Politecnico di Milano

Scuola di Ingegneria Industriale e dell'Informazione

Computer Science and Engineering

Software Engineering 2 Project – A.Y. 2014/15

Design Document

Authors

Francesco Lattari (838380)

Alessandro Rimoldi (835506)

Summary

1. General description

This document has the purpose to provide a structural and technical description of the system that will be implemented; we can see the *Design Document* as a link between the RASD (that shows the problem in a detached way from the technical aspects) and the practical implementation of the project.

1.1 Project overview

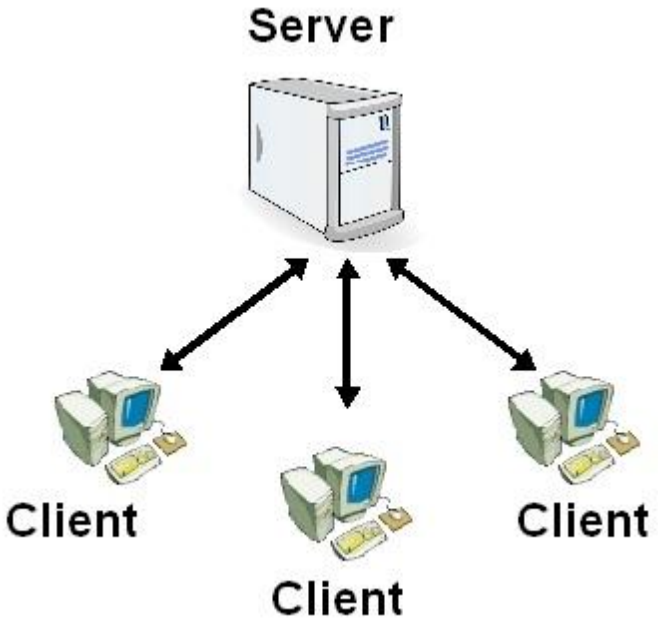
The project that we are going to design and implement is MeteoCal that is a new weather based online calendar whose purpose is to provide a system for creating and managing events. The registered user will be able to create, update, or delete an event with the ability to add more participants and specifying the place, date and time. The special feature of this calendar is to provide weather forecasts during the creation of events that can be of two types, indoor or outdoor, indicating possible bad weather conditions on the dates planned for the second type of events. Finally the organizer can define if the event should be public or private.

2. Technological choices

This section will provide an overview of the technological choices that will be adopted for the implementation of the system.

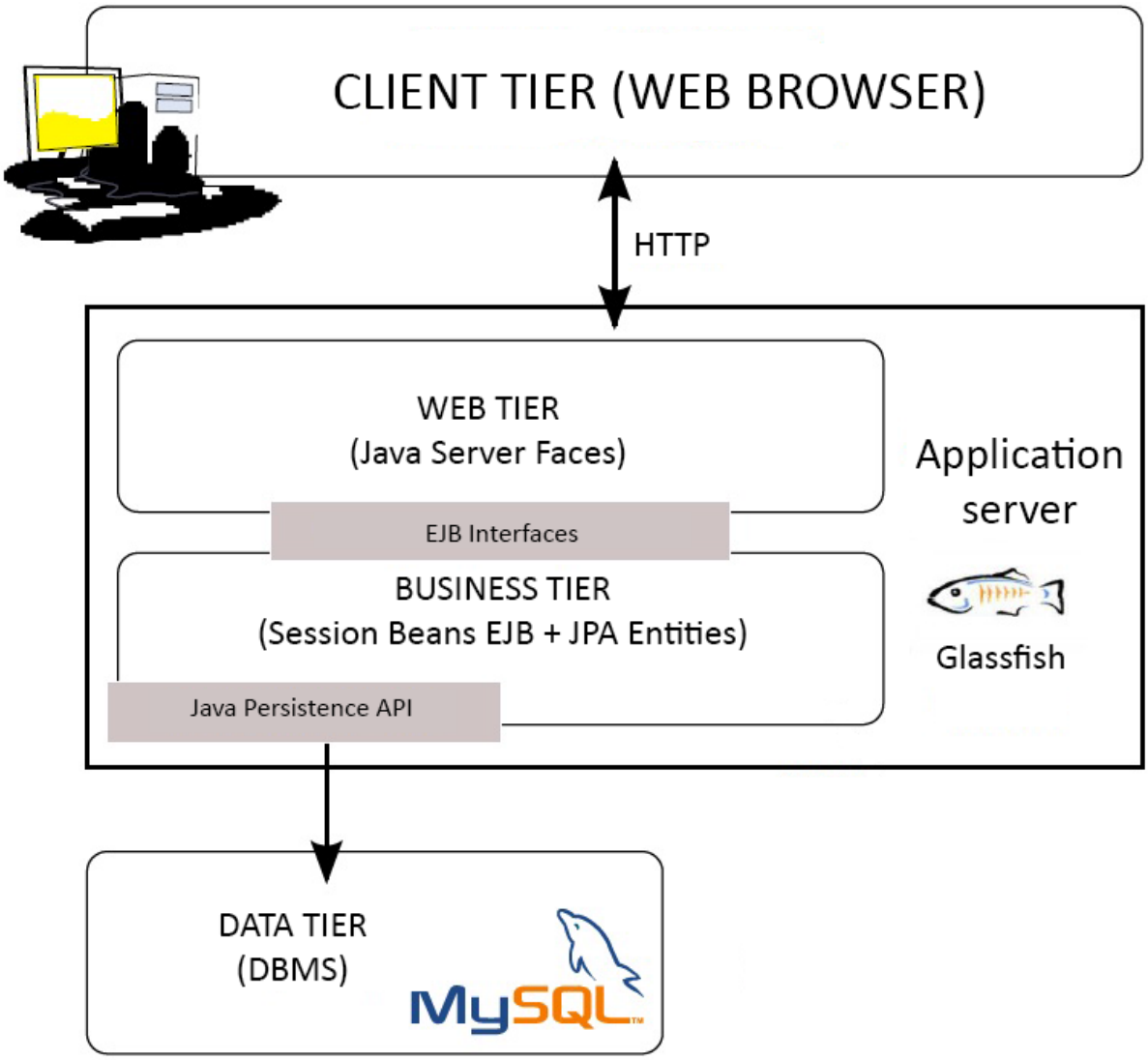
2.1 Overall schema

The system will be a four tiered web application, based on the *client-server paradigm* in which there are computers in a network (*servers*) that "provide services" to other computers (clients) that request and enjoy these services.



The four tiers of MeteoCal are:

1. **Client Tier:** : it's the level from which users access the application via a standard web browser;
2. **Web Tier:** this level has the task of mediating the relationship between the client tier and the business tier. It receives the requests from the client and forwards the collected data to the business tier waiting for a reply to be sent back;
3. **Business Tier:** it's the level of application logic where all the requests from the web tier will be managed using EJBs;
4. **Data Tier:** it's the database level. The Database Management System (DBMS) used is MySQL.



Returning to the *client-server paradigm*, the *client* will be constituted by the client tier, while the web tier, the business tier and the data tier are elements of the *server*.

Regarding the development environment, we will use the *NetBeans IDE v8.0.1* and to synchronize the work among the two members of the group, we have activated a Google Code.

2.2 Architecture description

The architectural style adopted, multi-tier, is typical of enterprise applications, in which the various logic levels are distributed on different physical machines communicating with each other via the network. In an architecture of this type, furthermore, each layer communicates only with the level immediately before and with the level immediately following, making the application as much as possible

modular.

Client Tier

It is the level from which users access the application via a standard web browser. This tier communicates via HTTP with the web tier, sending to the server the data entered by the user and viewing the HTML pages received.

In addition, the browser interprets the eventual Javascript code, that implements some functionality to improve the performance of the service and the user experience.

Since the main function of the client tier is collect user input and show the results of his operations, we can consider it as a *thin client*.

Web Tier

This layer receives HTTP requests from the client tier and responds by sending HTML pages. The pages are dynamically generated based on the data received by the user and the interaction with the business tier. The web tier will be implemented with technology JEE, and will work within an application server compatible.

Business Tier

This layer encapsulates the application logic and communicates directly with the database. The data of the application, in other words the *model* of the MVC pattern, are represented by objects like Entity Beans. The application logic and interaction with the database (the *controller* of the application) are made using EJB components.

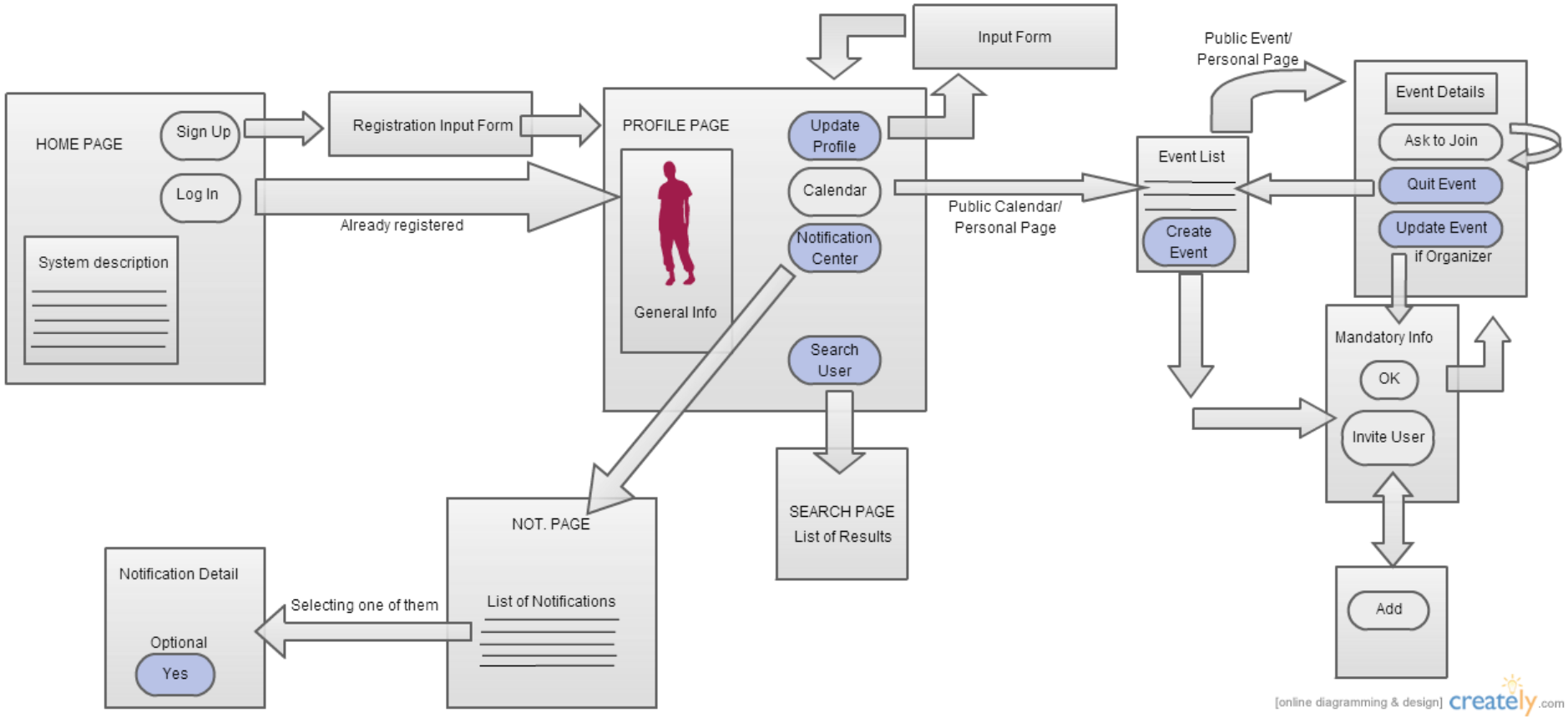
Data Tier

It consists of a DBMS, which achieves data persistence. The application server communicates with the database using the standard Java technologies (JDBC). Since communication between DBMS and application server is via network, the database can reside either on the same physical machine server or on another.

2.3 Project subsystems

We imagined what could be the final result of our project and from this analysis we obtained the following graph, very informal, but nevertheless we believe it will be very useful to understand how to navigate the MeteoCal application and identify key parts of the service.

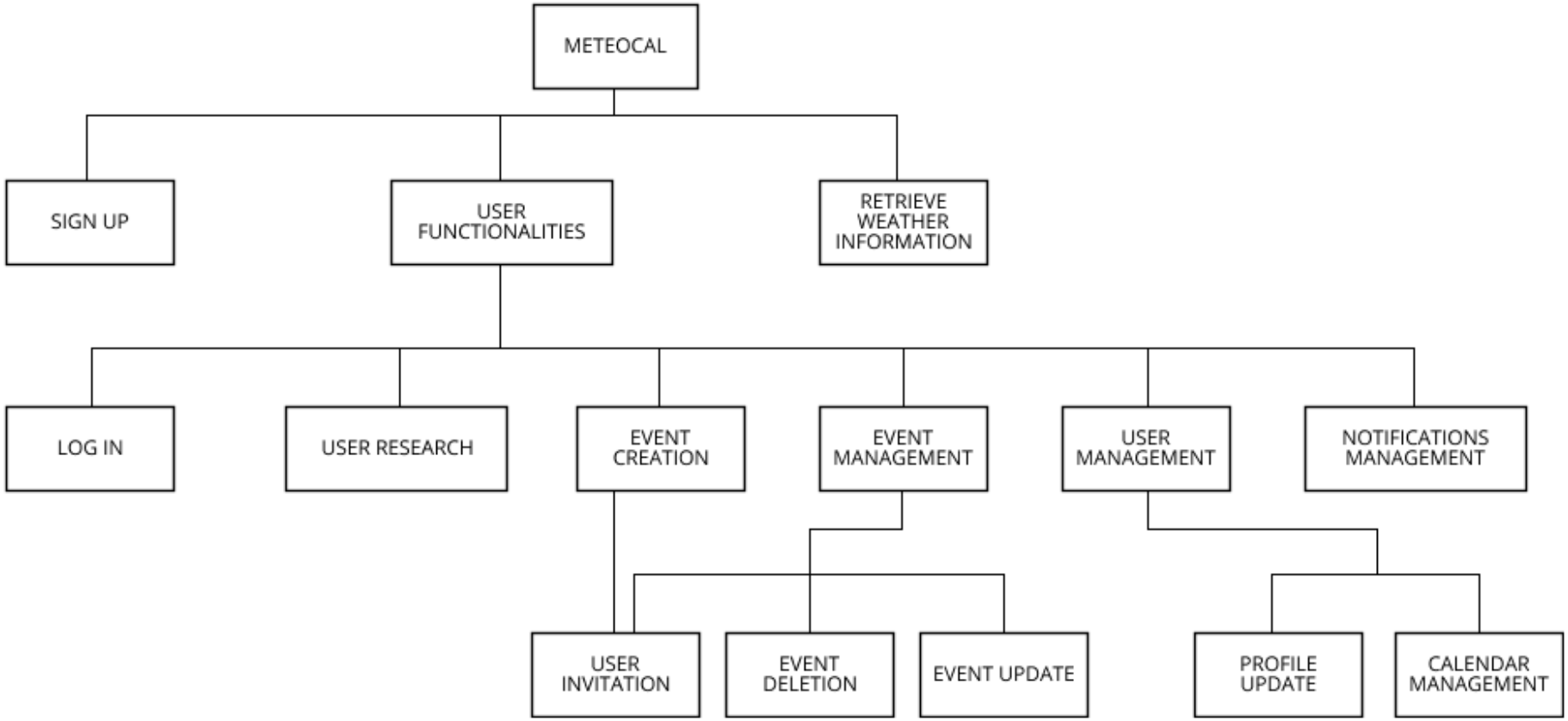
Through this approach, it was also possible to identify the possible subsystems of our system, “macro functions” or “interaction sequences” that should be able to be implemented independently from each other.



Although the graph is self explanatory, it should be noted that the *blue buttons* are visible only by the user on his profile or according to special conditions specified. Rather, the system should always show the “same screens” and show/hide in an appropriated manner additional “buttons”. For example, it is clear that the “Quit Even” function is available if and only if the user is logged to the event in question, or the “Update Profile” button is only visible lying on own profile and not on other people’s profiles, while “Calendar” is always available, but allows you to have information on the calendar if this is public or is your own personal calendar.

The subsystems that we have identified are these:

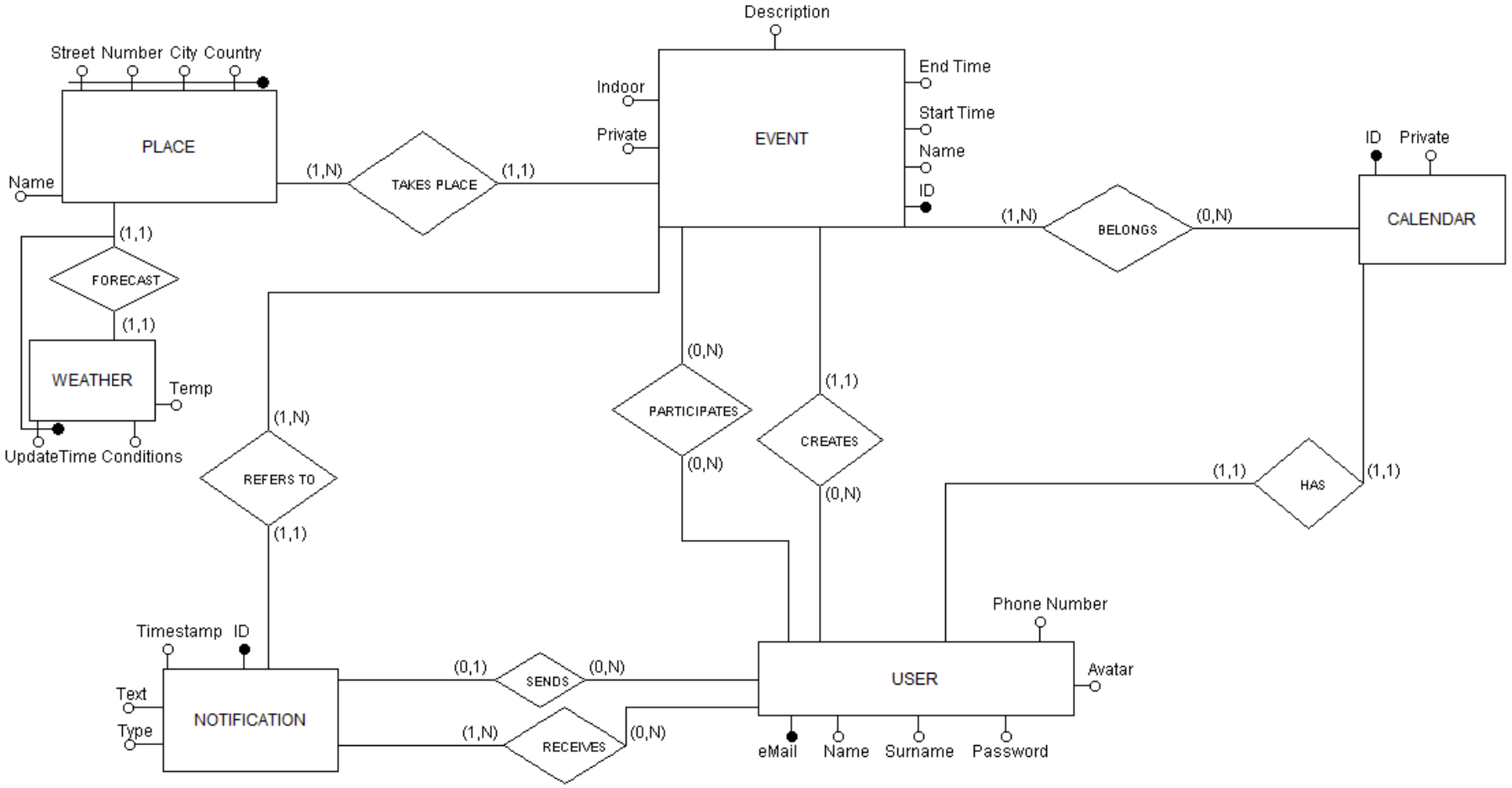
- *Sign up* subsystem;
- *Weather information* subsystem;
- *User functionalities* subsystem:
  - *Log in* subsystem;
  - *User research* subsystem;
  - *Event creation* subsystem, expanded by *user invitation* subsystem;
  - *Event management* subsystem, composed by: *user invitation*, *event deletion* and *event update* subsystems;
  - *User management* subsystem, composed by: *profile update* and *calendar management* subsystems;
  - *Notifications management* subsystem;


METEOCAL
SIGN UP
USER FUNCTIONALITIES
RETRIEVE WEATHER INFORMATION
LOG IN
USER RESEARCH
EVENT CREATION
EVENT MANAGEMENT
USER MANAGEMENT
NOTIFICATIONS MANAGEMENT
USER INVITATION
EVENT DELETION
EVENT UPDATE
PROFILE UPDATE
CALENDAR MANAGEMENT

3. Persistent Data Management

In this section we will present the structure of our database, starting from the ER diagram. We will describe all its aspects and we will derive from it the logic scheme which will correspond to the final structure of the relational database.

3.1 Entity-Relationship diagram



Entities and associations that have been identified are:

- **USER:** this entity represents the user registered to the system. He is identified by an univocal *e-mail*. We have decided to not use an ID because the system will not allow that the same email is used more over than one time in the sign up phase. The *user* entity has also a *password*, attributes related to personal data such as *name* and *surname* (maybe we will add also other info like the *birth date*) and a contact information (*phone*, additional and optional). Furthermore the *avatar* attribute will be optional. The entity is associated to the *notification* entity through two relationships, *sends* and *receives*, whose meaning is quite intuitive. Finally there are relationships related to the *event* entity and to the *calendar* entity.
- **CALENDAR:** this entity has been conceived with the idea to simplify access to all events belonging to the user's calendar, without having to verify that these events are organized by him or he is simply a participant of the event. Also *calendar* contains the boolean attribute *private* that identifies the state of privacy set by the user on his own calendar. *Calendar* is univocally identified by an *ID* and is related to *user* entity and *event* entity.
- **EVENT:** this entity is uniquely identified the attribute *ID*. Each event belongs to one or more calendar, so that there a many to many relationship with the *calendar* entity. The event could have only one creator, but many (or zero) participants. The number of participant is limited to an integer N defined, event by event, by the creator of the event. An event takes place in only one location and are linked to it some notifications of different kind. The *event* entity is also characterized by these attributes: *name*, *start time*, *end time*, *description*, *private* (boolean) and *indoor* (boolean). In the original idea the last boolean attribute was not present and in its place there were two sub-entities of *event*. In this way we avoided to associate "unnecessary" weather forecasts to an event that was declared immediately as indoor, but there was too much duplication of the other data.
- **PLACE:** the attributes of this entity are: *name*, *street*, *number*, *city* and *country*. "Unfortunately", except the first attribute, all the others are key for the entity. An easier solution, in term of number of keys, was to use longitude and latitude but it will not be so convenient to use. *Place* is obviously linked to *event* and *weather*.
- **WEATHER:** this entity is identified using the time of the last weather forecast update in union with the key attributes of the related place. At the moment

we put only two attributes, *temperature* and *weather conditions*, but, if necessary, we can complete the parameters with other information, typical of a weather service, such as the wind speed or the humidity or the quantity of rain expected. Originally we thought to not create a *weather* entity but just to use an attribute in *event* or *place* containing, in the form of string, all the information related to the weather forecast and update every time that value.

- **NOTIFICATION:** this entity is characterized by an univocal *ID* and other attributes such as the *timestamp* (in other word when the notification was generated), the *text* (the content of the notification) and the *type*. The last attribute is the result of an upward collapse of the sub-entities of *notification*: event invitation, participation request, participation confirmed, bad weather alert, event update or deleted and quit from the event. A notification is always destined to at least one user but the sender may not exist in case of the notification is sent by the "system", for example the weather forecast update for the organizer.

3.2 Logical scheme

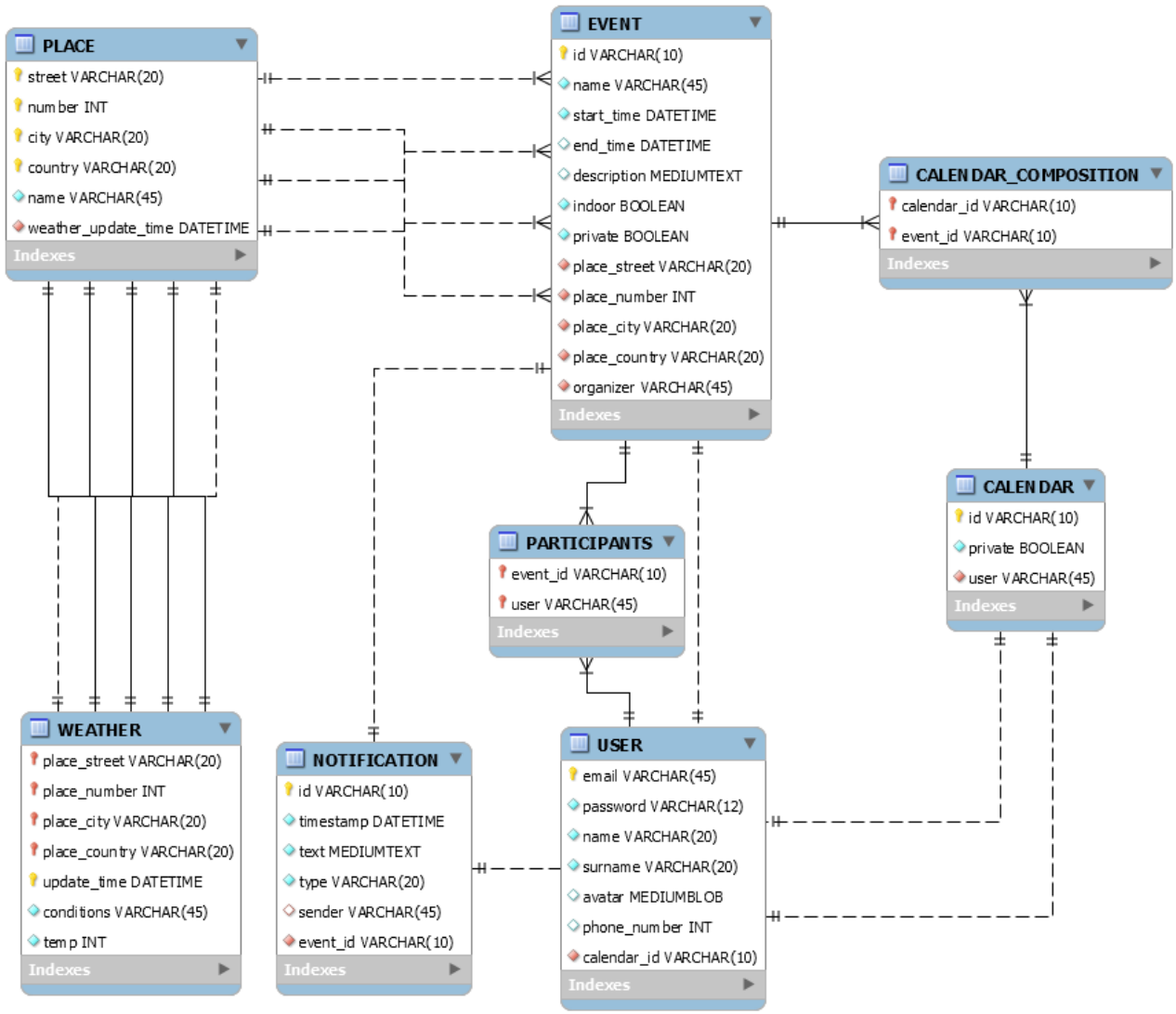
In this phase the entities and relationships present in the ER diagram are translated in the logic diagram below which corresponds to the actual structure of the relational database.

Before we start, we specify the convention used: an underlined attribute is a primary key of the entity and an attribute with asterisk is optional, in other words it can assume a *null* value.

- USER (eMail, password, name, surname, avatar*, phoneNumber*, calendarID)
- CALENDAR (ID, private, user)
- NOTIFICATION (ID, timestamp, text, type, sender*, eventID)
- ADDRESSEE (notificationID, user)
- EVENT (ID, name, startTime, endTime, description, indoor, private, placeStreet, placeNumber, placeCity, placeCountry, organizer)
- PARTICIPANTS (eventID, user)
- CALENDAR_COMPOSITION (calendarID, eventID)
- PLACE (street, number, city, country, name, weatherUpdateTime)
- WEATHER (street, number, city, country, updateTime, conditions, temp)

In the translation from the ER diagram to the logical schema, we have included *bridge tables* for the "many to many" relationships (*Addressee*, *Participants* and *Calendar_composition*) and we have removed all "one to many" relationships adding a foreign key in the entity with cardinality equal to N.

Drawn below, there is the logical model of database.



3.3 Final analysis

At the end of this analysis, we would still like to make a final consideration on the design choices made for the database: the entity *weather* is most likely overestimated for our system and is seen more as a "choice for the future". More precisely, the possibility to have an entity like that allows you to enrich the information related to weather, simply by adding attributes: in fact we could add information about the wind speed, humidity, amount of rain expected and more. However, this requires even more difficulties in managing the data base and for this reason, most likely, the weather forecast will be implemented as a simple attribute of the entity *place*.

In fact, taking the example of this XML file of a weather forecast returned by *openweathermap.org*, you might think to develop weather information as a long string and extract only the data of interest.

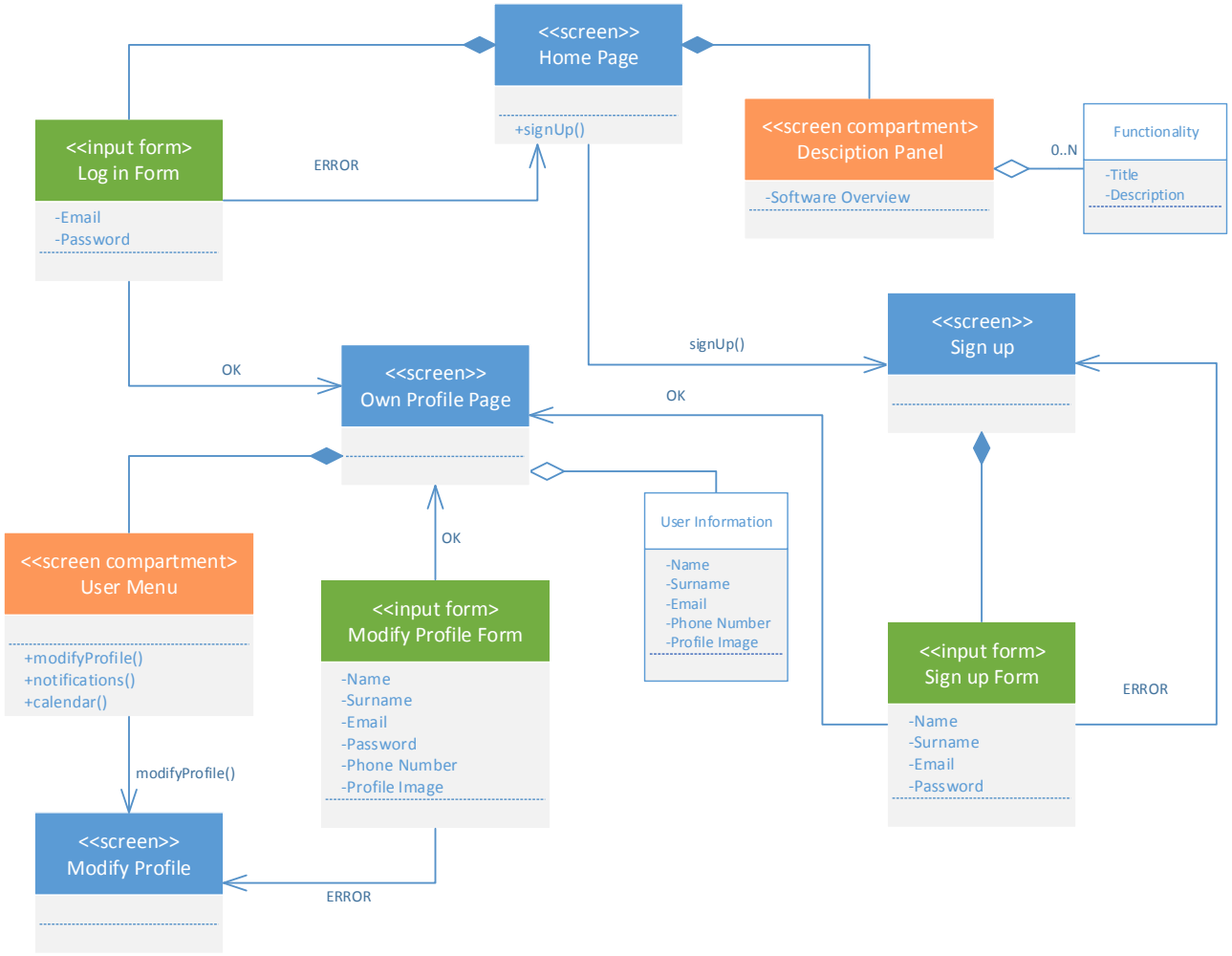
```
▼<current>
  ▼<city id="2643743" name="London">
     <coord lon="-0.13" lat="51.51"/>
     <country>GB</country>
     <sun rise="2014-12-06T07:50:53" set="2014-12-06T15:52:34"/>
   </city>
   <temperature value="276.56" min="274.95" max="278.15" unit="kelvin"/>
   <humidity value="65" unit="%"/>
   <pressure value="1027" unit="hPa"/>
  ▼<wind>
     <speed value="2.6" name="Light breeze"/>
     <direction value="240" code="WSW" name="West-southwest"/>
   </wind>
   <clouds value="0" name="clear sky"/>
   <visibility/>
   <precipitation mode="no"/>
   <weather number="800" value="Sky is Clear" icon="01n"/>
   <lastupdate value="2014-12-06T16:16:07"/>
 </current>
```

4. User experience

In this paragraph we use Class Diagrams in order to clarify our idea of the User Experience given by MeteoCal. In addition to regular classes (white colour) we used three standard type of classes in the diagrams: **<<screen>>** (blue colour), **<<screen compartment>>** (orange colour) and **<<input form>>** (green colour). The first type is used to describe the *pages* of our web application and it can contain attributes and/or actions that the user can execute to change the status of the application. The <<screen compartment>> is designed as a part of a <<screen>> and it is useful to describe those parts of page that have a specific purpose. Finally the <<input form>> has no methods and represents that part of page in which there are input fields that can be fulfilled by the user.
Likewise to the construction of Use Cases in the RASD we decided to represent through UX Diagrams the main functionalities of our system. Every presented UX Diagram describes more than one functionalities.

4.1 Log in, sign up and change profile data

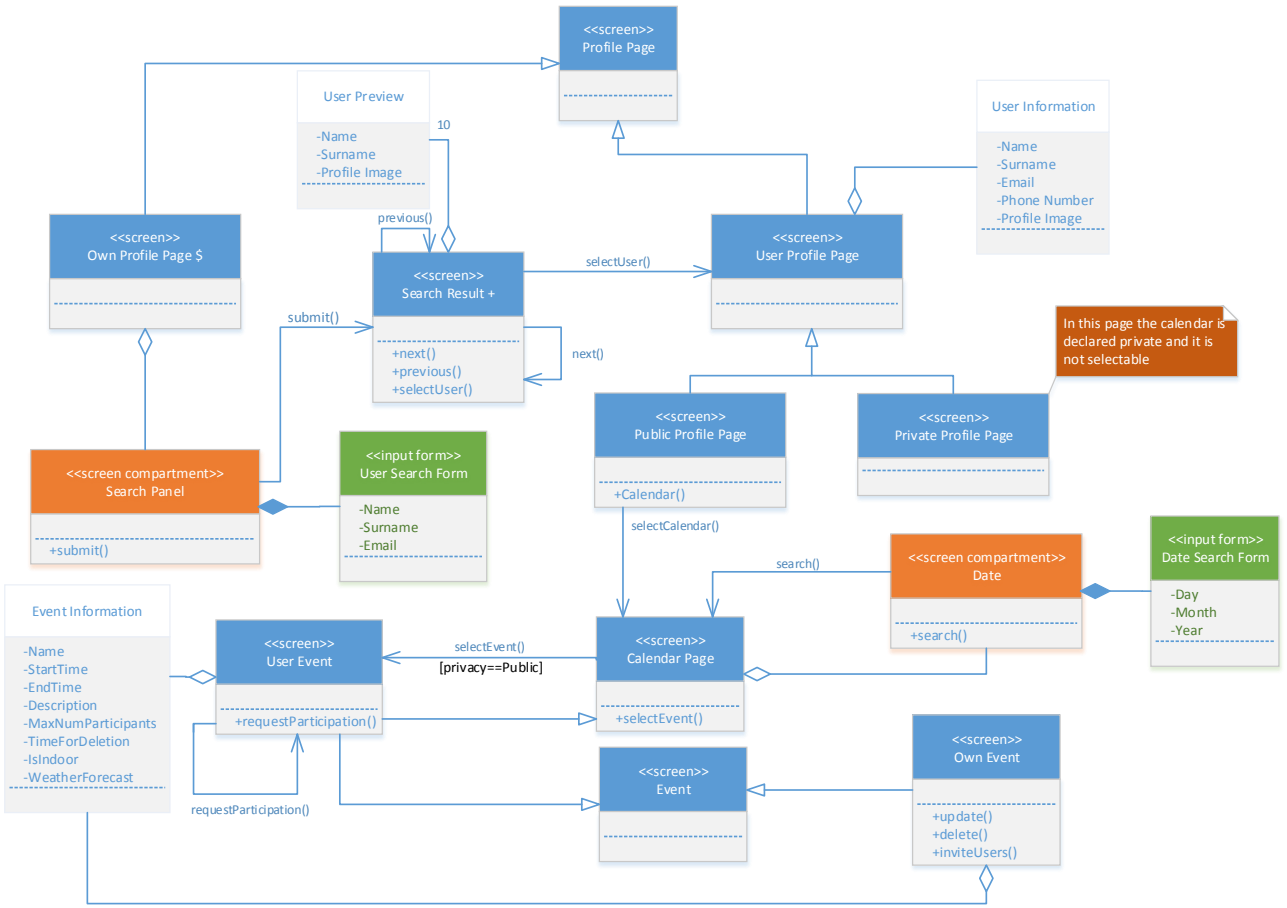
This UX Diagram shows the transition between the various states of the web application during the log in or the registration of the user, and in the case of his profile data modification. The *Home Page* contains the input form that permits to log into the system and, in case of positive result, allows the transition to the *Own Profile Page*. The profile page is also reachable from the registration process, which can be accessed from the *Home Page*. Once arrived at his profile page the user can read his own personal data and he has at his disposal a menu of the main functionalities available to him. One of these is the possibility to change personal profile data through the use of another input form.



4.2 Search user and participation request to an event

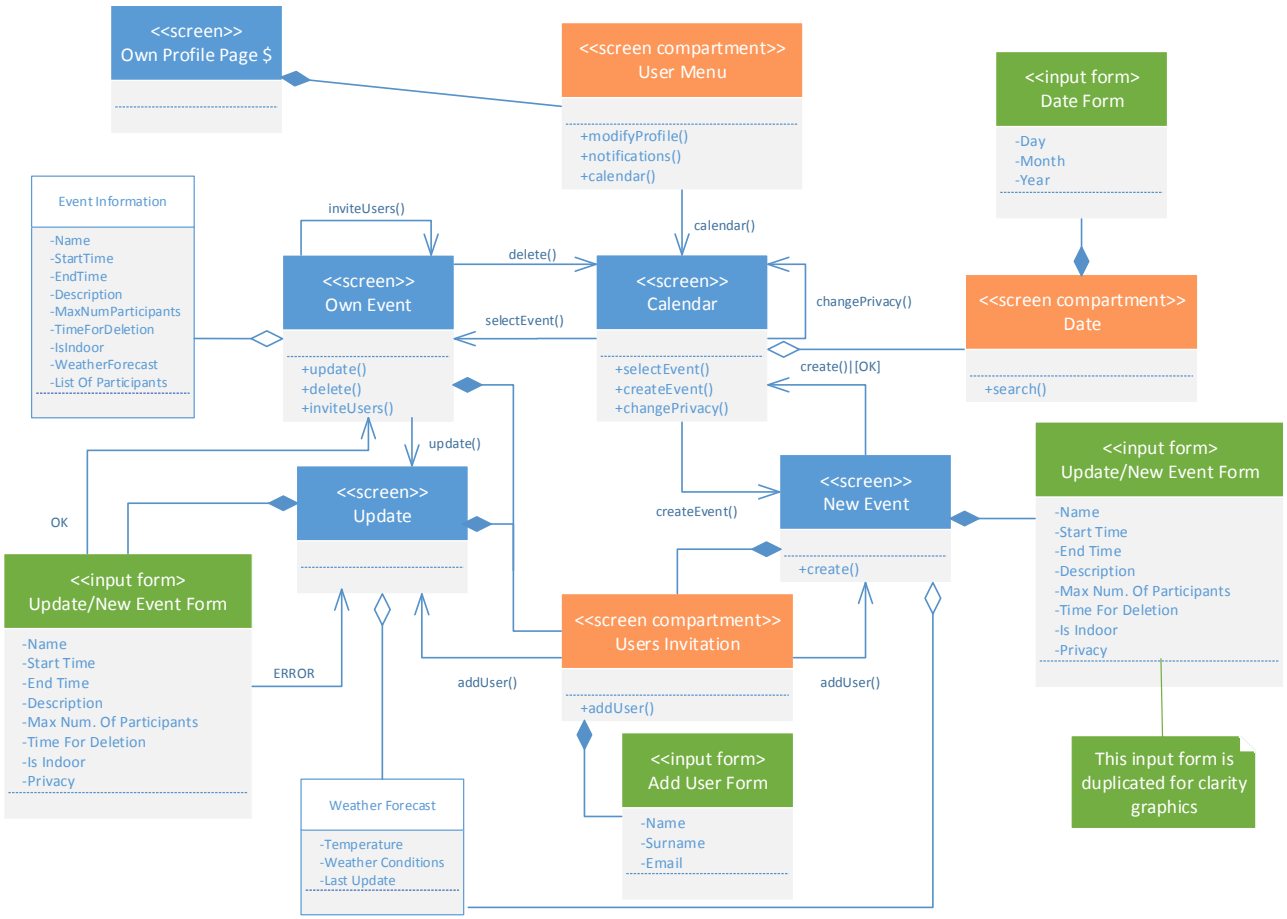
The search for a user starts from the *Own Profile Page* thanks to the *Search Panel* that contains an input form in which is possible to enter the data of the user. This process leads to a page containing the list of the users that correspond to the research parameters. From this page we can select a user and the system will show his *User Profile Page*.

The *Own Profile Page* and the *User Profile Page* inherit from the *Profile Page* because there are the same page with changes depending on whether it is your own page or not. In the *User Page* a user can only see the profile data and select the calendar, viewing it in the *Calendar Page*. The calendar is selectable only if it is declared as *public*. In the *Calendar Page* the user can select an event (if its privacy is *public*) and the system shows it in a new page. Once the *User Event Page* is loaded the user can request the participation to that event clicking on a button.



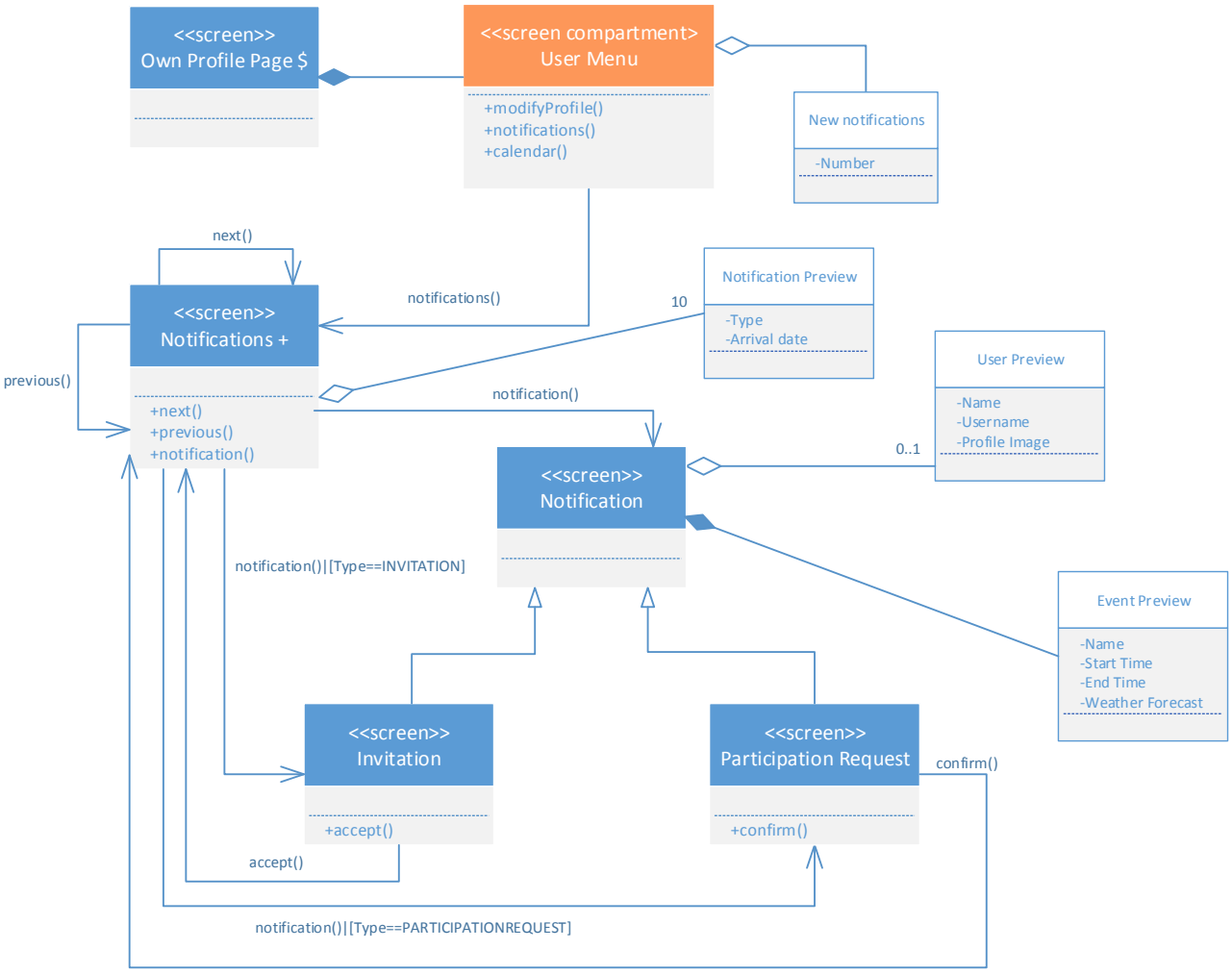
4.3 Own calendar management

In his profile page a user can choose to display his own calendar by clicking the corresponding button on the menu. The calendar will be show in another page that will allow the user to display events saved in the calendar or create a new one. If the user chooses to display an event and if he is the creator of that event then system makes available the management functionalities of the event such as: update the event, delete the event or invite users to the event. Both in the *Update Page* and in the *New Event Page* is present a screen compartment that permits to search users and add them to the event (the system will then send the invitation). Moreover both pages are composed of an input form that allows user to enter the data required to create or update event. Finally in the *Calendar Page* the user can change the calendar privacy in order to allow other users to see his calendar or not.



4.4 Notifications management

The third functionality available in the profile page menu is the visualization of user notifications by clicking on the specific button near which is present a counter of unread notifications. Once the system has loaded the page, the user displays a list of last arrived notifications with the corresponding preview (type of notification and arrival time). The user can choose one of the notifications that will be displayed in a new page where the interface will show: type and description of the notification, event associated and, if exists, the sender.



5. BCE diagrams

In order to better understand the functionalities of the system, which will be implemented according to the MVC pattern, in this section we present the Boundary-Control-Entity pattern of some MeteoCal subsystems through the standard UML.
Each class contains some methods to give an idea of how it will work communication between the Boundary (View), the Control (Controller) and the Entity (Model). These methods are generalized to be more easily understood and therefore may be different when MeteoCal will be implemented in details.

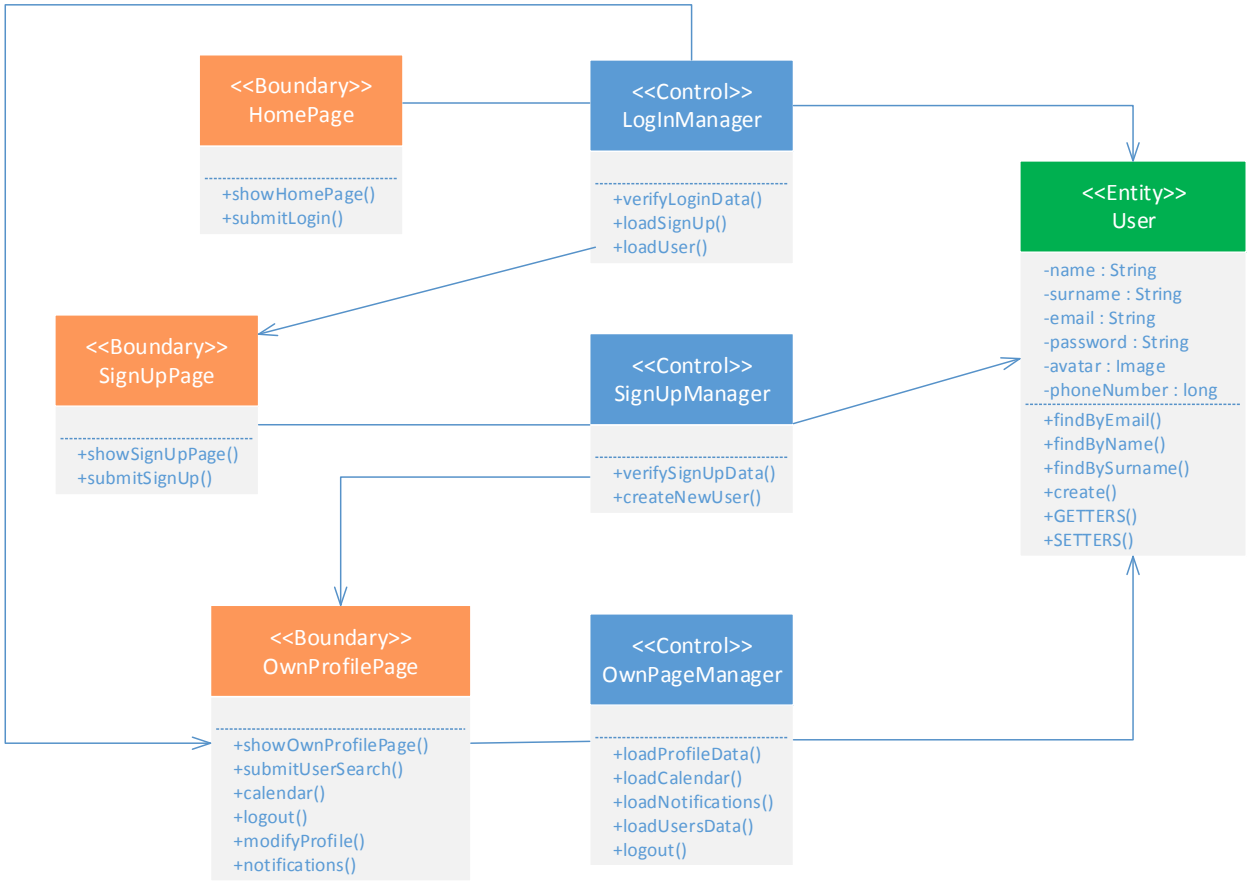
5.1 Log in and sign up

Both the log in that registration are accessed from the boundary *HomePage* that contains the screen compartment to log in and the link to the page to sign up.

In fact, the sign up is displayed in another boundary called *SignUpPage* that will allow the user to fulfil the registration form.

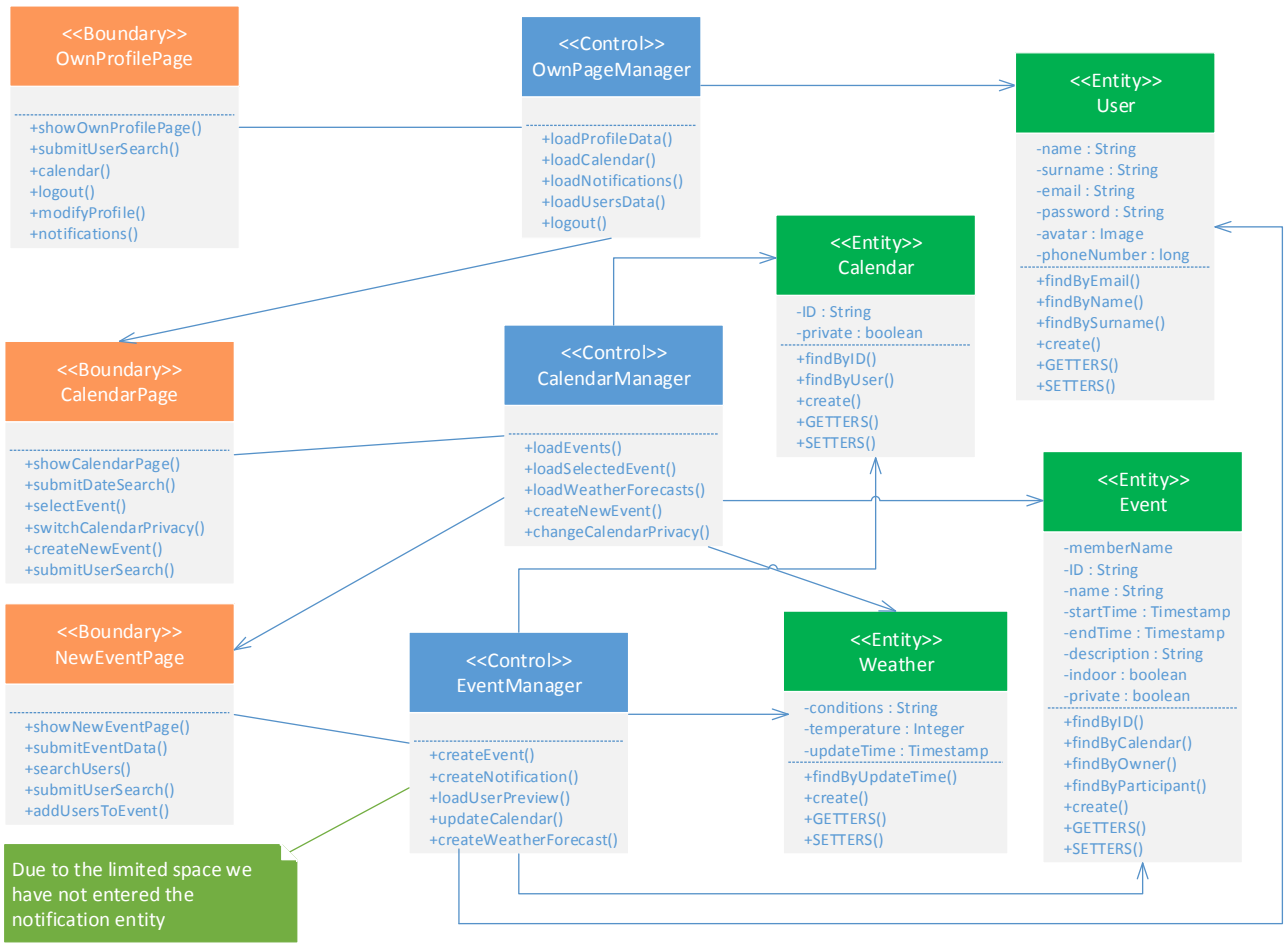
The log in and the sign up are managed respectively by the *LogInManager* and the *SignUpManager* that verify the user input data and draw the necessary information to load the respective profile page (boundary *OwnProfilePage*) where the user can view the main menu of the system and his profile data.

Finally the user data are represented by the entity *User* that is similar to the representation of the data in the database.



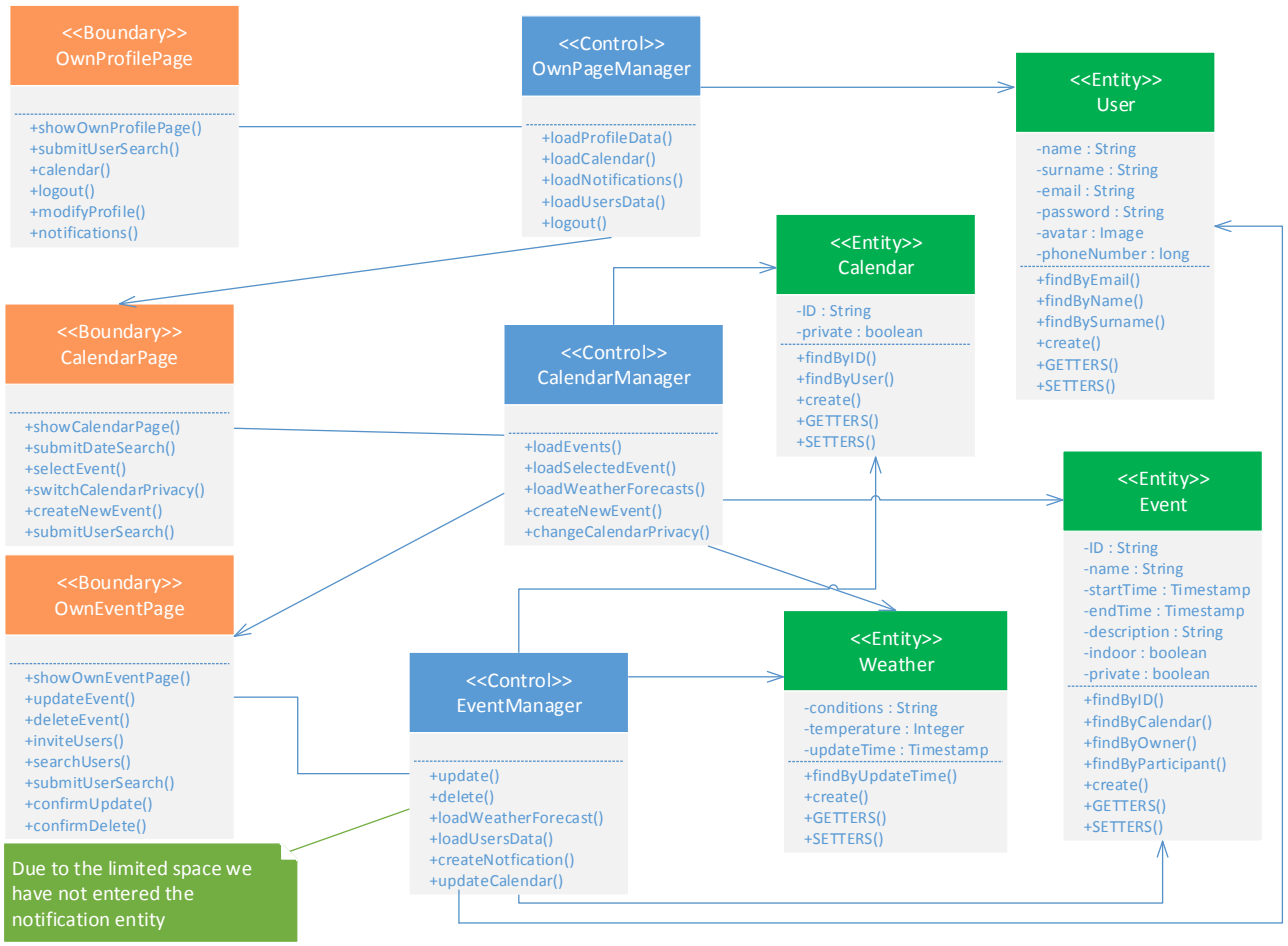
5.2 Event creation

Once selected the calendar option from the *OwnProfilePage* the controller loads the information contained in the entity *Calendar* in a new boundary: the *CalendarPage.* Here the user is able to manage the calendar and its events and to create a new event through the control *CalendarManager.* If the user chooses to create a new event, the system displays the new event form in a new page (boundary *NewEventPage*). The controller manages the functionalities of this page and loads the necessary information for the event creation such as the weather forecast from the entity *Weather* or the data of the other users that the event creator choose to add to his event. Finally the controller creates the event and generates (if necessary) the new notification (entity *Notification*) for the participation of the users and uploads the user calendar with the new event.



5.3 Event management

This BCE is similar to the previous one but it shows the functionalities for managing events that are available clicking on them. In fact, if the user chooses an event the system loads the event information in a new page (boundary *OwnEventPage*) where is possible to update or delete it and invite other users. All this functions are performed by the *Event Manager* that will then update the calendar.



5.4 Notifications

Finally this is the BCE that shows the process that allow the user to manage his notifications. The starting point is always the *OwnProfilePage* where the user can choose to view his notifications. The controller loads the notifications data from the database, represented by the entity *Notification,* and shows the notifications list in the boundary *NotificationsPage*. In order to view a notification the interface allows the user to choose one of them and the controller loads the detail in a new page (notification description, event and weather associated). If the notification is a participation request, in the page there will be the buttons for accept or decline it.