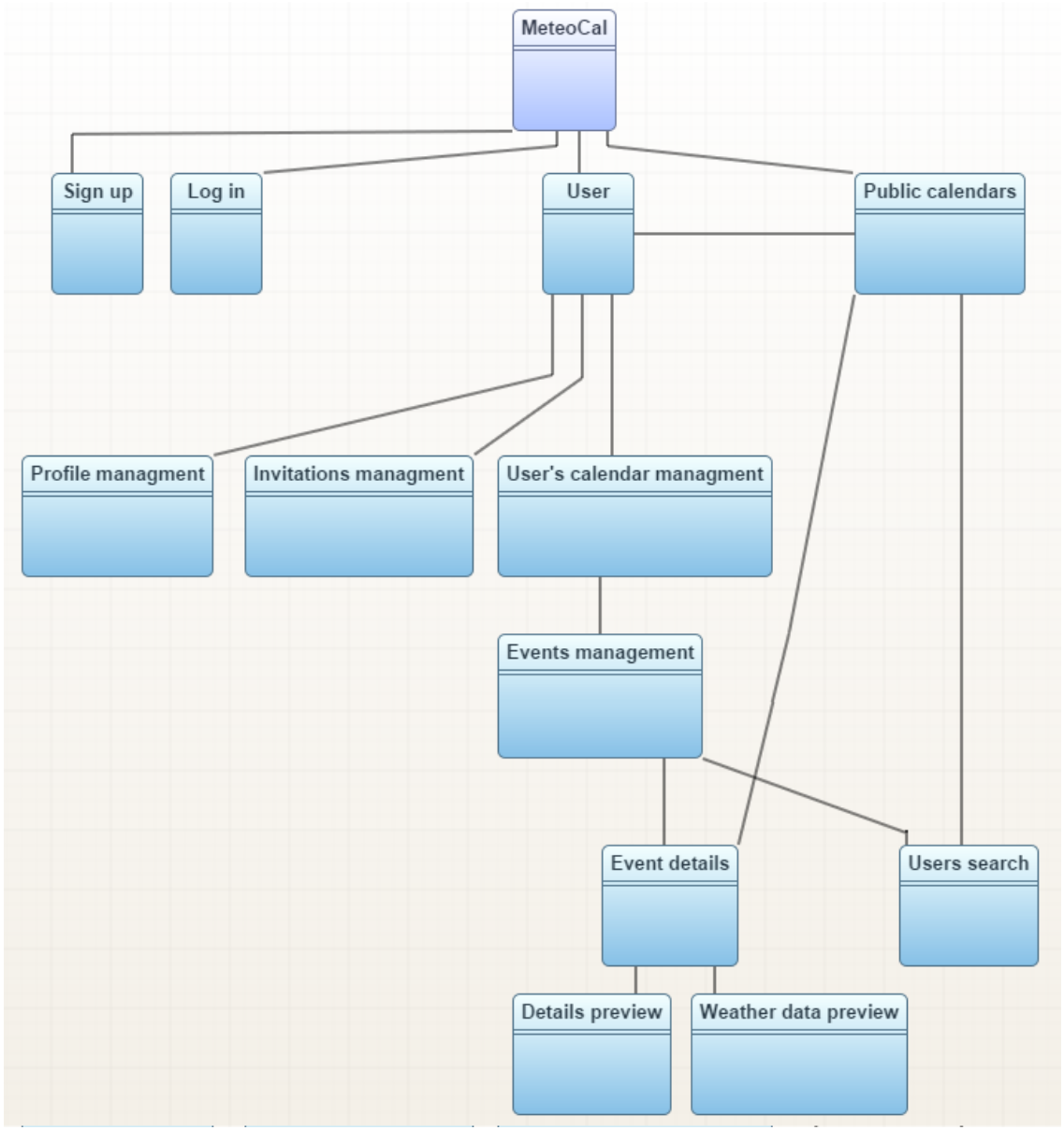
We decided to adopt a top-down approach at least at this point of the project. Maybe, once defined the sub-systems, we will adopt a bottom-up approach in order to create more reusable components.

So we think it is now necessary to decompose our system into other sub-systems, in order to make it easy to understand the issues that we found in implementing functionalities and to separate, logically, groups of functionalities and state clearer their interaction.

We separate our systems into these sub-systems:

- Sign up subsystem;
- Log in subsystem;
- User subsystem;
    - Profile management subsystem;
    - Invitations management subsystem;
    - User's calendar management subsystem;
- Public calendars subsystem;

Milan, 5th December, 2014



Milan, 5th December, 2014

2. PERSISTENT DATA MANAGEMENT

Our data is stored into a relational database. Database design represented by Entity-Relationship Diagram can be found in the subsection below. Moreover, we will explain in details entities, relations and provide the description for specific parts of each design diagram.

2.1. CONCEPTUAL DESIGN

Conceptual design allows us to start thinking about the data we want to store and about the relations between them.

The most important entity in our system is a *User*. Regular visitor, after completing the procedure of signing up becomes the user of a system. To each *User*, an entity *Calendar* is assigned from the start. Thus one *User* has one *Calendar*, and one *Calendar* has one owner of type *User*. *One-to-one* relation (and also other relations) are presented later in the diagram using Crow's foot notation.

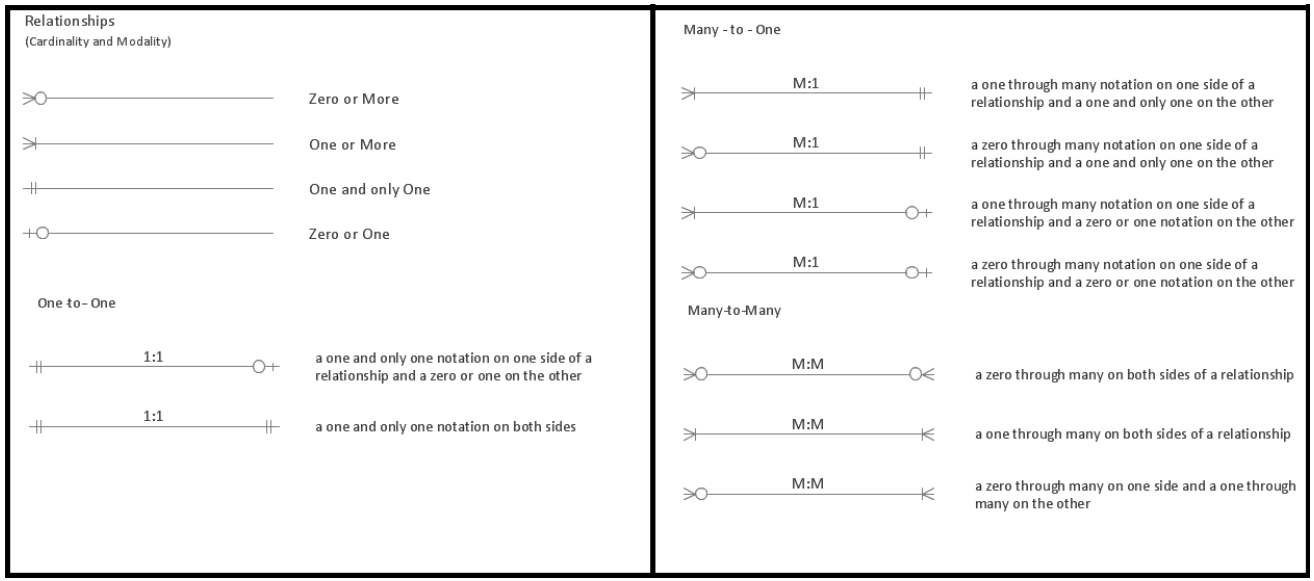
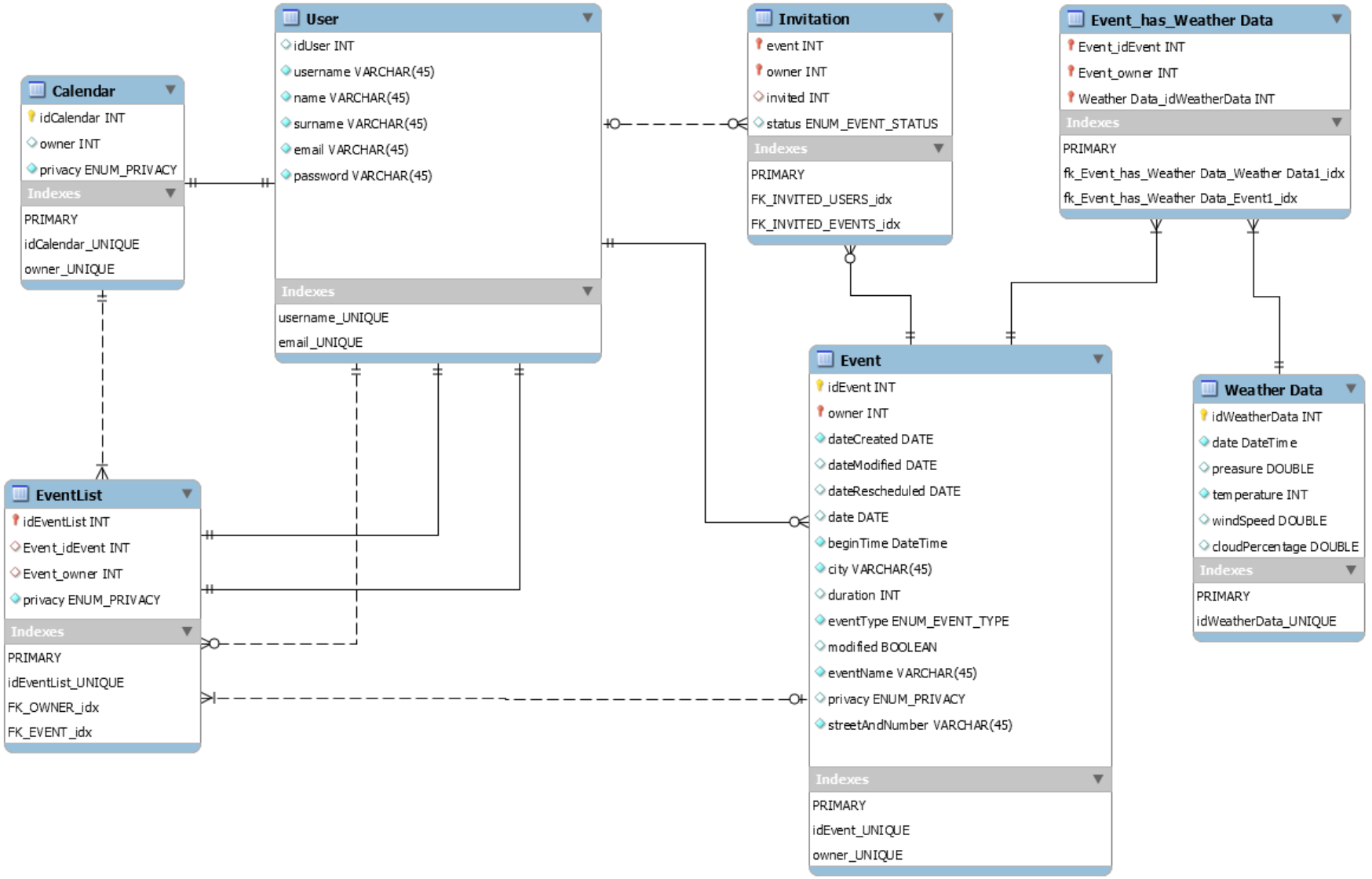


Figure: Crow's foot notation

Milan, 9th November, 2014

In the following diagram our system's conceptual design is presented:



Milan, 9th November, 2014

Each Calendar entity has two *Event List*s entities. One *Event List* is the container for owner's events, and other serves for holding events to which user participates. Hence, the relation between a *Calendar* and an *Event List* is *One-to-many*.

User holds information about events which he owns and to which he is participating. *User* must have two *Event List*s, even if they are empty. Each of the lists must have unique *User* associated to it. We added two *One-to-one* relations and not one of type *One-to-Many*. The reason is that we wanted exactly two *Event List*s to be implemented.

User may be an owner of an *Event*, or many, or none of them. But each *Event* must have exactly one owner of type *User*. This relation is represented in the diagram as *One-to-many* relation between a *User* and an *Event*.

User may also receive many *Invitations* to *Event*s, and *Event*s may have many *Invitation*s for *User*s. This relation is of type *Many-to-many*, and it is modelled in indirect way, like it is usually done in some popular implementations: via *One-to-Many* and *Many-to-one* relation - (User) *One-to-many* (Invitation) *Many-to-one* (Event).

User may participate in many *Event*s, and *Event*s may have many participants. In similar fashion, like in last example, *Many-to-many* relation is implemented as follows: (User) *One-to-many* (EventList) *Many-to-one* (Event).

Similarly, for one Event many Weather data (for many hours) can be provided, and also one Weather data may correspond to many Events (if they are overlapping in time). *Many-to-many* relation is implemented as follows:
(Event) *One-to-many* (Event_has_Weather Data) *Many-to-one* (Weather Data).

2.2. LOGICAL DESIGN

Logical Design has the aim to better represent the database structure of our system, but, in order to build this model from the ER diagram drawn above, we have to perform some transformations.

2.2.1. TRANSLATION TO LOGICAL MODEL

1. User table: Relation 'Invited' is a n:m relation so it is modeled trough the Invitation table, relation between User and Invitation table is 1:n relation that maps idUser(User:PK)->invited(Invitatio:FK). Relation 'Owns' is a 1:n relation that maps idUser(User:PK)->owner(Event:FK). Relation 'OwnedEvents'

is a 1:1 relation that maps idUser(User:PK)->idEventList(EventList:PK). Relatiom ‘OtherEvents’ is a 1:1 relation that maps idUser(User:PK)->idEventList(EventList:PK). 1:n relation between User and EventList table is a part of a n:m relation which is modeled through EventList table, it maps idUser(User:PK)->Event_Owner(EventList:FK). Relation ‘CalendarOwner’ is a 1:1 relation that maps idUser(User:PK)->idCalendar(Calendar:PK).

2. Calendar table: Relation ‘AppearsIn’ is a 1:n relation that maps idCalendar(Calendar:PK) + owner(Calendar:FK) -> Event_idEvent(EventList:FK) + Event_owner(EventList:FK) which is used to map events owned by multiple users that are a part of a public calendars and are also themselves marked as public.
3. Event table: Relation 1:n between Event and Invitation is a part of n:m relation that connects users and events through invitations. It maps (event+owner)(Invitation:CK)->idEvent(Event:PK) + owner(Event:FK). Relation 1:n between Event and Event list is a part of n:m relation that maps users and events they are participants of. It maps idEvent(Event:PK) + owner(Event:FK) -> Event_Owner(EventList:FK) + idEventList(EventList:PK). Relation between Event and Event_Has_WeatherData is a part of n:m relation between Event and WeatherData, it maps idEvent(Event:PK) + owner(Event:FK) -> (Event_idEvent+Event_owner)(Event_Has_WeatherData:CK).

The final model has the following physical structure:

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI
idUser	INT	☐	☐	☐	☐	☐	☐	☑
username	VARCHAR(45)	☐	☑	☑	☐	☐	☐	☐
name	VARCHAR(45)	☐	☑	☐	☐	☐	☐	☐
surname	VARCHAR(45)	☐	☑	☐	☐	☐	☐	☐
email	VARCHAR(45)	☐	☑	☑	☐	☐	☐	☐
password	VARCHAR(45)	☐	☑	☐	☐	☐	☐	☐

Figure 1: User Table

Milan, 9th November, 2014

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI
idCalendar	INT	☑	☑	☑	☐	☐	☐	☑
owner	INT	☐	☐	☑	☐	☐	☐	☐
privacy	ENUM_PRIVACY	☐	☑	☐	☐	☐	☐	☐
		☐	☐	☐	☐	☐	☐	☐

Figure 2: Calendars Table

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI
idEvent	INT	☑	☑	☑	☐	☐	☐	☑
owner	INT	☑	☑	☑	☐	☐	☐	☐
dateCreated	DATE	☐	☑	☐	☐	☐	☐	☐
dateModified	DATE	☐	☐	☐	☐	☐	☐	☐
dateRescheduled	DATE	☐	☐	☐	☐	☐	☐	☐
date	DATE	☐	☐	☐	☐	☐	☐	☐
beginTime	DateTime	☐	☑	☐	☐	☐	☐	☐
city	VARCHAR(45)	☐	☑	☐	☐	☐	☐	☐
duration	INT	☐	☐	☐	☐	☐	☐	☐
eventType	ENUM_EVENT_TYPE	☐	☑	☐	☐	☐	☐	☐
modified	BOOLEAN	☐	☐	☐	☐	☐	☐	☐
eventName	VARCHAR(45)	☐	☑	☐	☐	☐	☐	☐
privacy	ENUM_PRIVACY	☐	☐	☐	☐	☐	☐	☐
streetAndNumber	VARCHAR(45)	☐	☑	☐	☐	☐	☐	☐

Figure 3: Event Table

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI
idEventList	INT	☑	☑	☑	☐	☐	☐	☑
Event_idEvent	INT	☐	☐	☐	☐	☐	☐	☐
Event_owner	INT	☐	☐	☐	☐	☐	☐	☐
privacy	ENUM_PRIVACY	☐	☑	☐	☐	☐	☐	☐

Figure 4: EventList Table

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI
event	INT	☑	☑	☐	☐	☐	☐	☐
owner	INT	☑	☑	☐	☐	☐	☐	☐
invited	INT	☐	☐	☐	☐	☐	☐	☐
status	ENUM_EVENT_STATUS	☐	☐	☐	☐	☐	☐	☐

Figure 5: Invitation Table

Milan, 9th November, 2014

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI
idWeatherData	INT	☑	☑	☑	☐	☐	☐	☐
date	DateTime	☐	☑	☐	☐	☐	☐	☐
preasure	DOUBLE	☐	☐	☐	☐	☐	☐	☐
temperature	INT	☐	☑	☐	☐	☐	☐	☐
windSpeed	DOUBLE	☐	☐	☐	☐	☐	☐	☐
cloudPercentage	DOUBLE	☐	☐	☐	☐	☐	☐	☐

Figure 6: WeatherData Table

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI
Event_idEvent	INT	☑	☑	☐	☐	☐	☐	☐
Event_owner	INT	☑	☑	☐	☐	☐	☐	☐
Weather Data_idWeatherData	INT	☑	☑	☐	☐	☐	☐	☐

Figure 7: Event_Has_WeatherData Table

3. USER EXPERIENCE

In this section we want to describe User Experience (UX) given by our system to its users. We used a Class Diagram with appropriate stereotypes *<<screen>>*, *<<screen compartment>>* and *<<input form>>*s and regular Classes to let a reader understand how the User Experience was designed. While *<<screen>>* represents pages, *<<screen compartment>>* represents part of the page that can be also implemented in other pages. *<<input form>>* represents some input fields that can be filled by a user (this information will be submitted to the system clicking on a button).

Also, we decided to split the UX Diagram in functionalities as for Use Cases in the RASD Document in order to better understand the whole Diagram.

Each of the sections below is entitled with the name of the functionality represented by the UX Diagram drawn.

3.1. LOGIN, SIGN UP AND CHANGE PASSWORD

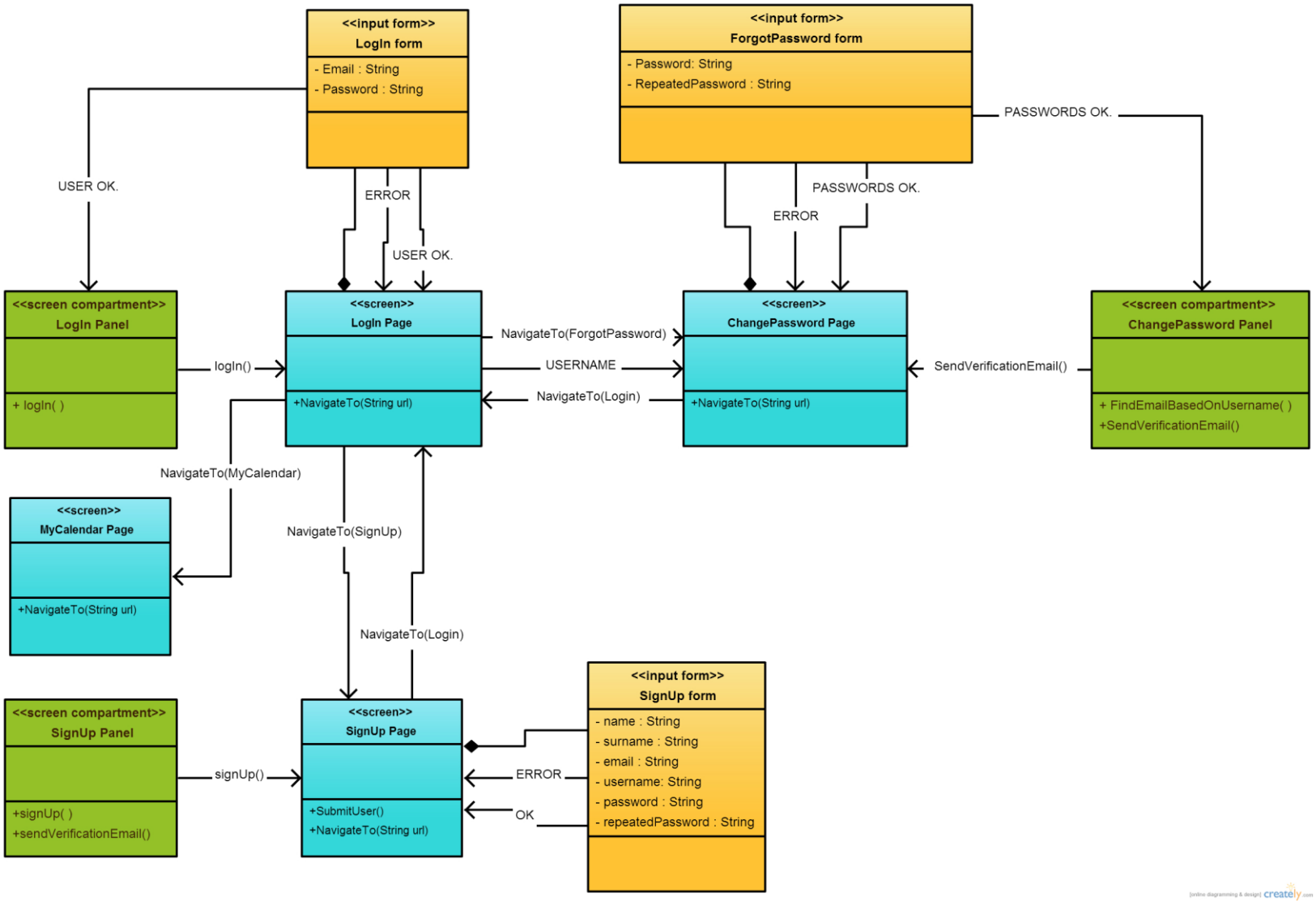
We can see below a Diagram that consists of the different pages which have a purpose of ensuring secure access to the system and authentication. Based on the type of visitor that is interacting with the system, the visitor can have limited access which we will describe briefly bellow.

The homepage of our system is *LogIn Page*. If the visitor does not have an account, he/she can chose to navigate to screen *SignUp Page* by clicking on Sign up button inside LogIn Page. If the visitor does not have an account it is not possible for him/her to access the system. When a user visits the page and submits all necessary data required for signing up into *SignUp form*, the *SignUp Pannel* will invoke *signUp()* function. If there is an error SignUp form will show the error on screen *SignUp Page*. In everything is ok, the system will execute *sendVerificationEmail()* inside *signUp()* function, and the user will be notified about successful signing up. Now the user can decide to navigate to *LogIn Page* by clicking on an appropriate button.

Inside *LogIn Page* there is *LogIn* form provided for entering credentials. When a user submits them, in case there is no error in the input, *logIn()* function is executed and the user gets redirected to *MyCalendar Page*. In case of error a message is shown on the screen.

If the user has forgotten his/her password he/she can click on a change password button and get navigated to *ChangePassword Page*. The *LogIn Page* forwards the username to *ChangePassword Page*, where the user can type the newly desired password. In case there are no errors the system will find the email of a user by executing *FindEmailBasedOnUsername()*, and will afterwards execute *SendVerificationEmail()*. Changing of password is done by assuming that the email system is a trusted element and the user can change the password only if he has access to his/her email.

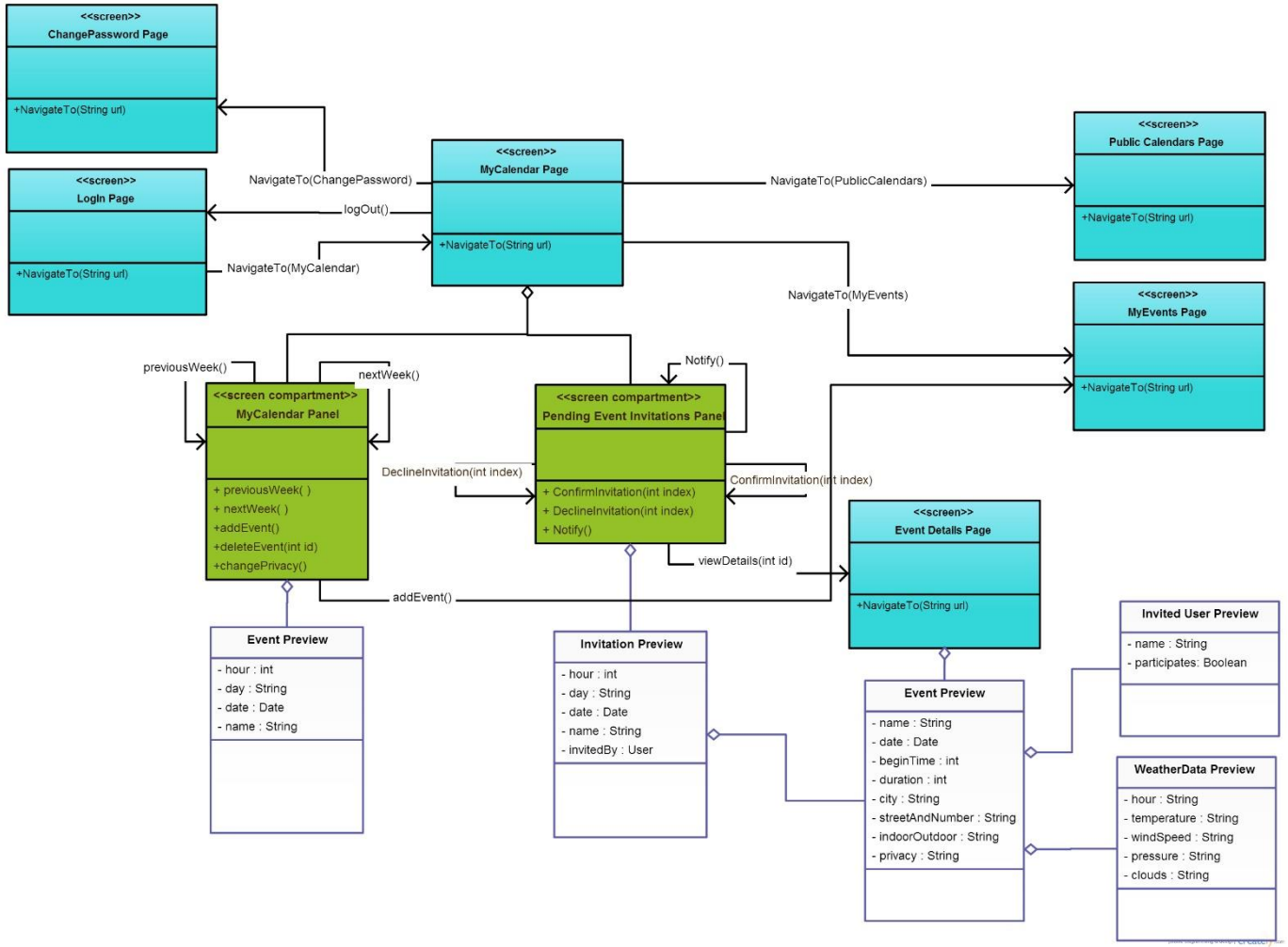
Note here that, for simplicity, all functionality of *MyCalendar Page* is omitted from the diagram. The mentioned page is described in next section.



Milan, 9th November, 2014

3.2. MY CALENDAR MANAGEMENT

This part of the UX Diagram allows to understand how the Event Invitations and User Events are managed and represented through the user interface. For simplicity, only the interaction that comes from *MyCalendar Page* is presented in the diagram.



Milan, 9th November, 2014

When the user has logged in he/she can change password, in the same manner like in previous diagram – by entering newly desired password and clicking on the link that he received in external email inbox.

The user may choose to log out, by clicking on appropriate button in User Interface (UI), that would lead him to *LogIn Page* that was described in previous section.

Now we will concentrate on functionalities of *MyCalendar Page*. It consists of two panels, *MyCalendar Panel* and *Pending Event Invitations Panel*.

MyCalendar Panel has a purpose of presenting user's events in a week-format table. In that table each day is divided into 24 hour cells. An event can occupy couple of cells meaning that the event lasts couple of hours. By clicking on a previous week button in UI, the system will execute the function previousWeek() and it will refresh the list of events. It similarly applies for next week button in UI and nextWeek() function.

Right clicking on some part of the table in UI selects appropriate field in table and brings context menu in which user can chose to add an event (if the field is empty) or delete an event (if there is an entry). Choosing to add an event invokes *addEvent()* function and navigates user to *MyEvents Page*. Choosing to delete an event only invokes *deleteEvent()* function, without any redirection.

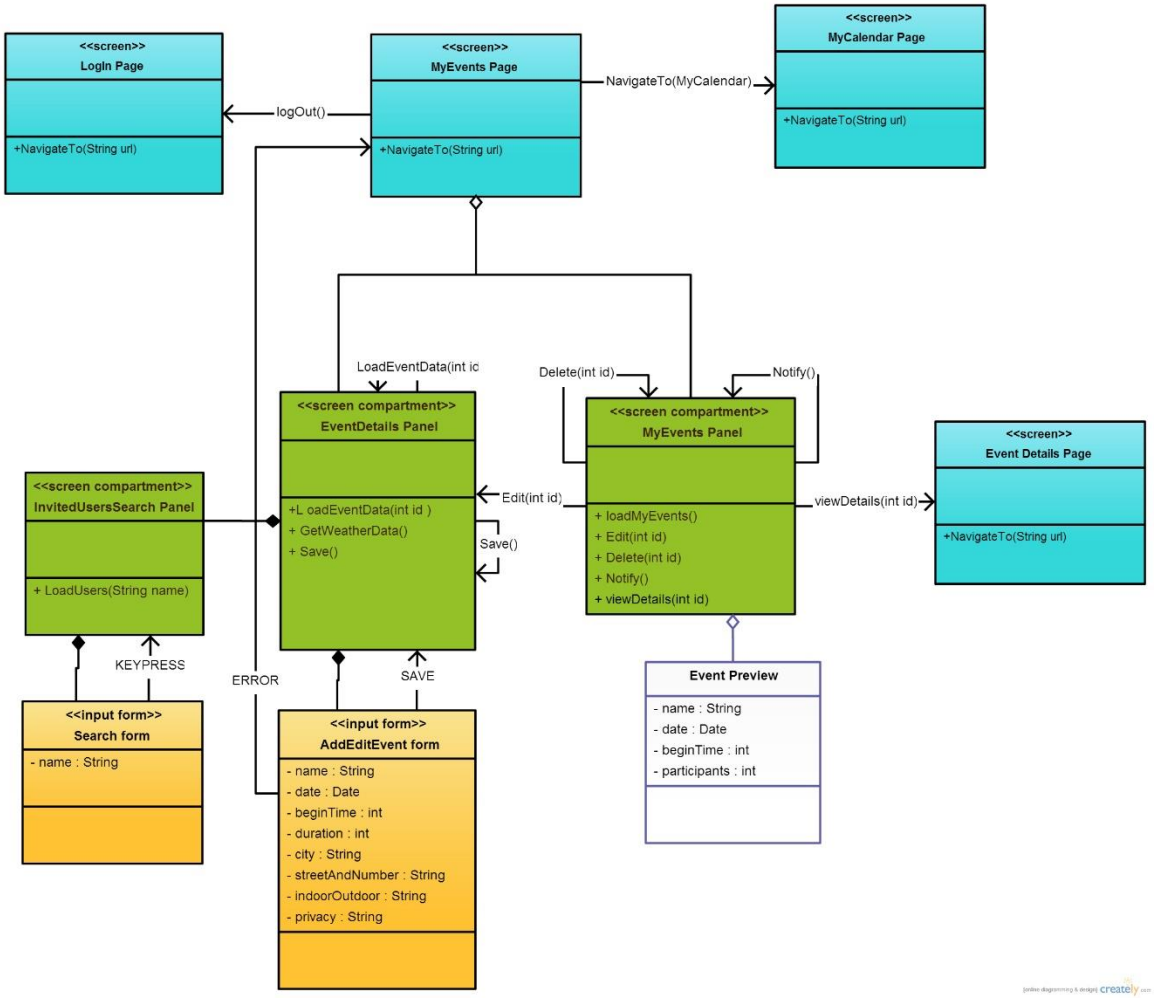
MyCalendar Panel also holds functionality for changing the privacy of user's calendar. Inside UI there is a combobox button where user can click and choose the privacy (private, public) of his/her calendar. Changing the value in UI invokes execution of function *changePrivacy().*

Pending Event Invitations Panel holds all invitations to events coming from other users (which are owners of those events). Invitation holds a hyperlink to event, placed on an event's name in UI. When a user clicks on an event's name, viewDetails(int id) will be executed and he/she is going to be redirected to EventDetails Page. User has two options, to confirm invitation or to decline it. ConfirmInvitation(int index) or DeclineInvitation(int index) is executed depending on which action user chose to perform. Notify() function tells to subsystem to reload invitations because new invitations are present.

User can also chose to navigate to *MyEvents Page* or *Public Calendars Page* by clicking on appropriate links in UI.

3.3. EVENTS MANAGEMENT

This part of the UX Diagram allows to understand how the event management functionalities are offered through the user interface.



Milan, 9th November, 2014

Through *MyEvents Page* a user can access the events that he/she created or the events to which he confirmed an invitation. More description on the possible actions and parts of the system follows.

MyEvents Panel holds informations about own events and participating events. It is presented as list in UI. Participating events are not modifiable, they can only be deleted from the user's list. That causes invocation of *Delete(int id)* and *Notify()* (for refreshing the list).

Own events can be modified and deleted. Last ones are deleted from entire system (not only from the list in UI) by invoking *Delete(int id)*. *Notify()* follows to refresh the list. Modification is done by invoking *Edit(int id)* and passing Event's id to *AddEditEvent form*, which is part of *EventDetails Panel*. After passing that id, LoadEventData(int id) is invoked.Now, editing of an event becomes similar to adding, so, for simplicity, in following paragraph we will describe adding/editing through invocation of *save()* function.

After filling the fields in AddEditEvent form and clicking on submit, the system invokes GetWeatherData() and save() functions.

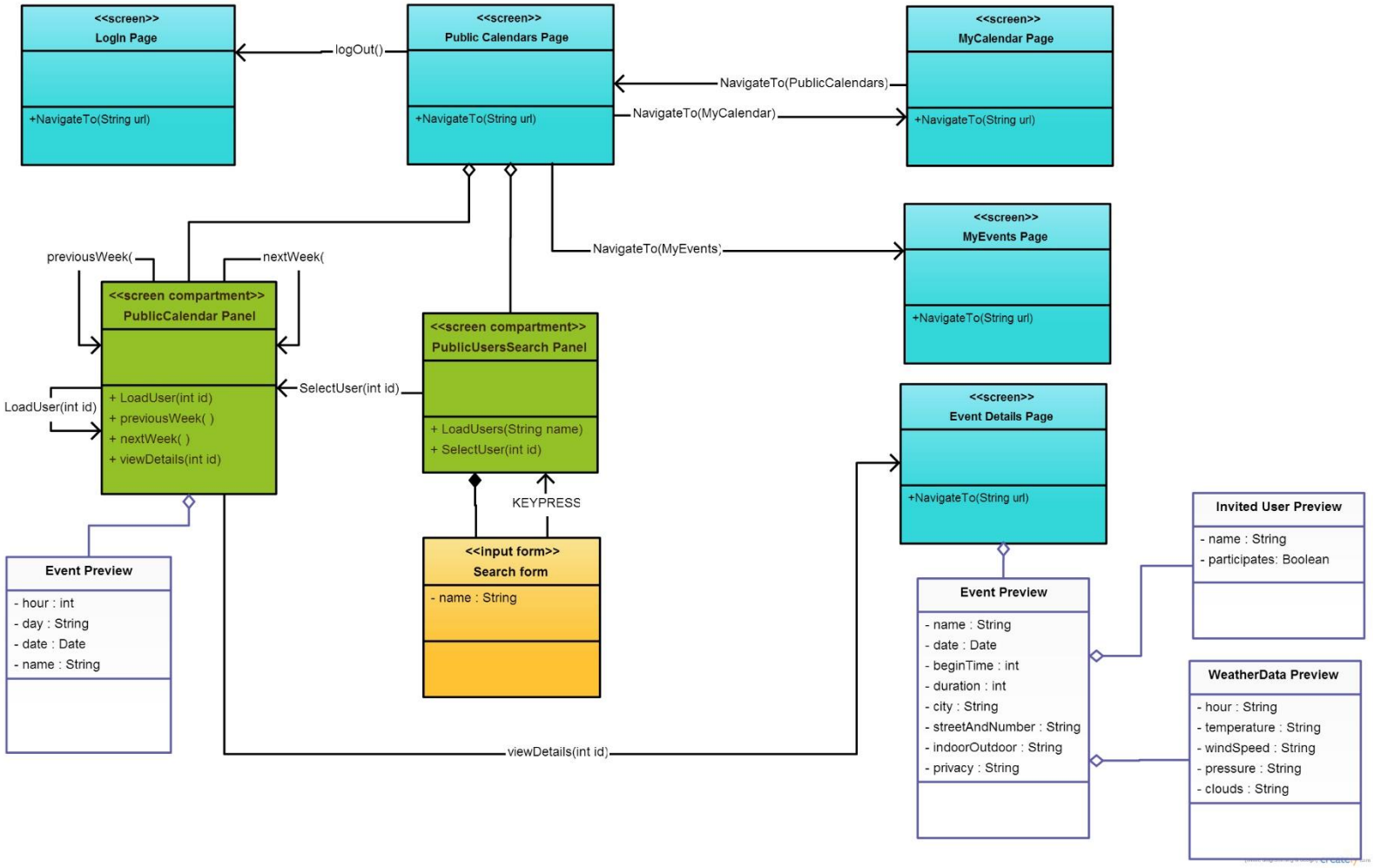
User can always delete or create invitations to his own event through *Search form* in *InvitedUsersSearch Panel*. Pressing any key in search form invokes *LoadUsers(String name)*.

Logging out and navigation shown in the diagram are similar as in previous section, thus repeating their descriptions would be useless.

Milan, 9th November, 2014

3.4. PUBLIC CALENDARS PREVIEW

This part of the UX Diagram allows to understand how the user searches and previews public calendars using the user interface's components.



Search functionalities can be performed through the Search form which is placed inside the *PublicUsersSearch Panel*. Like in previous diagram, keypress causes invocation of *LoadUsers(String name).*

When user selects some user in a list *SelectUser(int id)* inside of *PublicUsersSearch Panel* causes invocation of *LoadUser(int id)* in *PublicCalendar Panel* which loads the events of the selected user. *previousWeek()* and *nextWeek()* are the same as in previous sections. Only public events of public calendars are possible to view in detail. Private events of public calendars are shown as busy in the week table.

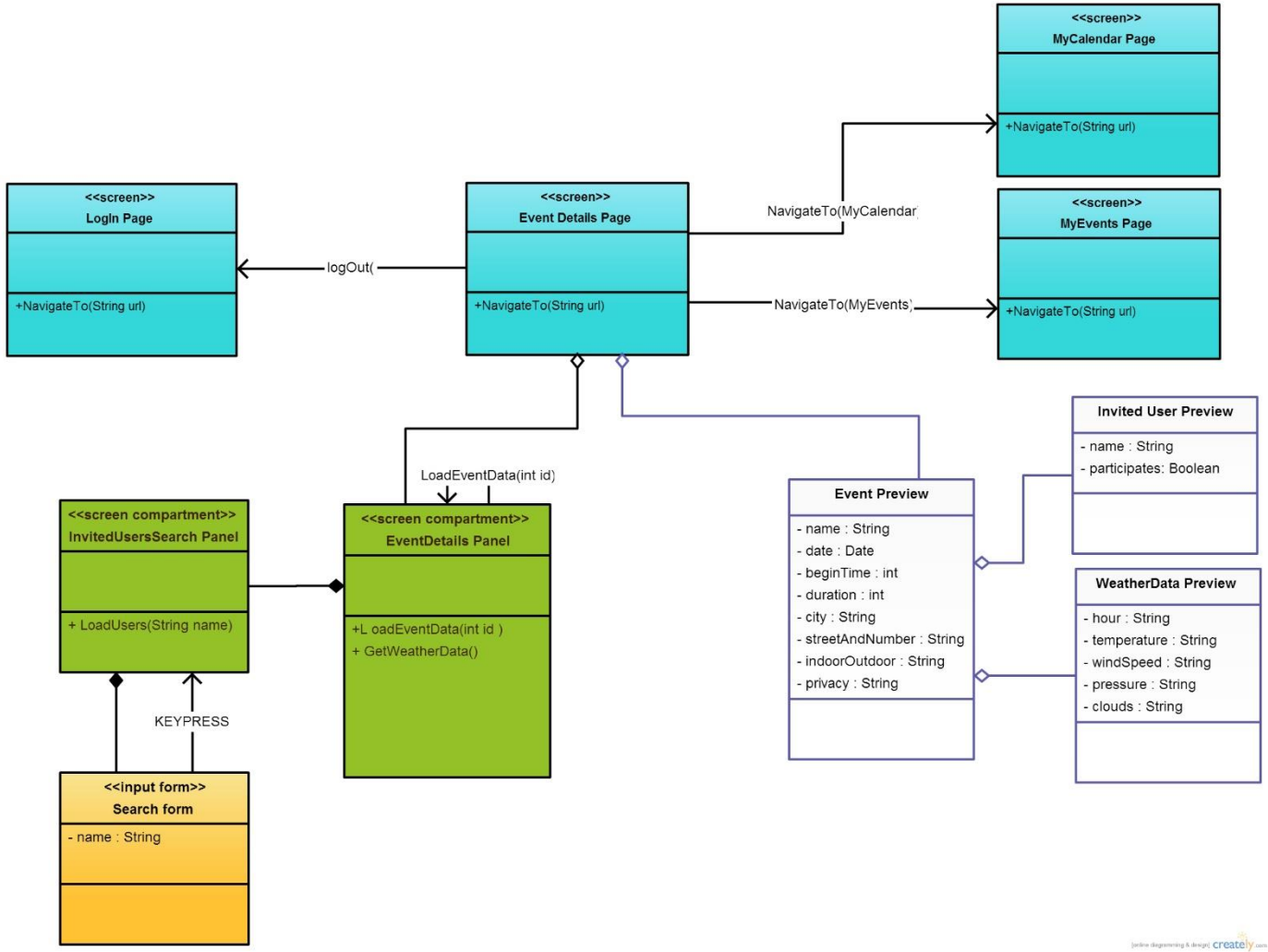
When user clicks on on an event *viewDetails(int id)* is invoked and he/she is navigated to *EventDetails Page*.

Logging out and navigation to *MyCalendar Page* and *MyEvents Page* are provided and are implemented similar as in previous sections.

Milan, 9th November, 2014

3.5. EVENT DETAILS

This part of the UX Diagram allows users to preview event details restricting them from any modification.



Milan, 9th November, 2014

EventDetails Panel (with embedded InvitedUsersSearchPanel) provides preview of the event's details. On loading of *EventDetails Page*: *LoadEventData(int id)*, *GetWeatherData()* and *LoadUsers("")* are invoked.

User may filter the search and find which users are invited through *Search form* inside of *InvitedUsersSearchPanel*.

logout() and navigation to *MyCalendar Page* and *MyEvents Page* are provided, as discussed in previous sections.

Milan, 9th November, 2014

4. BCE DIAGRAMS

We decided to give a further design schema of MeteoCal using the Boundary-Control-Entity pattern, because it is very close to the Model-View-Controller pattern (in fact we can say that boundaries maps to the view, controls map to the controller and entities map to the model) and UML defines a standard to represent it.

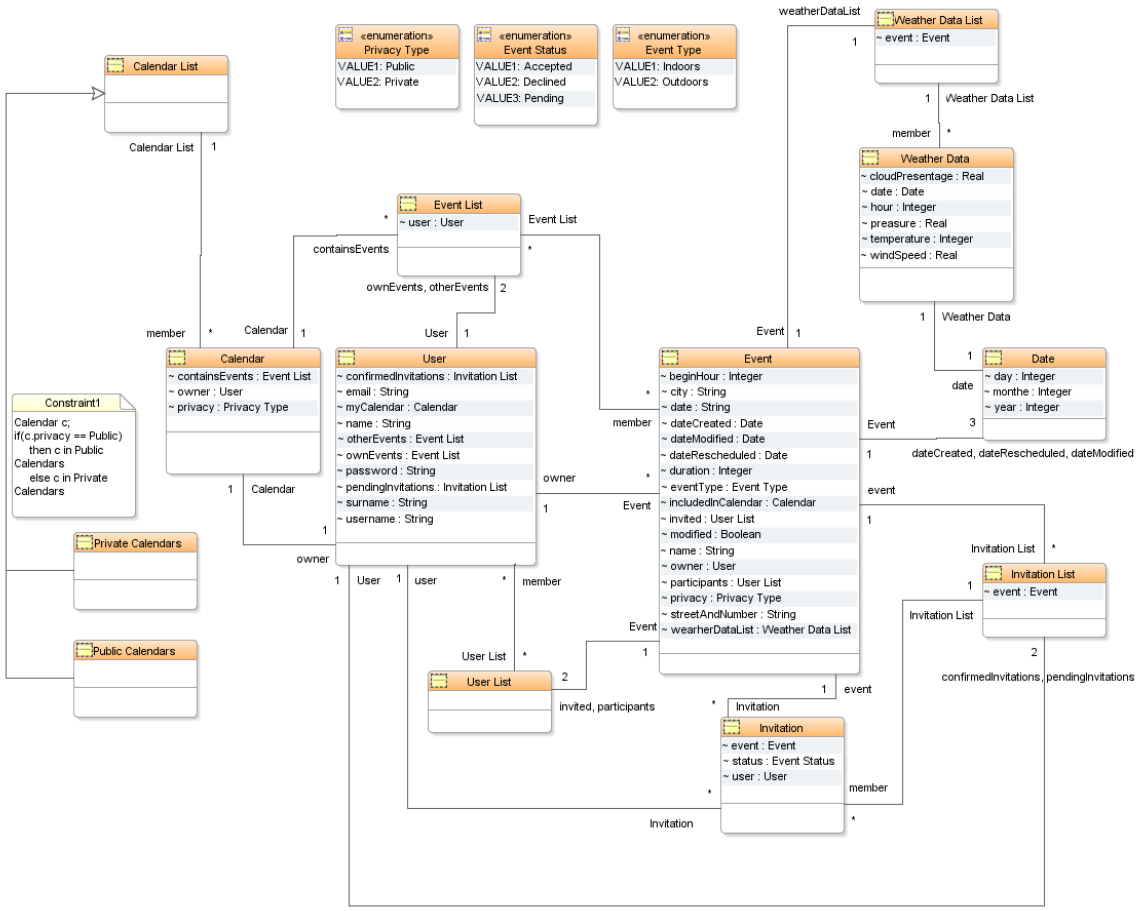
It is important to know that boundaries are partially derived from the Use Cases Diagram provided in the RASD (so they don't need further explanations) and they gather some screen in the UX Diagram. All the methods of the screens are written into the appropriate boundaries (maybe some names changed a bit only to make them more understandable in this context). There is a method show(Page) for every screen in the UX Diagram. Screen compartment doesn't have a "show method" because we considered them as pieces of screens and not as standalone screens.

Finally it is important to say that entities do not represent the ER Diagram, but only a conceptual view of the entities used in the BCE.

We decided to separate the BCE diagram into the sub-systems of MeteoCal. In this way everything will be clearer.

4.1. Entity Overview

All entities of the BCE model will be presented in a very fragmented way, to prevent the diagrams to be too chaotic. For this reason we provide an entity overview, in the way that the reader can always refer to this diagram.



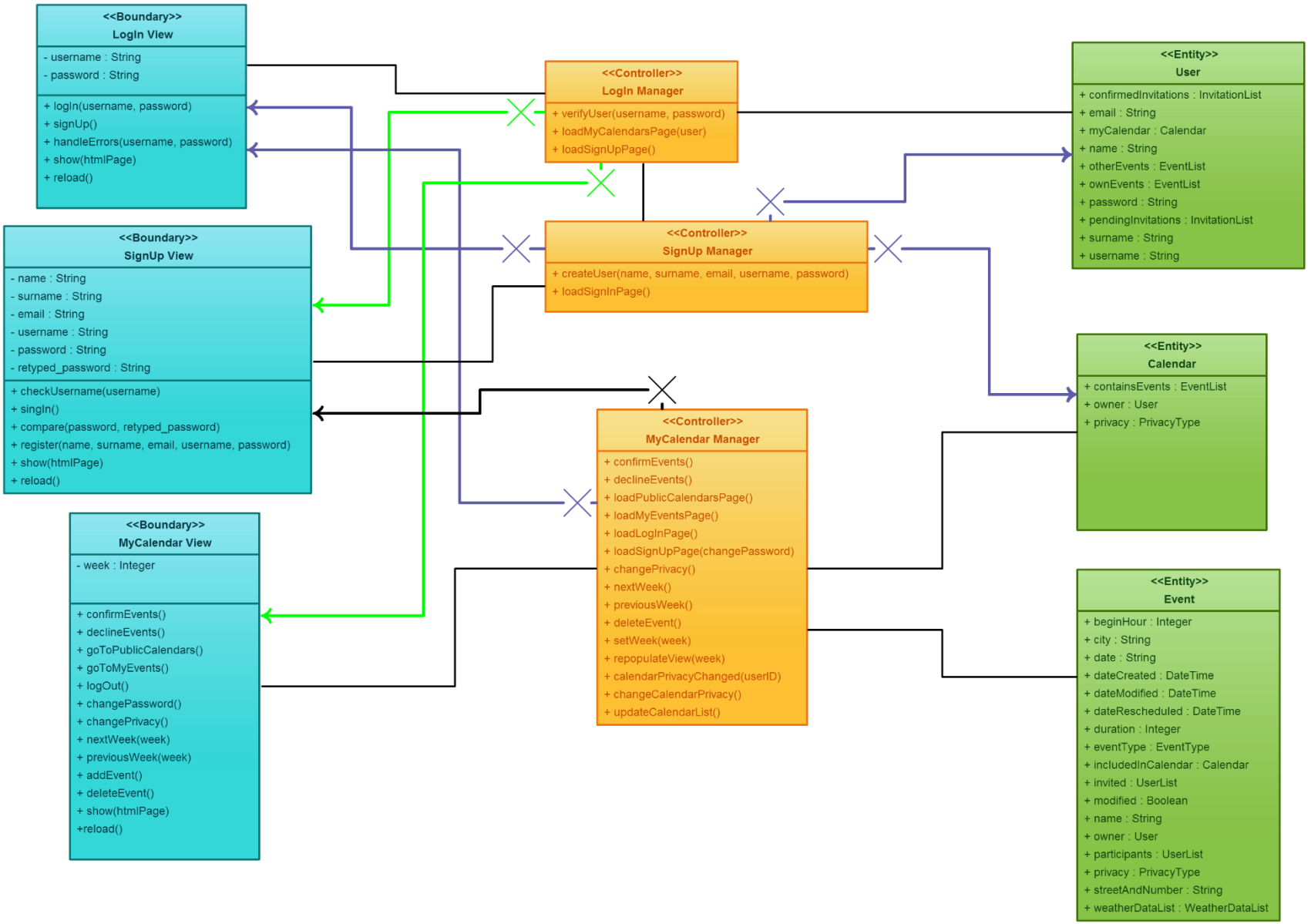
Milan, 9th November, 2014

4.2. Sign Up And Log In

The three boundaries LogIn View, SignUp View and MyCalendar View represent the interface with generic users and the basic functionalities that they can reach at a first glance of the system (in other words, they represent the functionalities that a generic user can exploit when his enters the system).

We write down now a synthetic description of the controllers used:

- LogIn Manager: this controller manages the verification of profile data in case of log in or for a new user which desires to register. It transfers control to SignUp Manager in case a user wants to register. It transfers control to MyCalendar Manager in the case of a successful log in operation.
- SignUp Manager: this controller manages the signing up process for the user along with functionalities to check desired username and the input of password. When the sign up operation is carried away it transfers back control to LogIn Manager since the newly registered user needs to log in first and then he can use the system.
- MyCalendar Manager: this controller manages the presentation of user's calendar data along with presenting invitations to events. It also allows navigation to other pages such as PublicCalendars, MyEvents, etc.



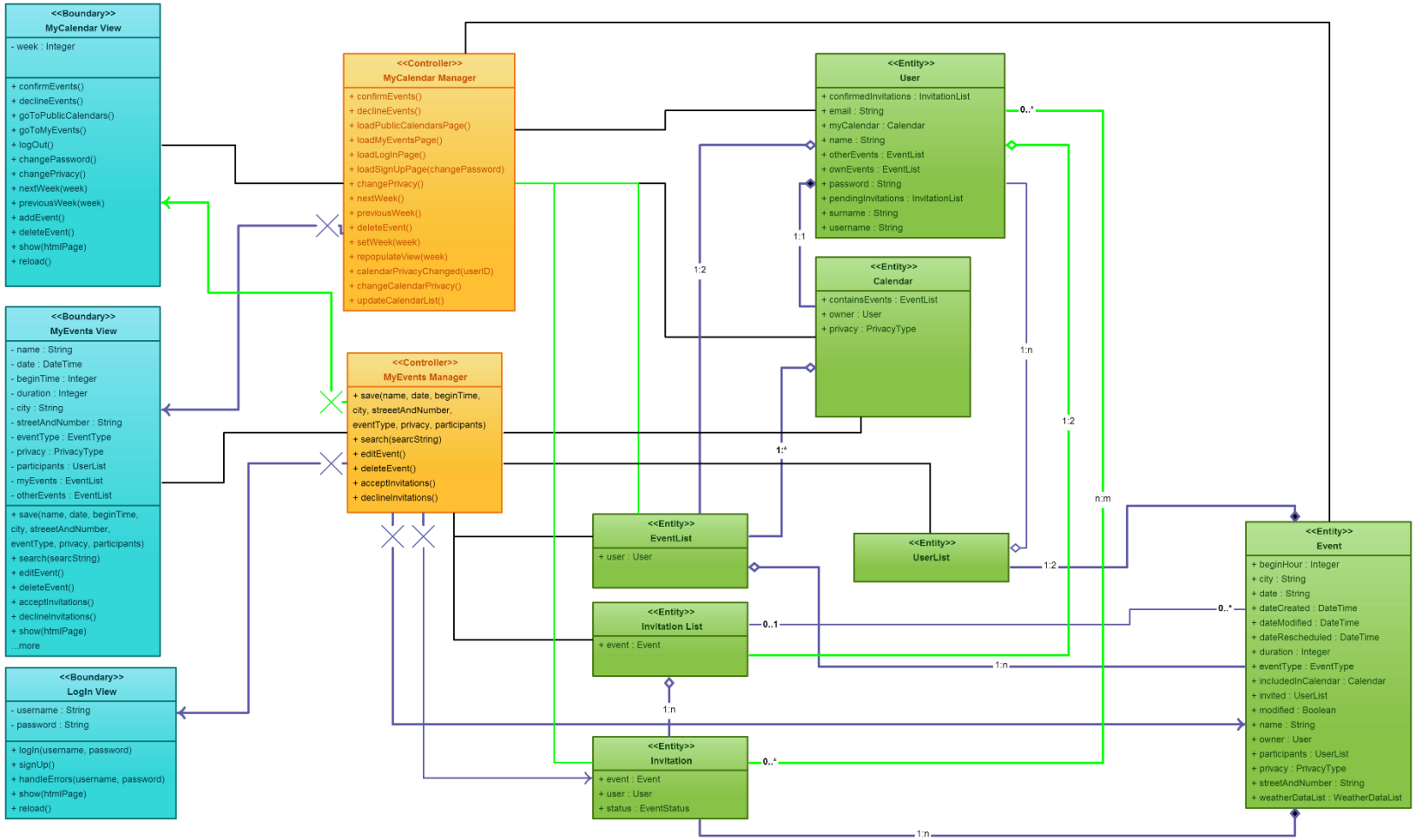
Milan, 9th November, 2014

4.3. MyCalendar BCE Diagram

In this diagram the MyCalendar View boundary and the LogIn View boundary are repeated for a better comprehension, along with their respective controllers.

In this diagram the MyEvents Manager controller appears. This controller is responsible for presenting functionalities for MyEvents page which can be reached from MyCalendar page, these functionalities are:

- Create a new Event
- Search for User you which to invite
- Edit existing Event
- Delete existing Event
- Accept Invitations to Events
- Decline Invitations to Events



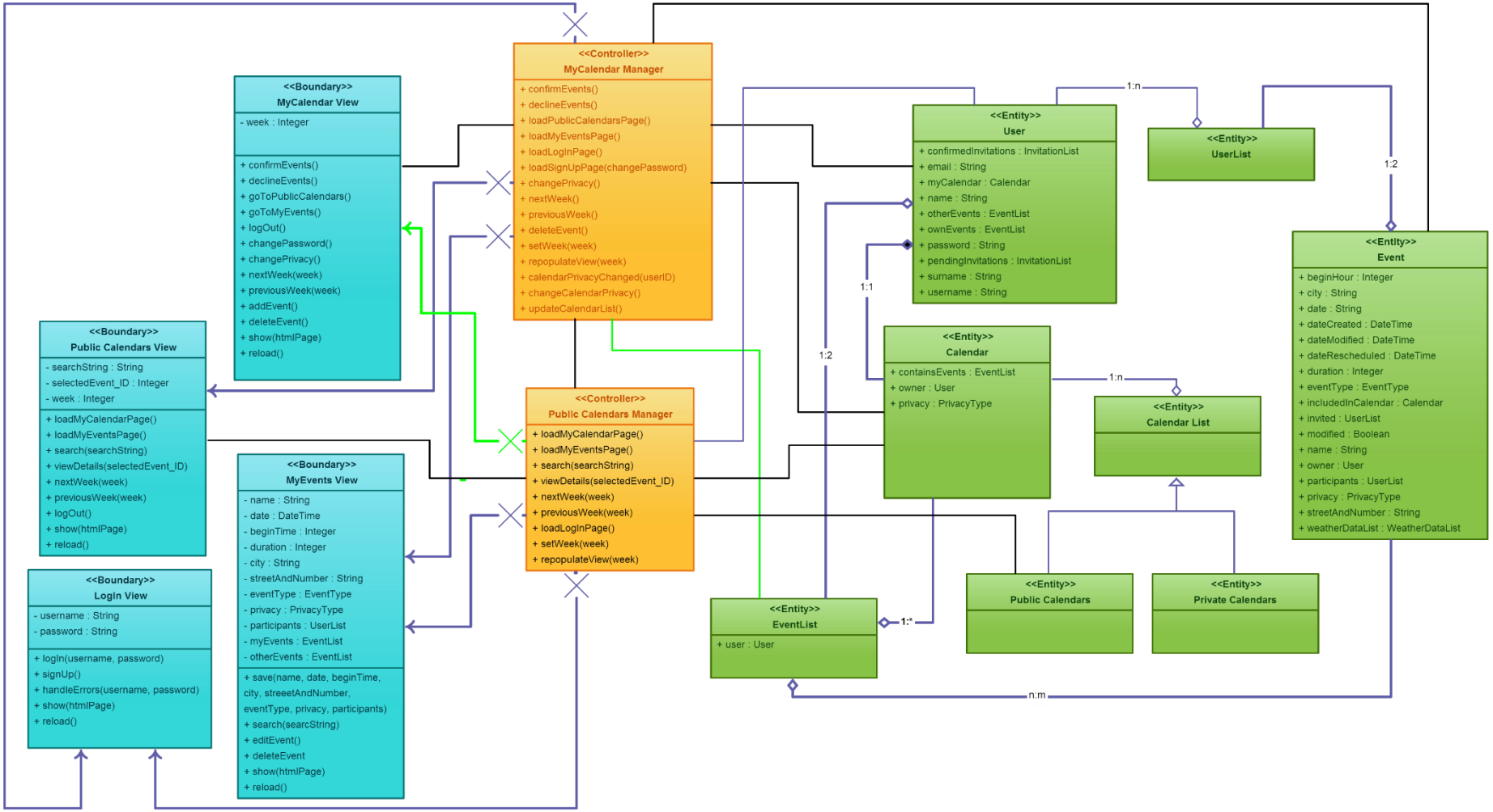
Milan, 9th November, 2014

4.4. PublicCalendars BCE Diagram

This section represent users functionalities available from PublicCalendars page and also includes MyCalendar Manager and MyCalendar View since user can navigate to PublicCalendars page from MyCalendar page.

The controllers are:

- **MyCalendar Manager**: this controller has already been described as a part of MyCalendar BCE diagram
- **PublicCalendars Manager**: this controller provides to the user functionalities offered at PublicCalendars page. This controller is responsible to load all public calendars for preview. It provides functionalities to navigate trough selected public calendar, also user can search for the other users that have made their calendars public. From this page a user can go to other pages, navigation to those pages is also a duty of this controller.

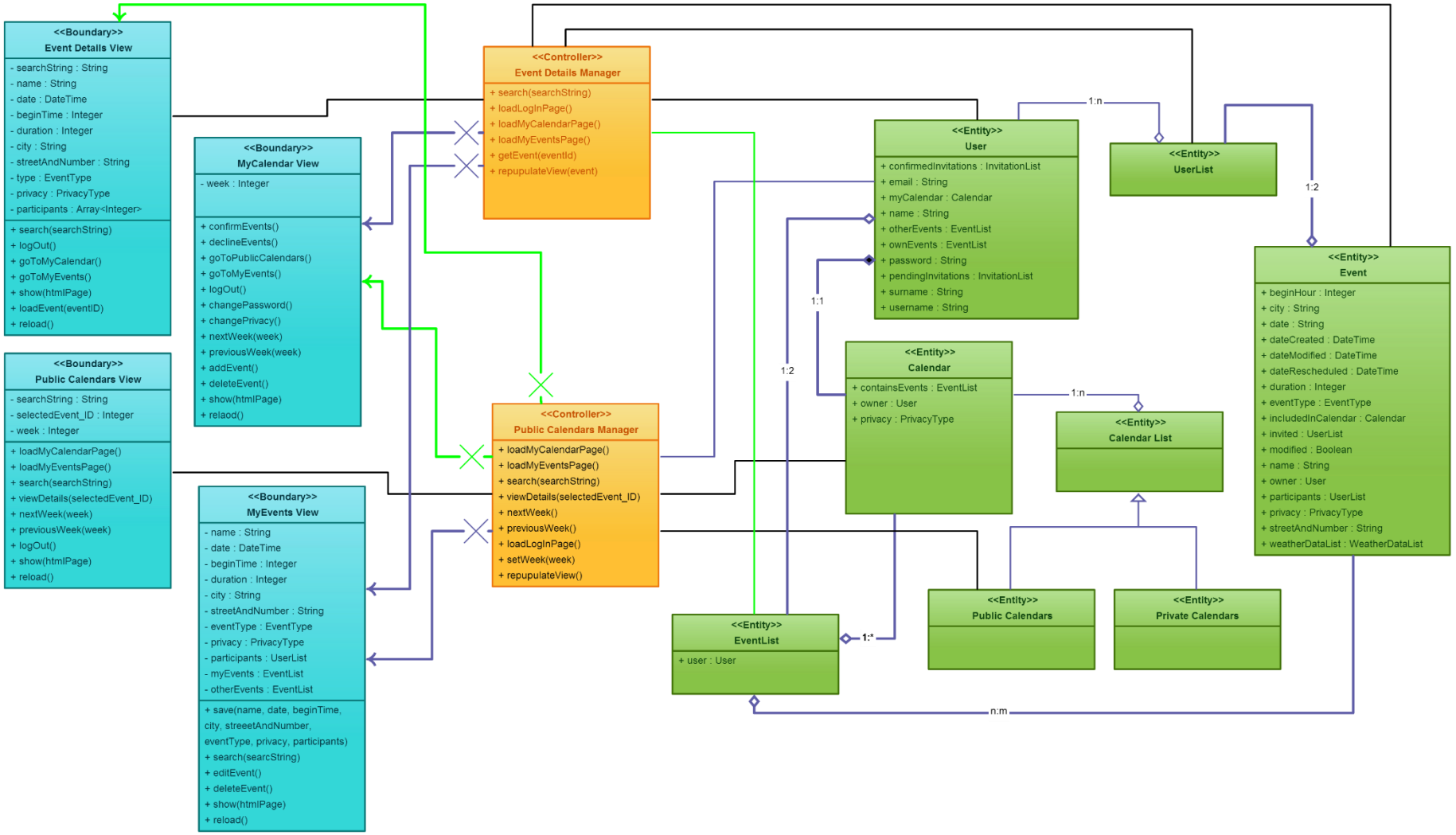


Milan, 9th November, 2014

4.5 EventDetails BCE Diagram

We write down the functionalities managed by the EventDetails Manager:

- Search for the user for which you want to inspect the event details (user is the event owner)
- Navigate to log in page
- Navigate to MyCalendars page
- Navigate to MyEvents page
- Fetch event by ID
- Repopulate the data in the view, based on the selected event



Milan, 9th November, 2014

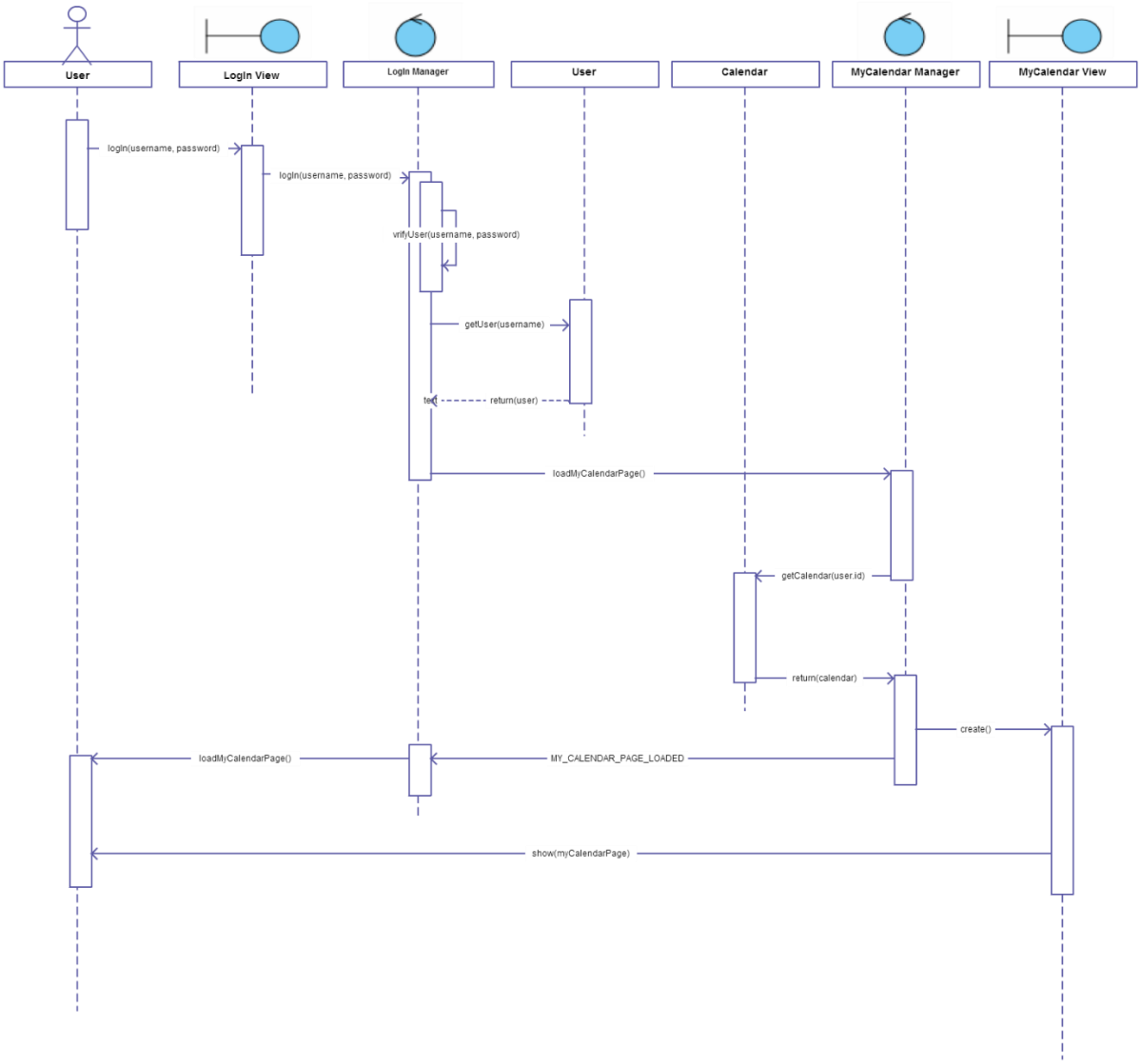
5. SEQUENCE DIAGRAMS

We provide some sequence diagram to let the reader better understand BCE diagrams described above. All the methods used are the methods listed into the BCE in boundaries, controls and entities.

5.1. Log In

A user:

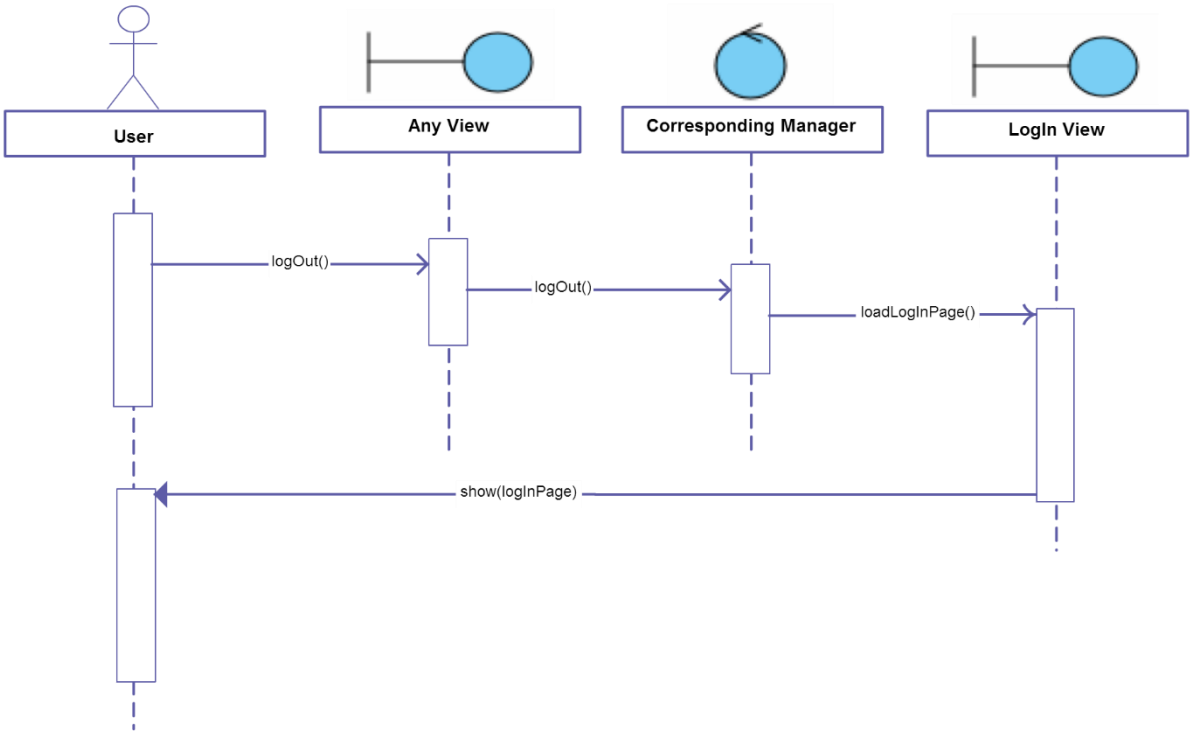
- Logs in.



5.2. User Logs Out

A user:

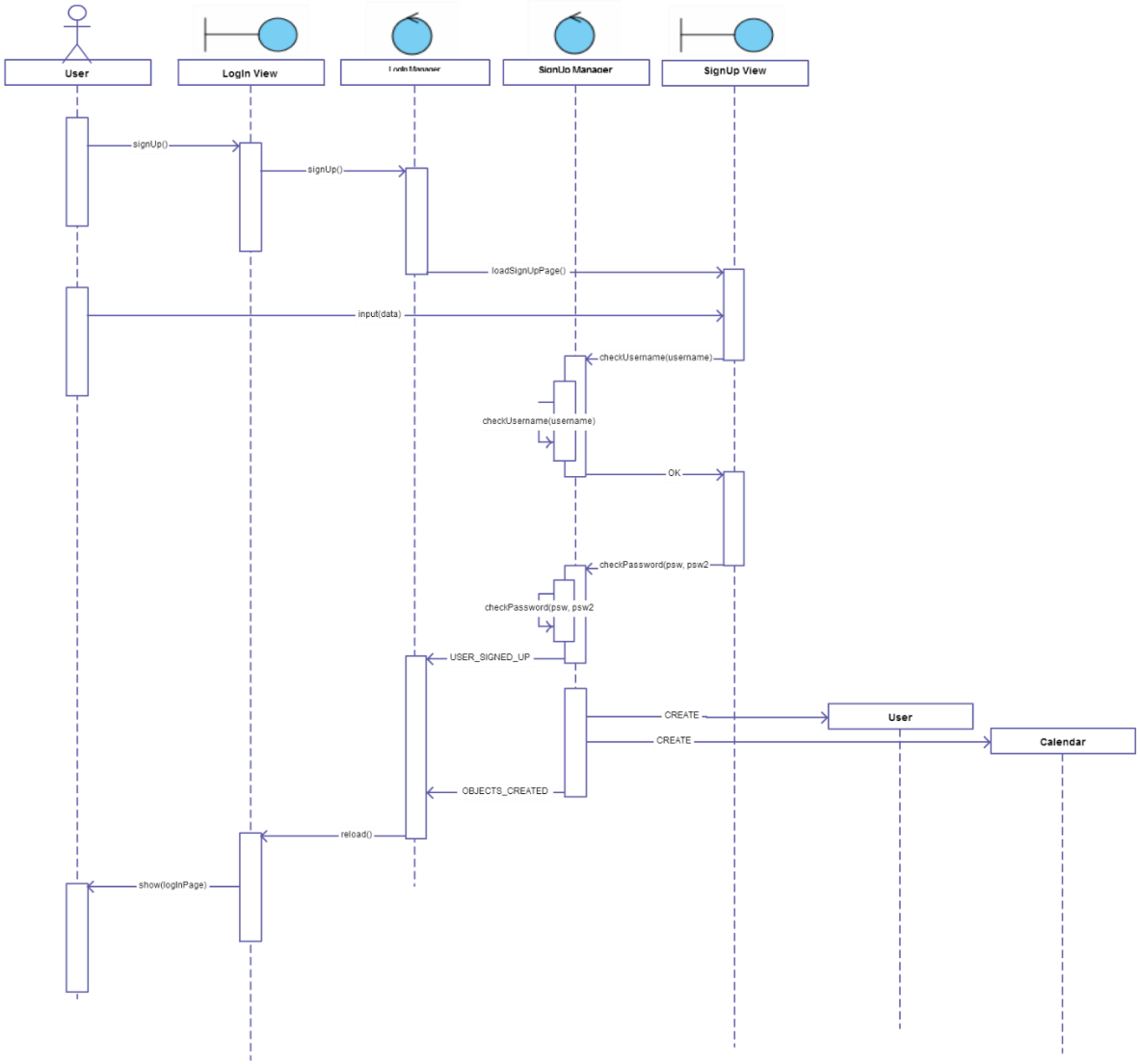
- Logs out of the system



5.3. Sign Up

A user:

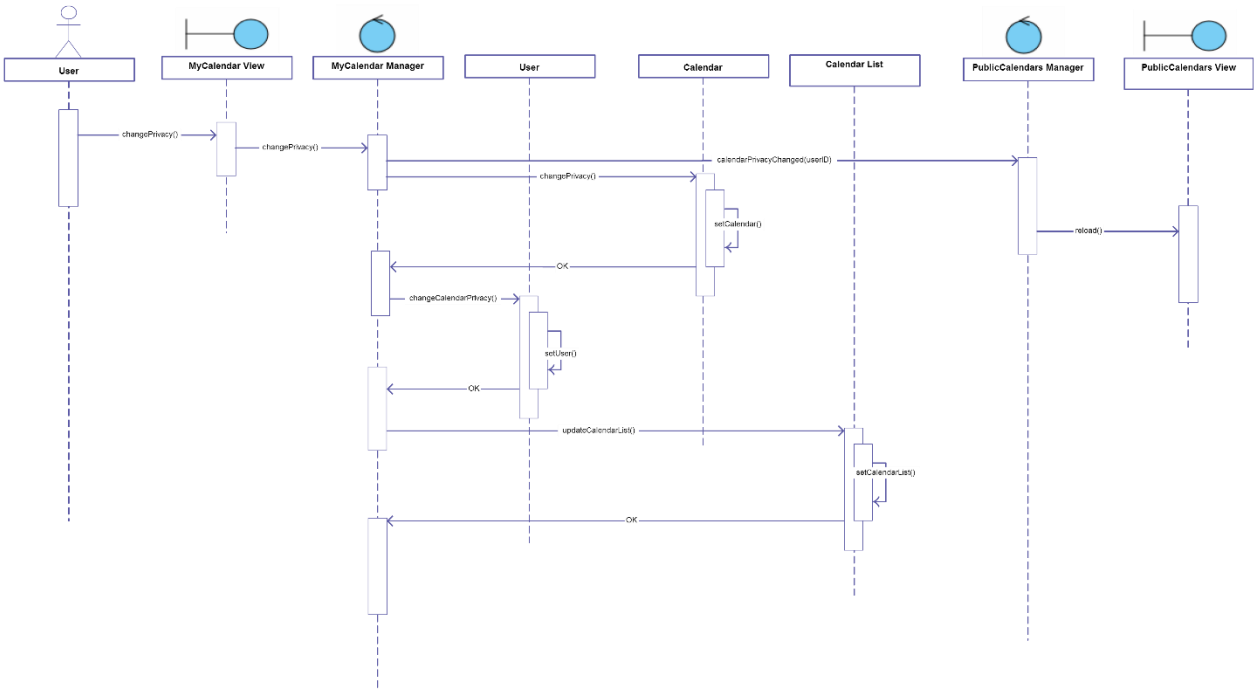
- Signs up as a new user of the system



5.4. Change calendars privacy

A user:

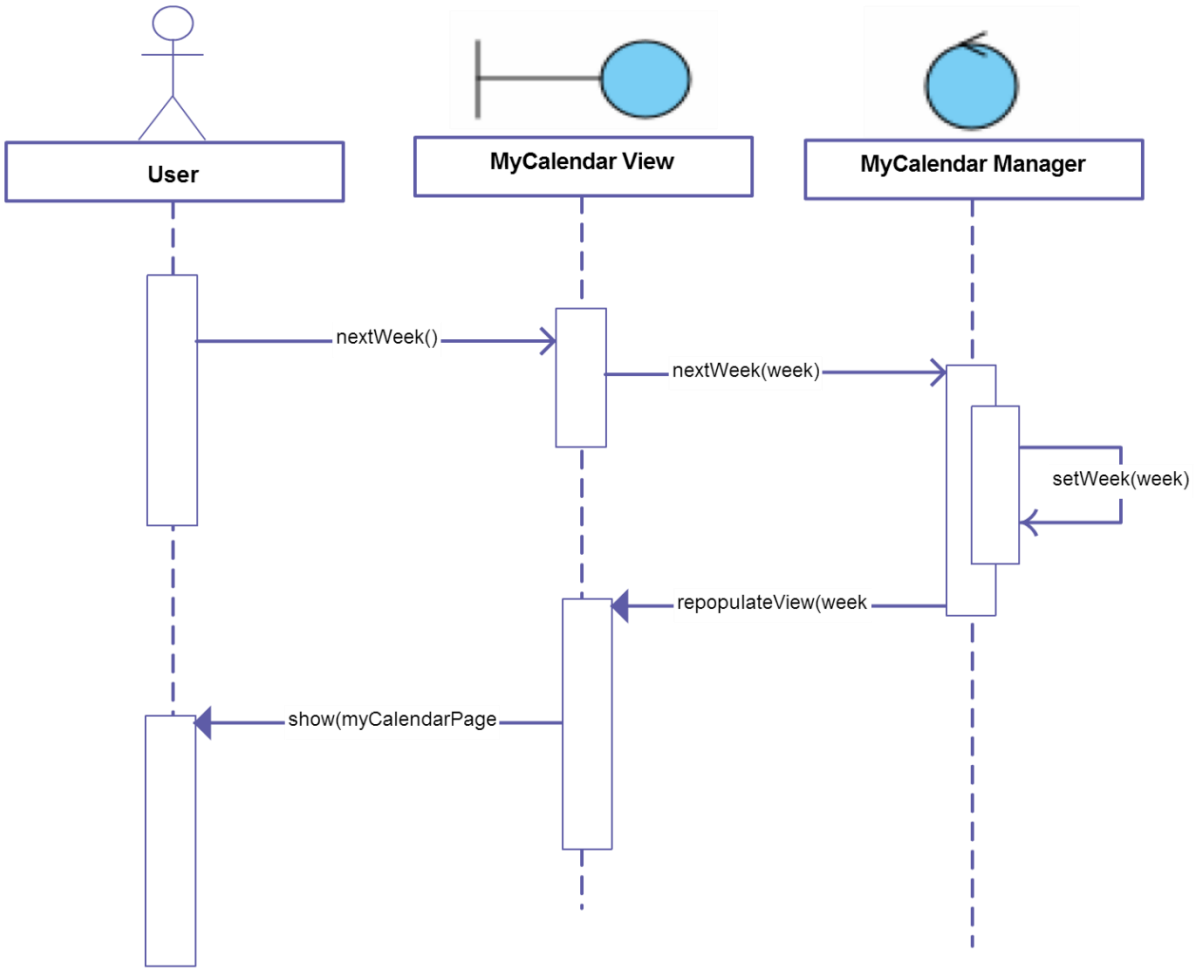
- Changes privacy setting of his calendar



5.5. Next week

A user:

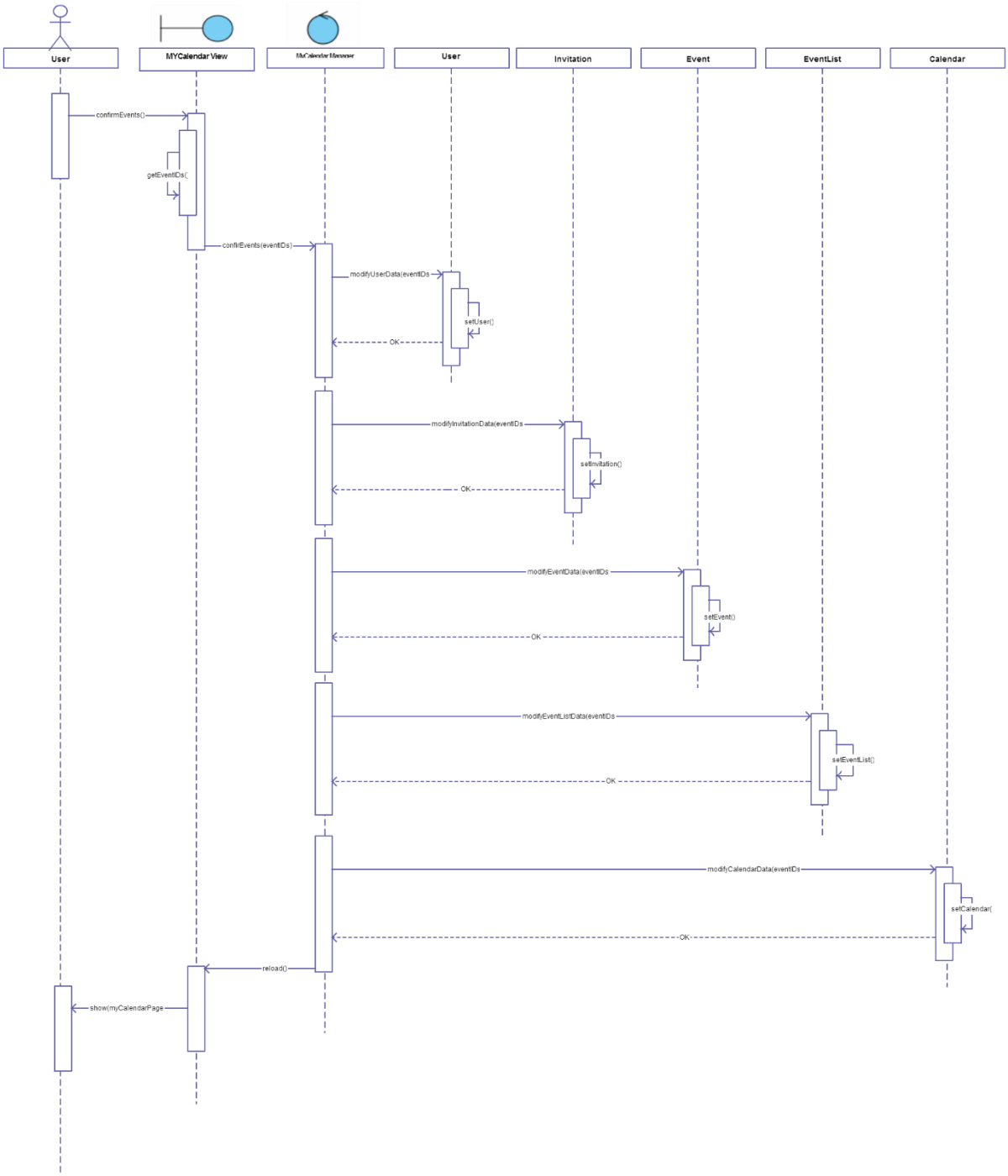
- User navigates through the calendar he is currently seeing



5.6. Accept Invitations

A user:

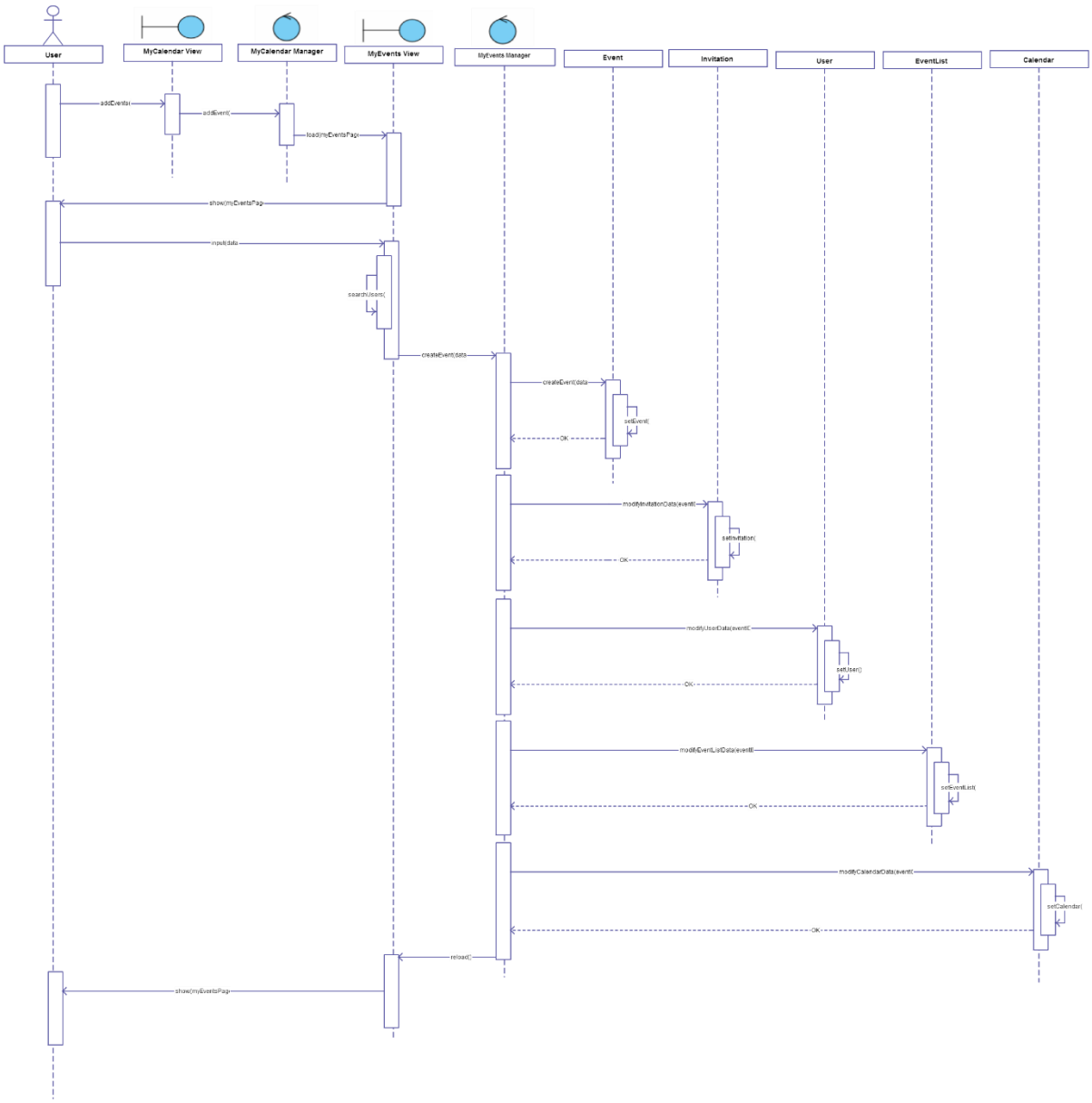
- User accepts invitations to the events he has selected



5.7. Add Event

A user:

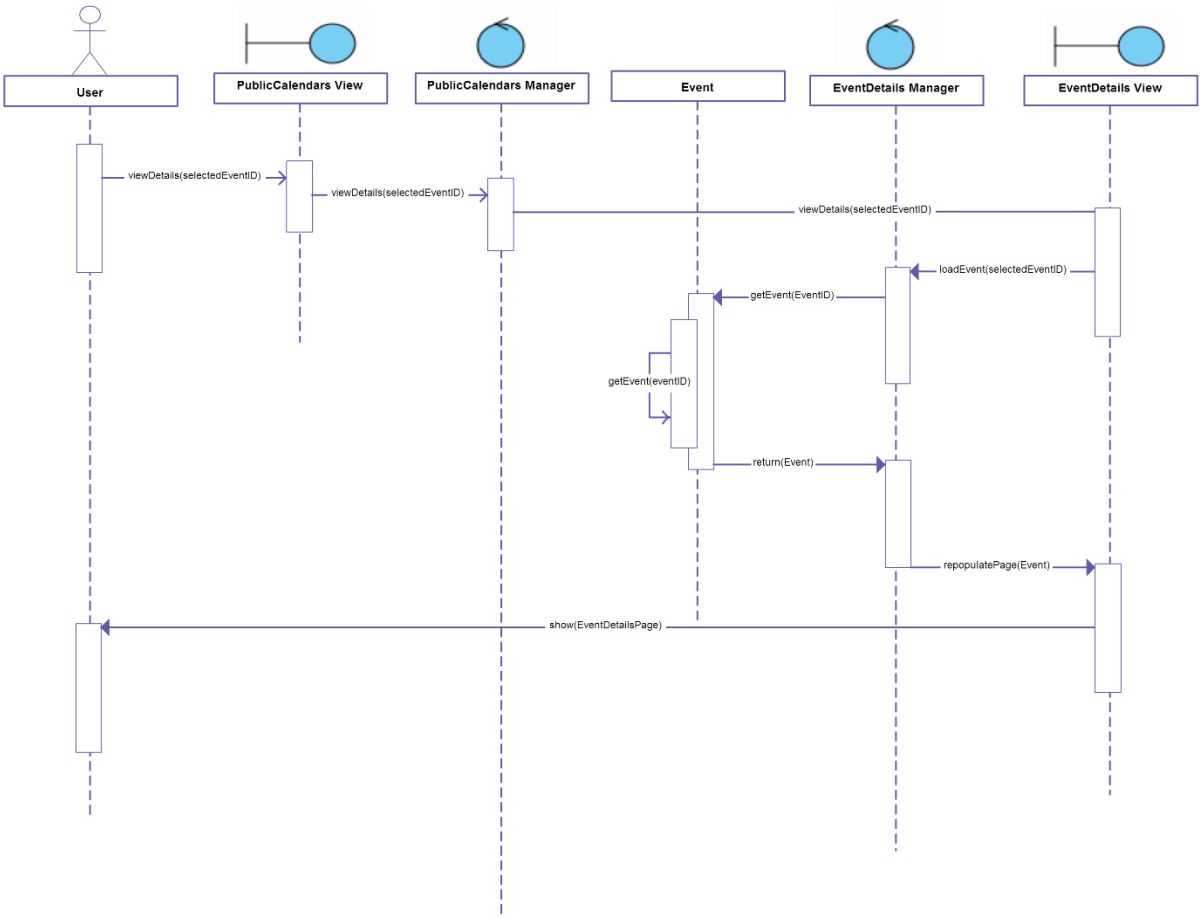
- User creates a new event



5.8. View Events Details

A user:

- User previews details of the event he has selected



6. FINAL CONSIDERATIONS

We decided not to draw any detailed diagram, because we think that a standard detailed diagram (with Server Page, Client Page, HTML Form and Control stereotypes) wouldn't have added meaning to our Design Document. In fact, with this diagram, we only had to have Server Pages if Client Pages (the same thing as Screens in the UX Diagram) are built dynamically, and we know that almost a large part of our pages will be dynamic.

Moreover, it is not clear if Server Pages and Controls represent directly Servlets or Beans.

For this reason we think that the standard detailed diagram couldn't bring us to a more specific knowledge of the implementation of our project.

Eventually, we drew UX Diagrams and BCE Diagrams instead, that are diagrams very detached from the architecture that lays under the project, but we decided not to draw more specific diagrams (such as Deployment View and Run-Time View), because we don't know so much JEE architecture to go into details. We know, in fact, that from now on we have to take a very big effort to understand the architecture well and to start implementing our project.

Name	**Working hours**
Stolic Nemanja	30 hours
Milos Colic	30 hours

Milan, 9th November, 2014