Meteocal

Design Document

Software Engineering 2

A.A. 2014/2015

Fabrizio Ferrai
Germano Gabbianelli
Andrea Grazioso

December 7, 2014

Contents

List of Figures

List of Tables

Part I
General Description

1 Introduction

1.1 Goal of this document

This document describe in detail the developement choises and the decision behind the specific implementation of the elements that compose the CalCARE architecture, the online calendar and weather forecast management platform (Meteocal).

The following description is referred to the arguments explained in the RASD document, to which the reader is redirected for a more specific debate on the system requirements, use case, logic model and software stack.

1.2 Overview of the document

The document is divided in 4 macroareas:

- in *Part I* (General Description) is described the client/server architecture and the technical choices made during developement.
- in *Part II* (Persistent Data Management) are discussed the concept and the logic structure for the database; it is also explained the real implementation of the model, in a MySQL oriented way - the DBMS which has been chosen for CalCARE
- in *Part III* (User Interface) is discussed the client architecture and are displayed some mockups of the user interface.
- in *Part IV* (Application Logic) is discussed the server architecture (*business tier*).

1.3 Goal

CalCARE aims to provide users a nice and functional way to schedule their events (be them indoor or outdoor) with the maximum flexibility of an online fully-featured calendar. In addition to that, for each outdoor event, a weather forecast (provided by the Open Weather API) is displayed, to keep the user updated about the weather situation of her event.

To reach this goal, CalCARE offers a private interface, accessible only after the creation of an account, in which users can view their personal calendar, create, modify, delete and share events.

All social funcionalities, except for the invitation system, such as text chat/messages, or friend list management, are not provided. Anyway the project is built

with modularity in mind, so it may be easily upgraded with new features in the future.

1.4 Acronyms and Abbreviations

Follow a list of the common acronyms and abbreviations used for brevity

DBMS DataBase Management System

EJB Enterprise JavaBeans

E–R Entity Relation

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

IDE Integrated Development Environment

JDBC Java Database Connectivity

JEE Java Enterprise Edition

JPA Java Persistence API

JPQL Java Persistence Query Language

Mockup Sketch of a graphical interface

MVC Model – View – Controller

BCE Boundary – Control – Entity

ORM Object-Relational Mapping

RASD Requirement Analysis and Specification Document

REST Representational state transfer

UML Unified Modeling Language

UX User eXperience

Other acronyms in the section 3.1 of the RASD.

2 Technological choices

JEE The client has imposed the usage of EJB, so our technical choices are made in consequence of that. The system is based on the JEE7 platform. In particular, in the platform will be used:

- *Enterprise JavaBeans (EJB)* for the development of the application logic
- *Java Persistence API* for the ORM interface between application logic and DBMS
- *Java API for RESTful Web Services* (JAX-RS) to realize the web interface

Reference implementarion is for Glassfish 4.0.0

MySQL DBMS choice is , yet, based on the client requirement, who imposed the usage of MySQL.

Maven The development cycle of build, packaging, deploying and testing is managed with the assistance of Maven, chosen for various reason:

- use the declarative structure and the inheritance offered by POM, useful to manage the EAR/WAR/WebApp split of the development model of EJB.
- manage in the simplest possible way dependencies, including the one related to the specific version.
- make the build process replicable and platform/IDE indipendent.

3 Description of the architecture

Considering the need of expansion and modularity of the CalCARE platform, will be adopted a *multi-tiered server-client* architecture, with the possibility to distribute components on different physical machines and thus create redundancy for every single component.

The base configuration consist in a single machine executing all the application logic and the data management – this will create less impact on the initial cost of the system.

Considered the predominant aspect of the client–server communication, it has been strategically chosen to make a *thin client*, so that all the SQL-query and elaboration part is made by the server.

CalCARE architecture will present 4 tiers:

1. *Client tier* level, in which the users will use the application. It is executed by the client, that communicates with other tiers via HTTP(S) using a web browser. The Web Browser does all the HTML rendering and Javascript scripts execution for interacting with the *web tier* and present response data to the user.

2. *Web tier* level that receives all the requests from the users through its REST endpoints, and either renders the response pages, or provides data responses in raw format (such as JSON or XML) to the *client tier*; on the other side, it communicates with the *business tier* through EJB APIs. It's executed within Glassfish.

3. *Business tier* level which encapsulates all the application logic, linking the data model to the functionality of the graphical interface. The *business tier* is built following a MVC (Model – View – Control) model, where the specific actions are managed by EJB and the entities of the DB represented as *entity beans* and abstracted by JPA.

4. *Data tier* is the data persistance level, managed entirely by DBMS. The *business tier* communicates with the *data tier* with Java standard technologies, in particular JDBC.

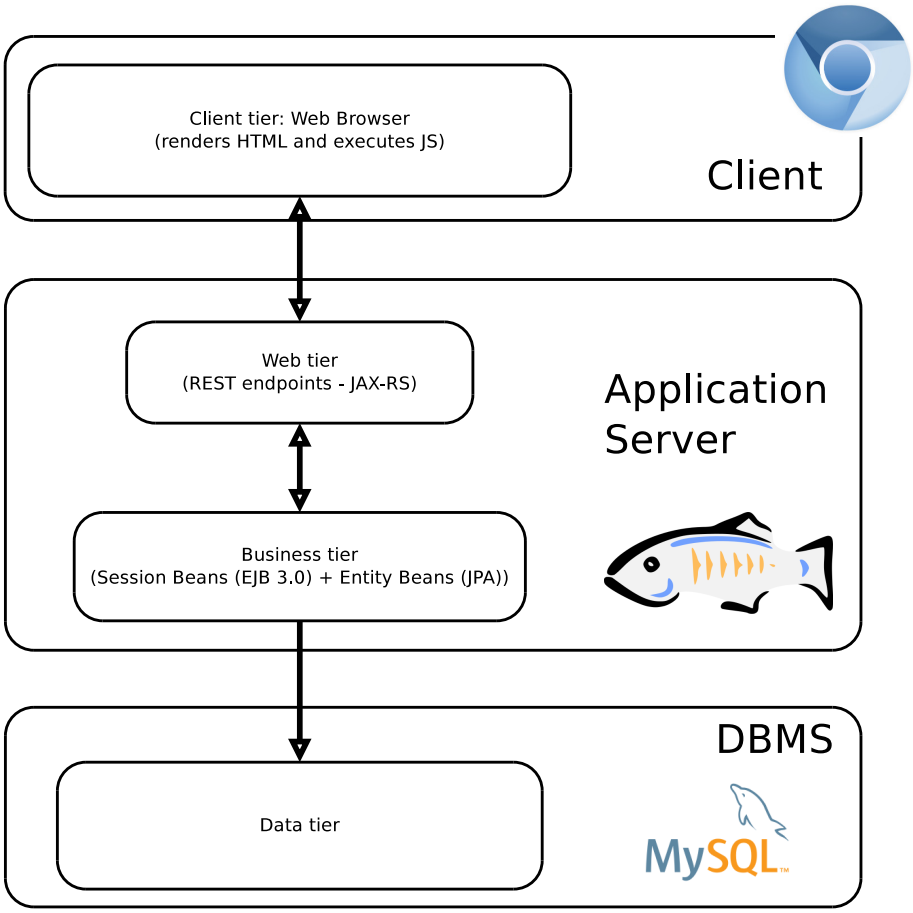

Figure 1: Generic diagram of the CalCARE architecture, detailing technologies used for each tier of the application.

Part II
Persistent Data Management

This chapter details the design of the Data Base Management System for CalCARE following the conventions and the terminology of "*Database Systems - Concept, Languages and Architectures*" [DBS] (Chapters 5, 6 and 7).

The design process has been divided in three main phases: *conceptual design*, *logical design* and *physical design*.

The purpose of the conceptual design phase is to produce a *conceptual data model*, that consists on the representation of the *information content* of the database, without considering implementation details and/or efficiency.

The purpose of the logical design phase is to translate the conceptual schema into the *data model* used by the chosen DBMS (i.e. MySQL), taking also into account possible optimizations. Since the use of MySQL as the DataBase Management System is an explicit requirement of MeteoCal, the examined data model is the *Relational Data Model*.

The purpose of the physical design phase is to translate the logical schema to a physical schema for a particular DMBS implementation. In our case MySQL.

Notation In most of the E-R diagrams of this chapter we will be adopting the Crow's foot notation to express relationships, instead of the Chen's notation referenced in the description of "*The Entity-Relationship model*" by [DBS] (Chapter 5 Section 2). This is because it offers greater compactness thus improving the readability of the diagrams and because most of the free E-R drawing tools tend to favor this notation.

4 Conceptual Design

For the conceptual design we adopted a method based on a mixed strategy, which tries to combine the advantages of the top-down, bottom-up and inside out strategies, as recommended by [DBS] (Chapter 6, Section 5). For the reader's sake we report here the main steps of the method:

1. Analysis of requirements
   (a) Construct a glossary of terms.
   (b) Analyze the requirements and eliminate any ambiguities.
   (c) Arrange the requirements in groups.
2. Basic step
   (a) Identify the most relevant concepts and represent them in a skeleton schema.

3. Decomposition step (to be used if appropriate or necessary).
   (a) Decompose the requirements with reference to the concepts present in the skeleton schema.
4. Iterative step: to be repeated for all the schemas until every specification is represented.
   (a) Refine the concepts in the schema, based on the requirements.
   (b) Add new concepts to the schema to describe any parts of the requirements not yet represented.
5. Integration step (to be carried out if step 3 has been used).
   (a) Integrate the various subschemas into a general schema with reference to the skeleton schema.
6. Quality analysis
   (a) Verify the correctness of the schema and carry out any necessary restructuring.
   (b) Verify the completeness of the schema and carry out any necessary restructuring.
   (c) Verify the minimality, list the redundancies and if necessary restructure the schema.
   (d) Verify the readability of the schema and carry out any necessary restructuring.

4.1 Analysis of requirements

We already detailed and discussed at length the analysis of the requirements in the RASD, so it will not be repeated here. However, to make this document self-contained and help the reader understanding the design, we report here (table 1) a brief summary of the *terms* used in the following sections, with a short description, the possible synonyms and a list of semantically related terms.

Term	Description	Synonym	Links
User	An authenticated user of the Application	Registered User	Guest
Guest	A non-authenticated and possibly not registered user, which has access only to Registration and Login.	Anonymous User	
Event Creator	An User who created a determined Event.		User, Event
Calendar	An User's Calendar, either public or private. Contains Events.		User, Event
Event	An Event contained in at least a Calendar. Created by an User.		Calendar, User
Participant	An User who has confirmed the Participation to an Event.		User, Event, Participation
Participation	The status of the participation of an User to an Event. Can be *Accepted*, *Declined* or *Pending*.		Invitation, User, Event
Invitation	An invitation sent from an Event Creator to any User to be invited to an Event. Can be either accepted or declined.		User, Event, Event Creator
Notification	A notification, related to a particular Event, sent by the System to an User.		User, Event
Forecast	A Weather Forecast related to a particular Event.		Event

Table 1: Glossary

4.2 Basic step

We first produced an initial Skeleton Schema, containing the principal concepts of the application as shown in figure 2.

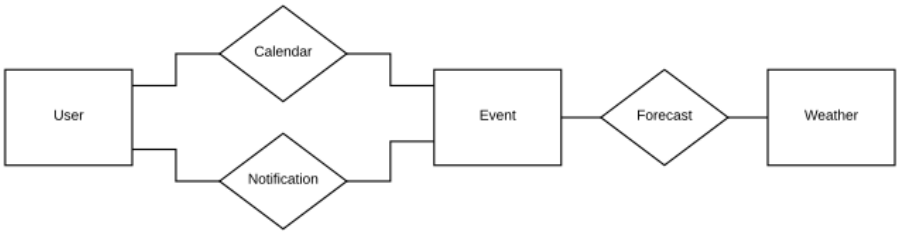

Figure 2: Skeleton Schema

4.3 Decomposition step

The Skeleton schema is easily divided into three area of interest regarding the Calendar, the Notification System and the Weather Forecasts.

Dividing the E-R into sub-diagrams allows also for better readability and understanding.

4.4 Iterative step: Calendar

The first and most important area of interest is the one about the Calendars. As the first step all the required attributes of the User and Event entities have been identified.

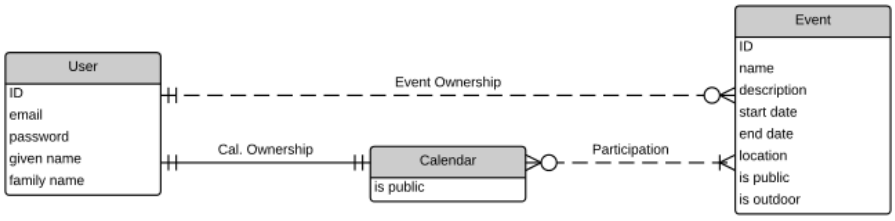

Figure 3: Calendar E-R

Entity	User
Description	An authenticated user of CalCARE
Attributes	• ID • email • password • given name • family name
Identifier	ID. Even if the email is a natural identifier for the User entity, since the identifier of the User entity has to appear in URLs, an artificial ID field has been added both for privacy and practical reasons.

Table 2: User Entity

Entity	Event
Description	An Event contained in at least a Calendar. Created by an User.
Attributes	• ID • name • description • start date • end date • location • is public • is outdoor
Identifier	ID. Event has no natural identifier, so an artificial ID must be added.

Table 3: Event Entity

Entity	Calendar
Description	A Calendar that belongs to an User.
Attributes	• is public
Identifier	External: Calendar Ownership. The Calendar is identified by the 1-1 relation with User.

Table 4: Calendar Entity

Relationship	**Description**	**Involved Entities**	**Attributes**
Event Ownership	Associates an User with the Events she created	User (1,1) Event (1,1)	
Calendar Ownership	Associates an User with her unique Calendar	User (1,1) Cal. (1,1)	
Participation	Associates a Calendar with the Event that it contains. The status can be either Accepted, Declined or Pending.	Cal. (0,N) Event (1,N)	status

Table 5: Relations of Calendar E-R

Constraints
• The email of an User must be unique. • The start date of an Event must be strictly less than (i.e. it must chronologically precede) the end date
Derivations
• An Invitation to an event is created by adding a Participation relation with status Pending between the desired Event and User to be invited. • The sender of an invitation (who invited an User to an Event) is obtained by the 1-1 Event Ownership relation between Event and User.

Table 6: Constraints and Derivation for Calendar E-R

4.5 Iterative step: Notifications

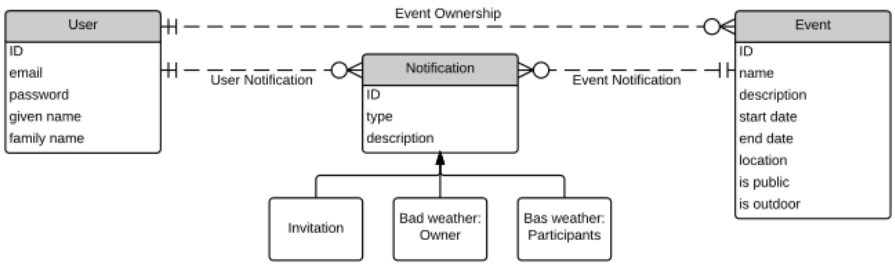

Figure 4: Notifications E-R

Entity	Notification
Description	A notification, related to a particular Event, sent by the System to an User.
Attributes	• ID • type • description
Identifier	ID

Table 7: Notfication Entity

Relationship	**Description**	**Involved Entities**	**Attributes**
User Notification	Associates an User with all her Notifications.	User (0,N) Notification (1,1)	
Event Notification	Associates an Event with all the notifications sent.	Event (0,N) Notification (1,1)	

Table 8: Relations of Notifications E-R

4.6 Iterative step: Weather

In this step we designed the ER for the Weather API subsystem of CalCARE. The external API chosen to provide weather forecast informations for the events is OpenWeatherMap [OWM], thus the structure of the following entities tries to fit the data that is returned by the service. For additional details see the documentation of OpenWeatherMap.

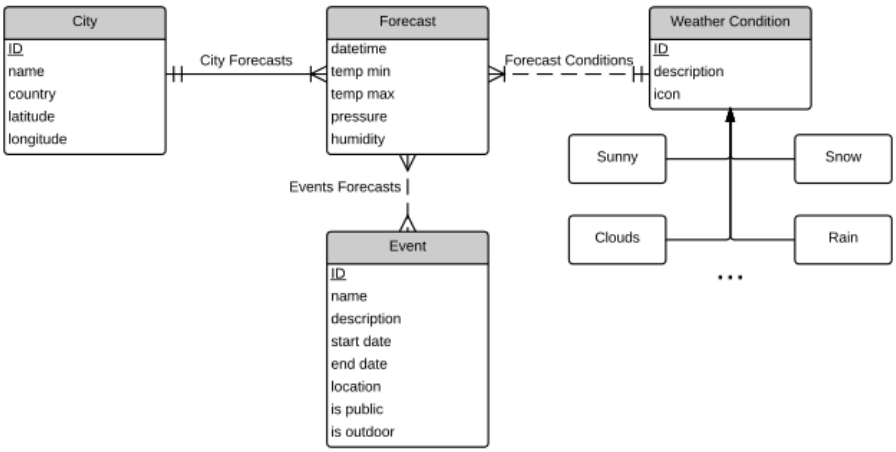

Figure 5: Weather Forecasts E-R

Entity	City
Description	A city relative to a Weather Forecast. See the documentation of OpenWeatherMap Weather Data [WD]
Attributes	• ID • name • country • latitude • longitude
Identifier	ID

Table 9: City Entity

Entity	Forecast
Description	A weather forecast relative to a particular City. See the documentation of OpenWeatherMap Weather Data [WD]. We excluded some fields from the API that were not deemed interesting for our use case.
Attributes	• datetime • temp min • temp max • pressure • humidity
Identifier	datetime and city (external). Every forecast is unique for a given time and city.

Table 10: Forecast Entity

Entity	Weather Condition
Description	The actual weather condition of a Forecast. Described as a separate entity for normalization purposes. See OpenWeatherMap Weather Conditions documentation [WC]
Attributes	• ID • description • icon
Identifier	ID

Table 11: Weather Condition Entity

Relationship	**Description**	**Involved Entities**
City Forecasts	Associates a City with all its relative Forecasts.	City (1,N) Forecast(1,1)
Condition Forecasts	Associates a Weather Condition with all its relative Forecasts.	W. Cond. (0,N) Forecast(1,1)
Events Forecasts	Associates an Event with all its relative Forecasts. Since an Event could in principle last a long time, there could be more than one Forecast associated with it.	Event (0,N) Forecast(1,N)

Table 12: Relations of Weather E-R

Constraints
• The temp min of a Forecast must be equal or less than the temp max of the same Forecast.
Derivations
• Checking if an event has "Bad weather", in order to determine if notifications have to be sent to participants and/or the owner, is done by obtaining the worst Forecast associated with the Event and checking if it has a "Bad weather" Condition.

Table 13: Constraints and Derivation for Weather E-R

4.7 Integration step and Quality analysis

We then proceeded to integrate all the sub diagrams according to the Skeleton diagram. However before showing the final diagram, in figure 7, that is the product of many iterations and decisions, we decided to also show, in figure 6, one of the first E-R diagram we produced to highlight that initial decisions are often revised and changed in the following iterations and that the errors found have been fixed. This is thanks to the process of Quality analysis in which we attempted to make the final E-R have all the following properties:

- correctness
- completeness
- minimality

- readability

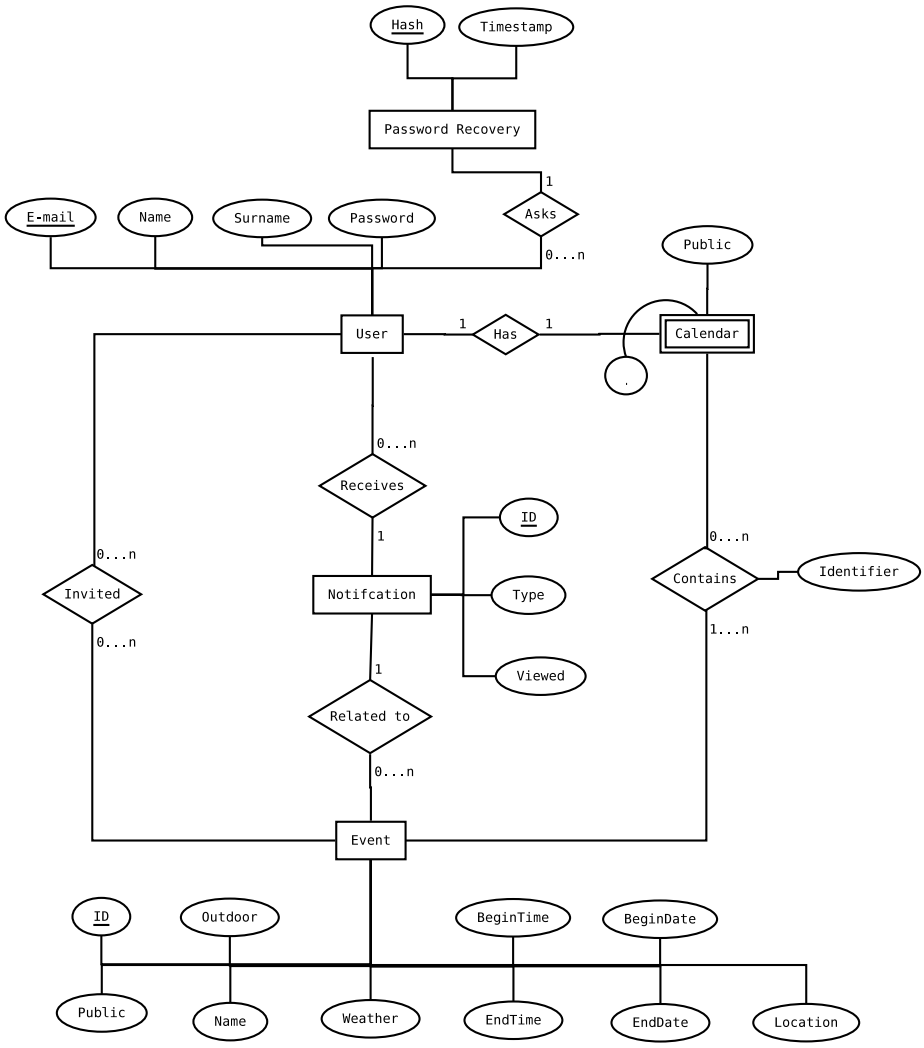

Figure 6: ER Diagram Draft

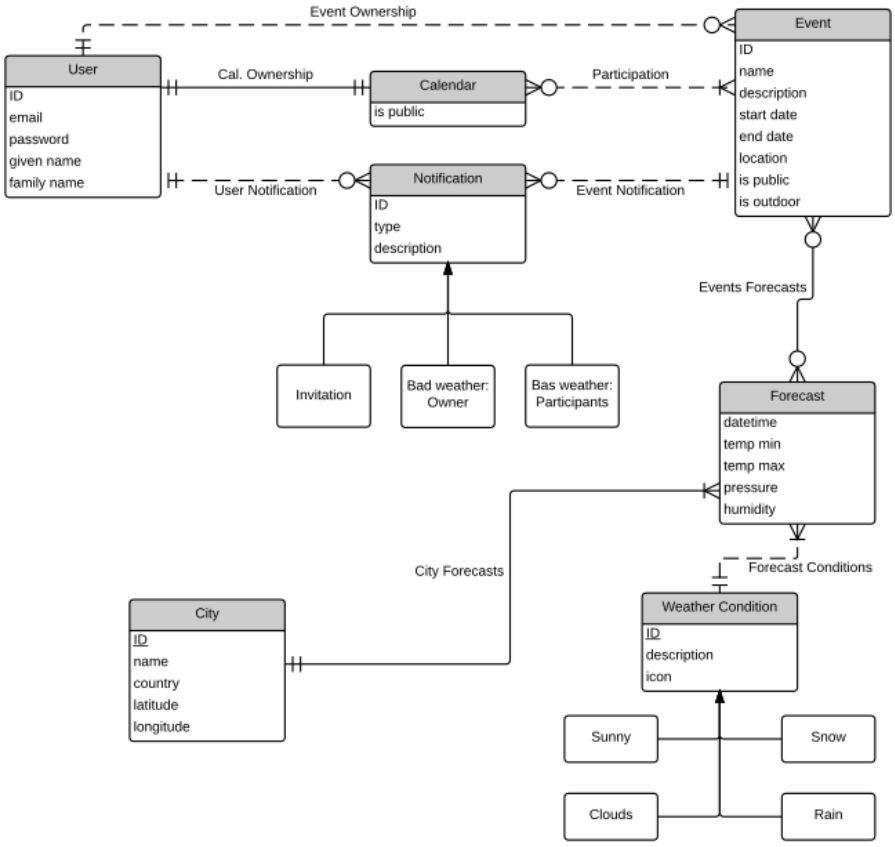

Figure 7: Conceptual ER Diagram

5 Logical Design

The conceptual design produced in the previous step has been translated to a logical schema, following the standard steps:

- Every entity is translated to a table with fields corresponding to the entity attributes.
- Every 1,N and N,1 relation is translated in foreign keys inserted in the table that has cardinality 1 in the relation.
- Every N,N relation is translated by generating a new bridge table and then considering two 1,N relations which are translated as stated above.

Generalizations and Special cases Our conceptual model contains two generalization regarding Notifications and Weather Conditions, and a relationship

with an attribute between Calendar and Event, named Participation. The following strategy was adopted to translate them:

- The generalization on the Notification entity represents all the possible notifications which must be implemented on CalCARE. Currently there are only three types of notification, but for extendibility reasons we decided to add a *notifications_types* table with a name and description attributes, to make it possible to add new notification in future releases without having to alter the schema of the database.
- The generalization on the Weather Condition entity represents all the possible types of forecast weather condition. Since there is already a table dedicated to weather conditions types (*weather_conditions*), the generalization can simply be dropped by adding a *name* field to the table, to make the different weather conditions easily identifiable.
- The Participation relationship between Calendar and Event has a status attribute. Since in our specific case the only possible choices for the status are *accepted*, *declined* and *pending* we decided to introduce a boolean field, named *accepted*, in the partiticaptions table. True means accepted, False means declined and NULL means pending. The default value is NULL (pending).

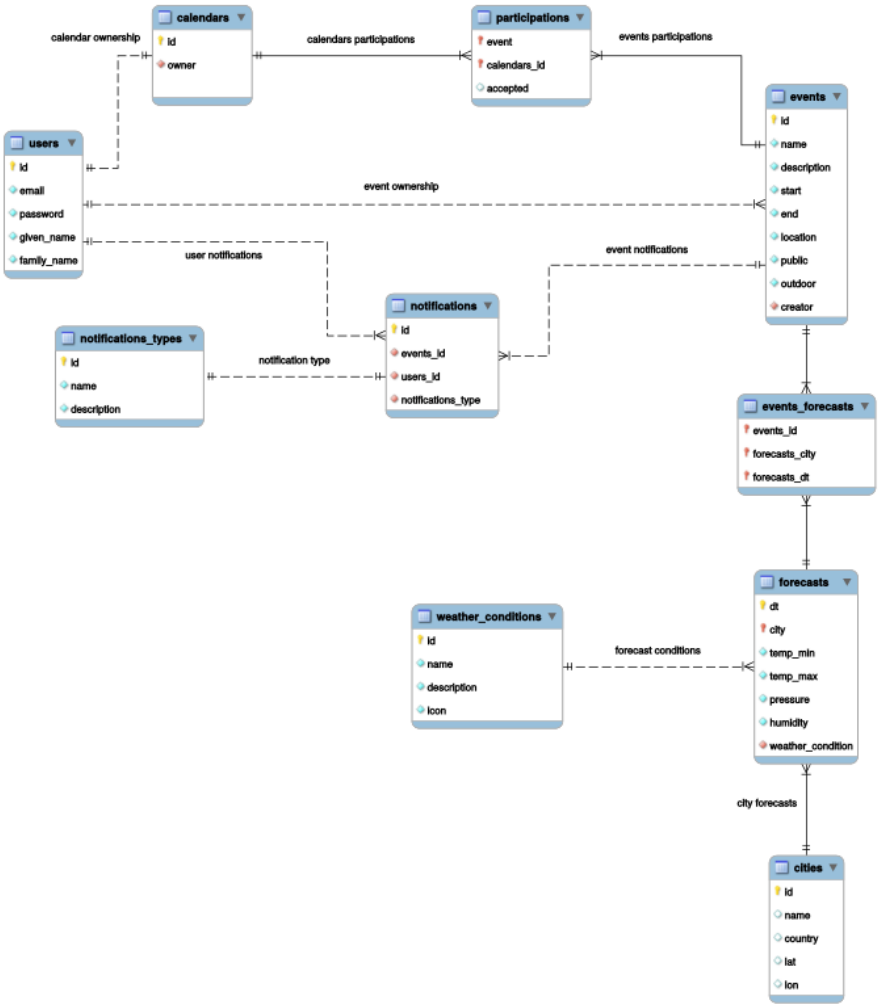

Figure 8: Logical Schema

6 Physical Design

The logical schema has been translated with MySQL Workbench to an SQL Schema specialized for MySQL.

Unfortunately MySQL does not support the SQL CHECK constraint, so it was not possible to translate some of the conceptual constraint (such as start date < end date) to MySQL. These constraint will have to be guaranteed at runtime by the Java application.

```
-- MySQL Script generated by MySQL Workbench
-- Sun 07 Dec 2014 15:05:53 CET
-- Model: New Model     Version: 1.0
-- MySQL Workbench Forward Engineering

SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS,
    FOREIGN_KEY_CHECKS=0;
SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL,
    ALLOW_INVALID_DATES';

-- -----------------------------------------------------
-- Schema meteocal
-- -----------------------------------------------------


-- -----------------------------------------------------
-- Schema meteocal
-- -----------------------------------------------------
CREATE SCHEMA IF NOT EXISTS `meteocal` DEFAULT CHARACTER
    SET utf8 COLLATE utf8_general_ci ;
USE `meteocal` ;

-- -----------------------------------------------------
-- Table `meteocal`.`users`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `meteocal`.`users` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `email` VARCHAR(255) NOT NULL,
  `password` VARCHAR(255) NOT NULL,
  `given_name` VARCHAR(45) NOT NULL,
  `family_name` VARCHAR(45) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE INDEX `email_UNIQUE` (`email` ASC))
ENGINE = InnoDB;


-- -----------------------------------------------------
```

```
-- Table `meteocal`.`events`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `meteocal`.`events` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(45) NOT NULL,
  `description` VARCHAR(255) NOT NULL,
  `start` DATETIME NOT NULL,
  `end` DATETIME NOT NULL,
  `location` VARCHAR(255) NOT NULL,
  `public` TINYINT(1) NOT NULL,
  `outdoor` TINYINT(1) NOT NULL,
  `creator` INT NOT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_events_users1_idx` (`creator` ASC),
  CONSTRAINT `event_ownership`
    FOREIGN KEY (`creator`)
    REFERENCES `meteocal`.`users` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `meteocal`.`calendars`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `meteocal`.`calendars` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `owner` INT NOT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_calendars_users1_idx` (`owner` ASC),
  UNIQUE INDEX `owner_UNIQUE` (`owner` ASC),
  CONSTRAINT `calendar_ownership`
    FOREIGN KEY (`owner`)
    REFERENCES `meteocal`.`users` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `meteocal`.`participations`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `meteocal`.`participations` (
  `event` INT NOT NULL,
  `calendars_id` INT NOT NULL,
  `accepted` TINYINT(1) NULL DEFAULT NULL,
```

```
  PRIMARY KEY (`event`, `calendars_id`),
  INDEX `fk_events_has_users_events1_idx` (`event` ASC),
  INDEX `fk_event_participations_calendars1_idx` (`
      calendars_id` ASC),
  CONSTRAINT `events_participations`
    FOREIGN KEY (`event`)
    REFERENCES `meteocal`.`events` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `calendars_participations`
    FOREIGN KEY (`calendars_id`)
    REFERENCES `meteocal`.`calendars` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `meteocal`.`notifications_types`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `meteocal`.`
    notifications_types` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(45) NOT NULL,
  `description` VARCHAR(255) NOT NULL,
  PRIMARY KEY (`id`))
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `meteocal`.`notifications`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `meteocal`.`notifications` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `events_id` INT NOT NULL,
  `users_id` INT NOT NULL,
  `notifications_type` INT NOT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_notifications_events1_idx` (`events_id` ASC),
  INDEX `fk_notifications_users1_idx` (`users_id` ASC),
  INDEX `fk_notifications_notifications_types1_idx` (`
      notifications_type` ASC),
  CONSTRAINT `event_notifications`
    FOREIGN KEY (`events_id`)
    REFERENCES `meteocal`.`events` (`id`)
    ON DELETE NO ACTION
```

```
    ON UPDATE NO ACTION,
  CONSTRAINT `user_notifications`
    FOREIGN KEY (`users_id`)
    REFERENCES `meteocal`.`users` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `notification_type`
    FOREIGN KEY (`notifications_type`)
    REFERENCES `meteocal`.`notifications_types` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `meteocal`.`weather_conditions`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `meteocal`.`weather_conditions
    ` (
  `id` INT NOT NULL,
  `name` VARCHAR(45) NOT NULL,
  `description` VARCHAR(255) NOT NULL,
  `icon` VARCHAR(255) NOT NULL,
  PRIMARY KEY (`id`))
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `meteocal`.`cities`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `meteocal`.`cities` (
  `id` INT NOT NULL,
  `name` VARCHAR(45) NULL,
  `country` VARCHAR(45) NULL,
  `lat` VARCHAR(45) NULL,
  `lon` VARCHAR(45) NULL,
  PRIMARY KEY (`id`))
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `meteocal`.`forecasts`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `meteocal`.`forecasts` (
  `dt` DATETIME NOT NULL,
  `city` INT NOT NULL,
```

```
  `temp_min` DOUBLE NOT NULL,
  `temp_max` DOUBLE NOT NULL,
  `pressure` DOUBLE NOT NULL,
  `humidity` DOUBLE NOT NULL,
  `weather_condition` INT NOT NULL,
  INDEX `fk_forecasts_weather_conditions1_idx` (`
      weather_condition` ASC),
  INDEX `fk_forecasts_cities1_idx` (`city` ASC),
  PRIMARY KEY (`dt`, `city`),
  CONSTRAINT `forecast_conditions`
    FOREIGN KEY (`weather_condition`)
    REFERENCES `meteocal`.`weather_conditions` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `city_forecasts`
    FOREIGN KEY (`city`)
    REFERENCES `meteocal`.`cities` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `meteocal`.`events_forecasts`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `meteocal`.`events_forecasts`
    (
  `events_id` INT NOT NULL,
  `forecasts_city` INT NOT NULL,
  `forecasts_dt` DATETIME NOT NULL,
  PRIMARY KEY (`events_id`, `forecasts_city`, `
      forecasts_dt`),
  INDEX `fk_events_has_forecasts_forecasts1_idx` (`
      forecasts_city` ASC, `forecasts_dt` ASC),
  INDEX `fk_events_has_forecasts_events1_idx` (`events_id
      ` ASC),
  CONSTRAINT `fk_events_has_forecasts_events1`
    FOREIGN KEY (`events_id`)
    REFERENCES `meteocal`.`events` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_events_has_forecasts_forecasts1`
    FOREIGN KEY (`forecasts_city` , `forecasts_dt`)
    REFERENCES `meteocal`.`forecasts` (`city` , `dt`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
```

```
ENGINE = InnoDB;


SET SQL_MODE=@OLD_SQL_MODE;
SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;
```

Part III
User Interface

In this part is described the User Interface of the software (so the part which is displayed in the users web browser); the aim is to provide a clear visualization of the User Experience (UX) through the application. The best way to provide this is to use a Class Diagram, with the proper UX stereotypes:

- <<screen>>: represents a page on itself
- <<screen compartment>>: represents some part of a webpage, which could eventually be shared with other pages.
- <<input form>>: is a specific representation of a web form (with inputs, checkboxes, buttons, etc.), which is used to exchange user input information with the application backend, through the form submission.

First a general UX diagram will be presented, focusing mostly on the homepages and providing an overall view of the interactions; then some of the Use Cases detailed in the RASD will be expanded, to better visualize the complex interactions going on under more specific conditions.

7 UX Overview

As it is shown in the General Diagram below, the system has two different homepages, inheriting from a generic homepage with the landmark "$", which it means it can be accessed from every page of the application.

7.1 Guest Homepage

Which homepage is viewed by the user depends on the authentication status of herself: if the user it is not logged in, then the login and signup functionalities are available, and they are organized in two compartments, and each of them has its own input form.

In case of unsuccessful login/signup the user is stuck on the guest homepage, while in case of successful login she's redirected to the user homepage.

7.2 User Homepage

The homepage of an authenticated user is organized in three main areas:

- *Navigation Bar:* contains the most useful shortcuts for the user actions:
  - *logout:* allows the user to close the actual authenticated session, and go back to the Guest Homepage.
  - *settings:* redirects the user to the account settings page, from where she can tweak some parameters and edit personal data.
  - *search bar:* provides a search functionality to reach the other users calendars (to see their public events, or to check if they are busy at any given time, if the calendar is public).
  - *create event:* triggers the creation of a new event for the calendar of the authenticated user.
- *Notifications Panel:* all the notifications regarding the user's events are displayed in this area of the screen.
- *Calendar View:* here are displayed the events for the next week/month (depending on the settings of the user)

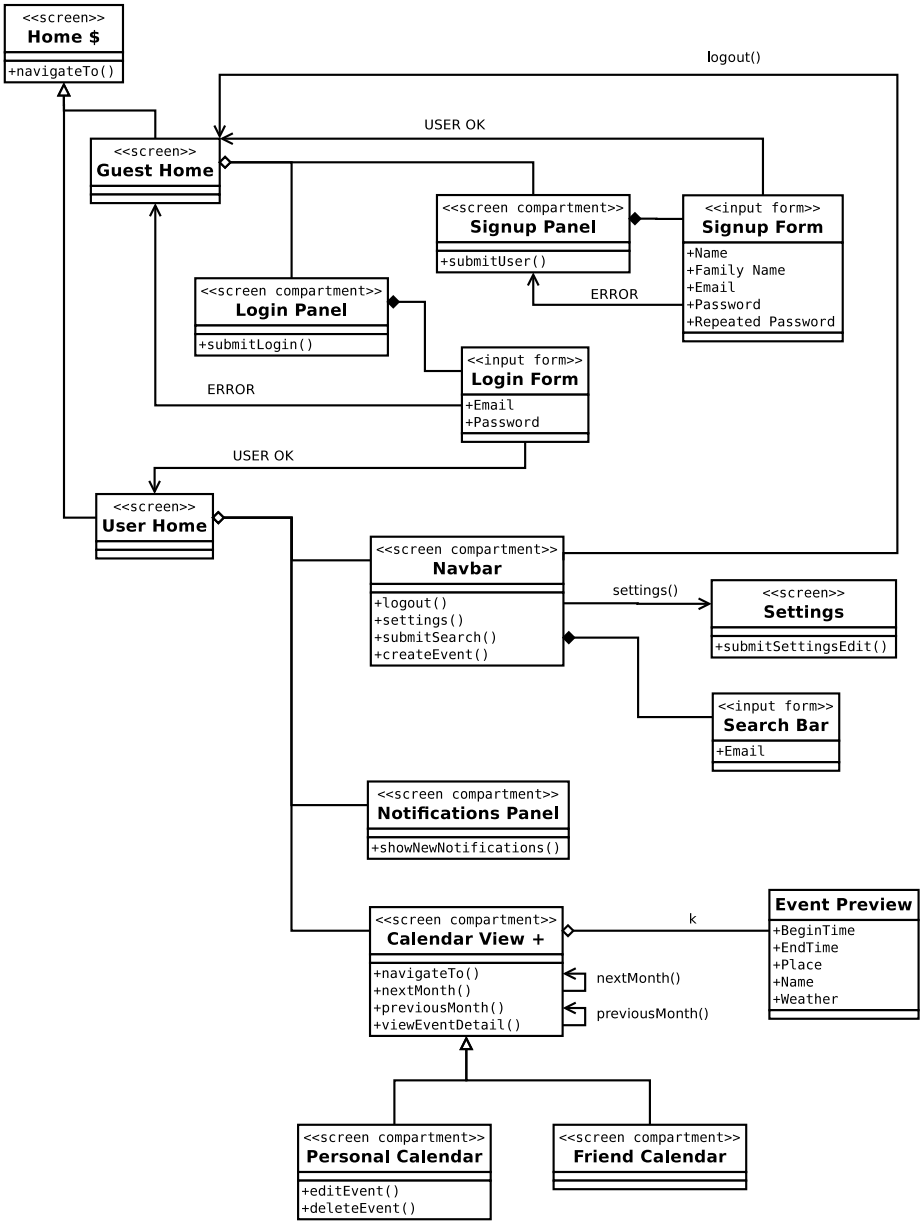

Figure 9: Generic UX Diagram of the CalCARE application, focusing mostly on the homepages and on the features overview

8 Use Cases UX

8.1 Event Creation

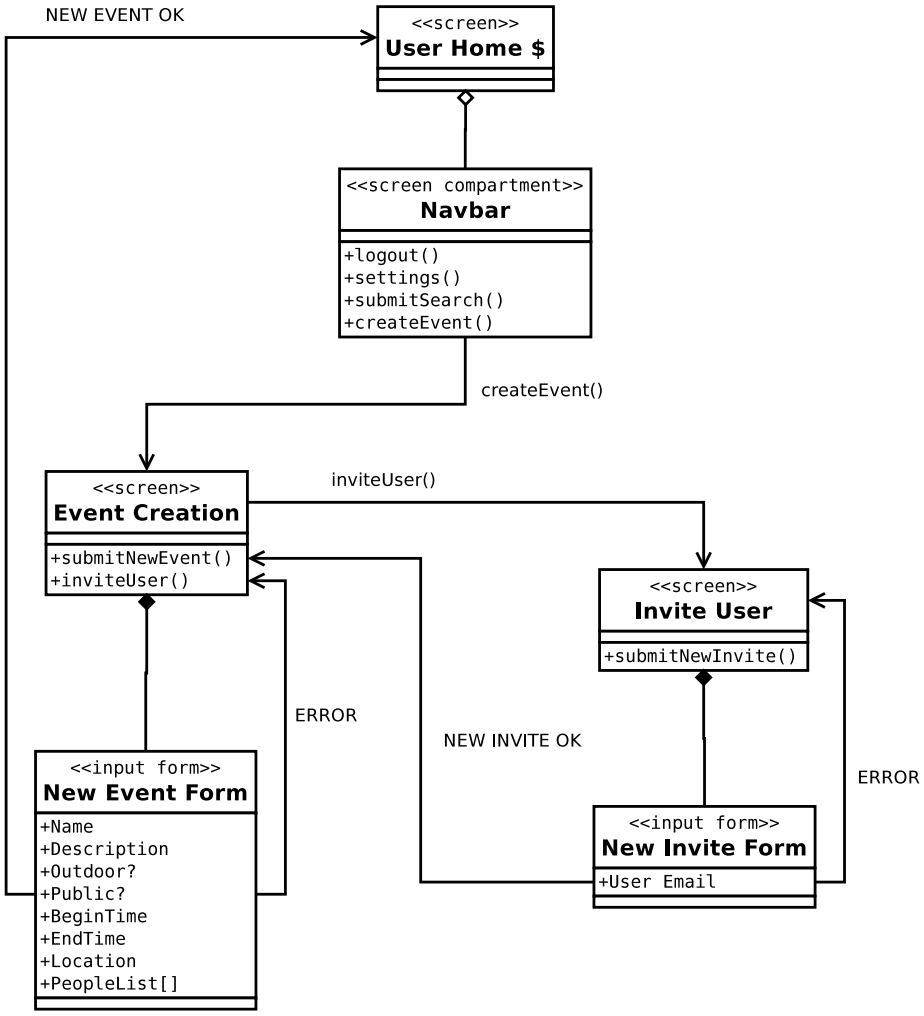

Figure 10: UX diagram of the interaction in case of the creation of a new event

8.2 Action after a Notification

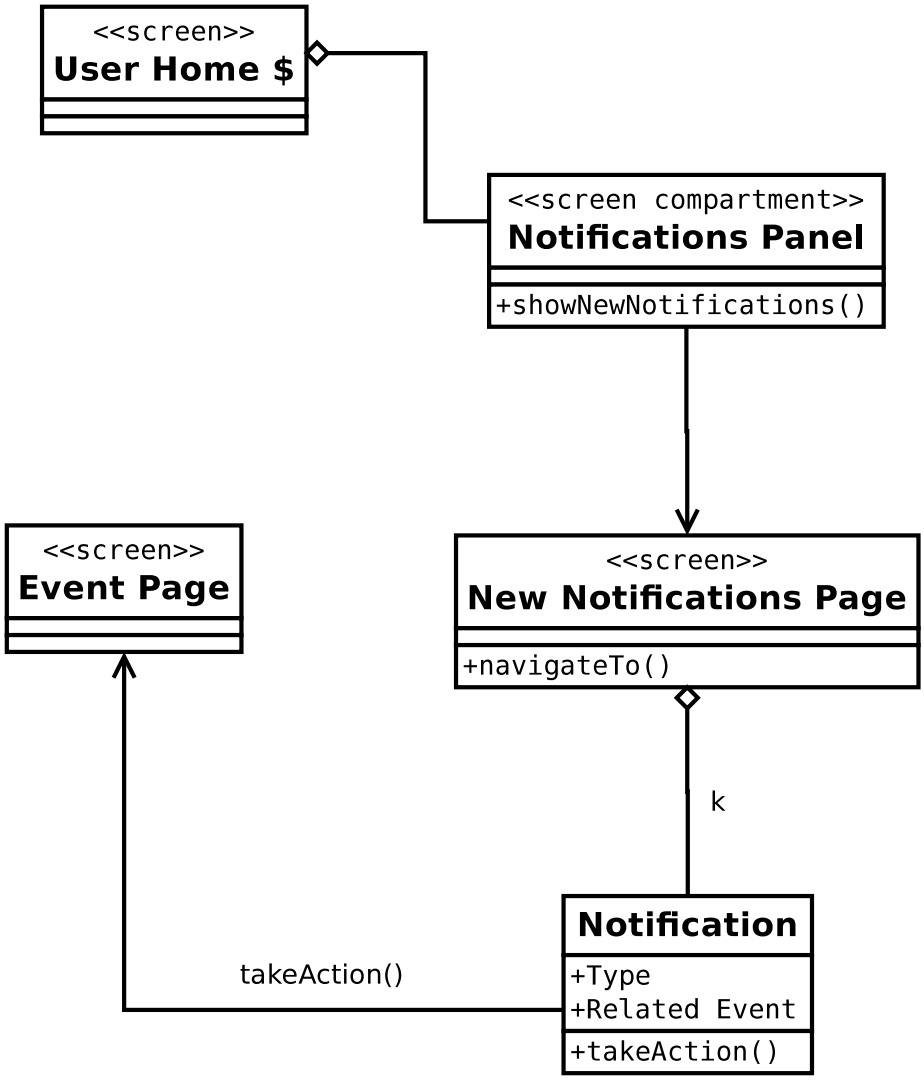

Figure 11: UX diagram of the action of responding to a new notification

Part IV
Application Logic

This section describes the technical choices relative to the implementation of CalCARE in Java, an object-orientated language, as requested by the specification.

9 Entities (*entity beans)*

The entities of the database are referred as *entity beans*, so classes which have

- as attributes, the attributes of the entities which they are referring to
- as methods, getters and setters for the attributes

Entities will be mapped in classes using JPA, that provides specific notation to manage cases, as shown below. However, the ideas are the same for any ORM.

The Database access is made by EJB and is done through JPA's *Entity Manager*. Interrogation will be made through *TypedQuery* in JPQL, simpler to write then the *cryteria query* but checked at runtime, so it will be necessary to set up all the appropriate test-cases to detect errors.

Entities *Entity beans* represent 1:1 the entities showed in the Database project. In any case, the dynamic type of the object will be determined during the query.

Entities relations Database relations are rendered as relations of inclusion among classes. In particular:

- for $1 \rightarrow 1$, an entity class will contain an attribute type of the other entity. The class which contains is the one who abstracts the natural domain of the relation.
- for the others, an entity class will contain an attribute-list of entities of the other type involved in the relation. An object-oriented language allows to represent directly the relations $n \rightarrow n$, so the link tables will not be rendered, but used to build such relations.

The UML Diagram of Entities follows, built keeping in mind the previous considerations. For the sake of brevity getters and setters will be omitted if not needed for understanding the model.

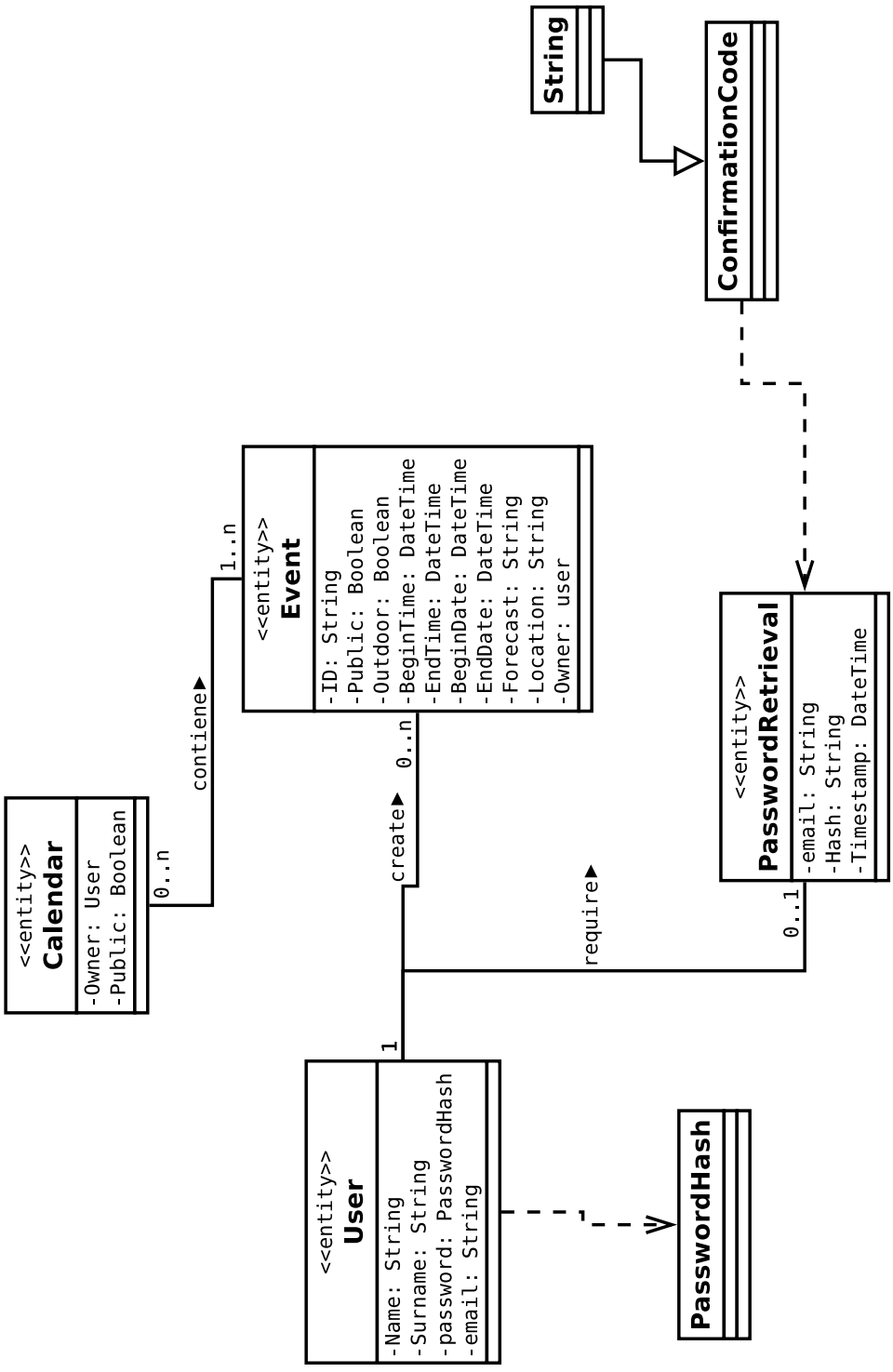

Figure 12: UML Diagram of Entities

9.1 BCE Diagram

To better lay out the structure of the MVC that is going to be implemented, we chose to represent furtherly the application design, using the *Boundary-Control-Entity* UML diagram, because is the one that maps more closely to the MVC: Entities are Models, Controls are Controllers, and Boundaries are the Views.

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 partially overlap with the representation of the interactions in the UX Diagram.

The Entities presented before do not represent the ER Diagram, but only a conceptual view of the entities used in the BCE.

9.1.1 Overview

The following diagram is a general BCE diagram that aim to give a general view of CalCARE. It's purpose is to to prevent the diagrams to be too chaotic, giving here a general view and in the following schema detailed view on the specific part of the system. In the general diagram are repesented the fundamental control system for the user and the most basic interactions with the system entities such as Calendar, Event, Profile.

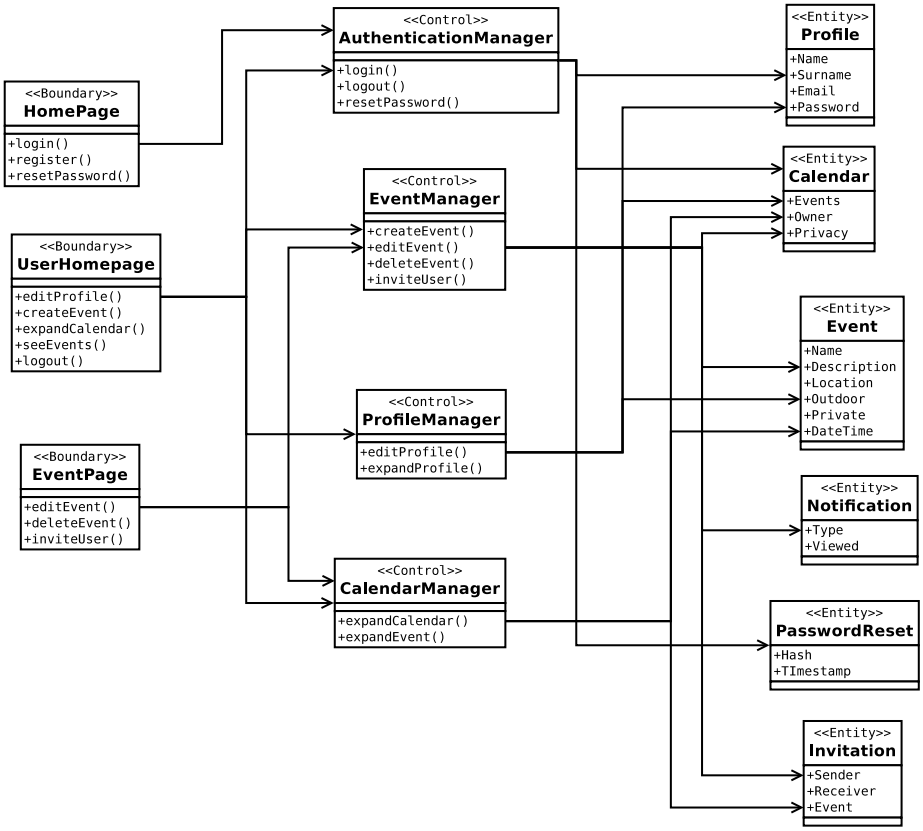

Figure 13: General BCE

9.1.2 Login Functionality

The next diagram will explain in detail the login and registration specification. In the figure below the Homepage is the screen prompter to the visitor who has the possibility to authenticate into the system (being redirected to her personal profile) or to register in the system.

At first the credentials will be validated throught comparison with the ones stored into the database (though the hash of the password will be compared, for security reasons), and sequentially all her personal data, such as the calendar, eventual notifications and the main page will be loaded and displayed to be easy reacheable and understandable.

An example of how the login manager implements an intelligent check is that it might happen that a user fills the password field with only three characters, and we know that, for instance, our password has to be at least eight characters long. In this case there is no need to look for the user in the database, because we know that the log in will be incorrect in any case).

Then the controller can load the correspondent user finding it by e-mail address and then check the password (hashed, as always).

In the visitor registration case the new user will be added into the system. Note that the entity "Person" is just fictitious, to better represent the association of an account to a physical person. In fact, since the email is unique in all the system, it is to be intended that every person should have at most one account into the system.

Also there is a simple mechanism to avoid users with a non-existent email account associated: the email address has to be verified before the user could do anything in the system, or the corresponding user will be deleted after a fixed timeout.

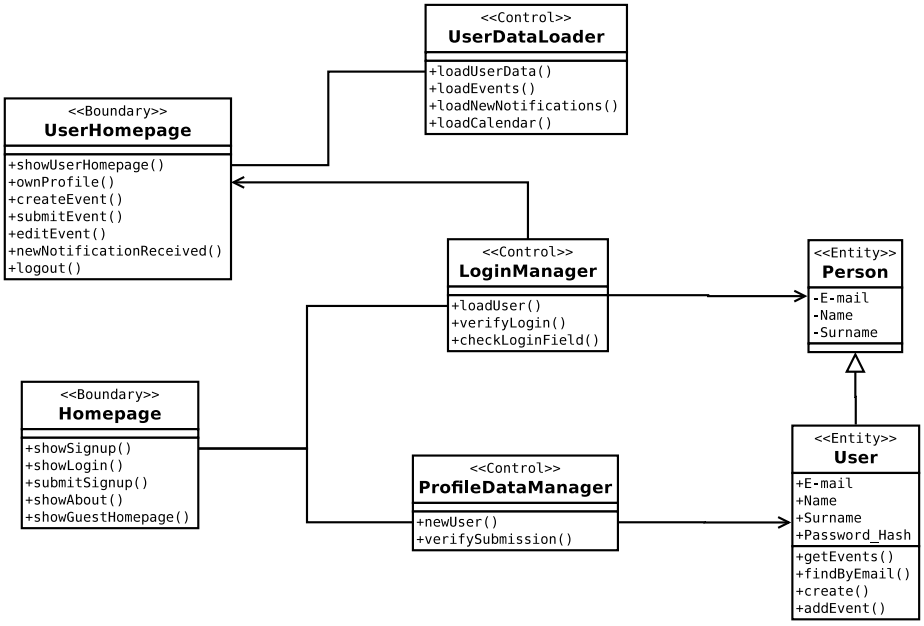

Figure 14: Login BCE

9.1.3 Invite Manager

Since there is no real "Invitation Manager" all the invitation pass through the "Event Manager" who is responsible for all the invites sent to the user when someone decides to create or edit an event adding new invited people to it.

The system will provide a simple box (see the UX detailed diagram for this use case) in which the owner of the event could identify other users inviting them to the event. They will be notified and then the same manager will be responsible for an eventual acception or denial of the invitation (this is reported in another sheet).

User Home This functionality is obviously accessible only to registered members and it's accessible from the user home.

Event Manager The system will accordingly create an event when the users requires it and simultanely will provide invitations for any number of registered users the owner of the event wants to invite.

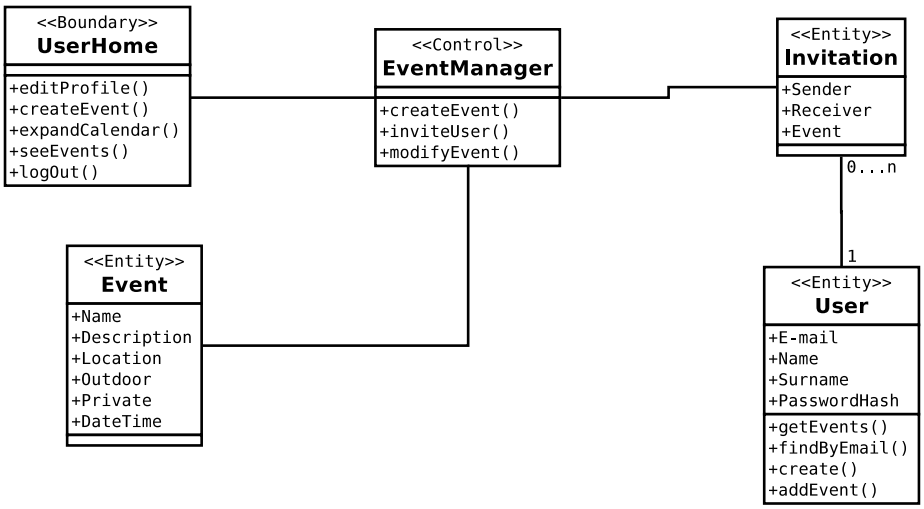

Figure 15: Invite BCE

9.1.4 Profile Managment

This part represents the only social network-like funcitonality of CalCARE. Indeed the user profile stores all the personal information about the user: Name, Surname and e-mail address. All this information may not be changed freely by users.

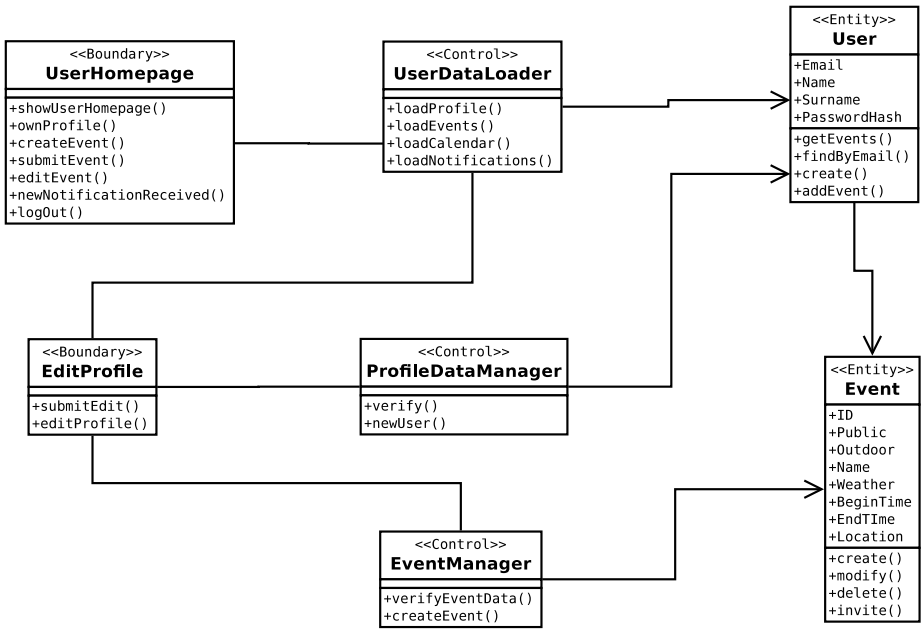

Figure 16: Profile BCE

9.1.5 Search Manager

It is possible for the user to search and view other users calendar. The process is explained in the following diagram.

User Homepage In the homepage of the user there is a search form, from which the user is prompted to insert other users email address, in order to reach their calendars. In this operation the user may be assisted by the system with an auto-complete functionality, which will help to discriminate among all possible email addesses, in order to reach the one of interest (though from the privacy point of view, this is not really nice).

User Calendar Loader is the loader of the searched user's calendar

Data Loader is responsible for loading the search result.

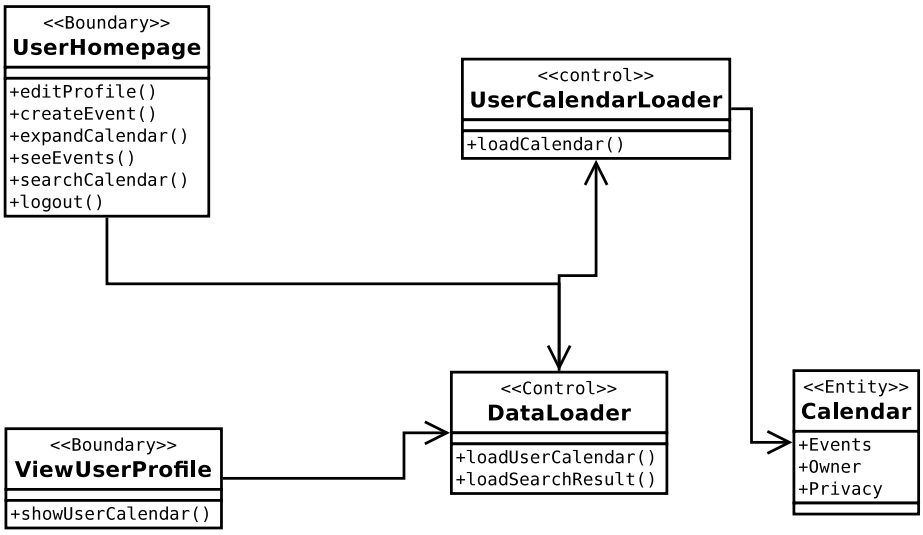

Figure 17: Search BCE

10 Application Logic (EJB)

The application logic of Calcare is managed by EJB *stateless session* –which means components without internal state that interact with the Dabatase to perform creation, modification and deletion of Events. Will be adopted a syncronous elaboration schema, because it offers much more design simplicity and is more adaptable to the request–response nature of CalCARE and, in general, of the HTTP(s) protocol through wich client and server communicate. The authorizaetion and the mantaining of the session are managed at *web tier*, in the menawhile an EJB will check the credential during login.

Registration Confermation During registration, a mail is sent to the user; it contains a link that she have to click in order to confirm the validity of the specified address. The above-mentioned link contains the e-mail address and a timestamp, all digitally signed with HMAC. In this way, is not needed to store a confirmation code in the Database, but it's only needed a boolean flag for verified email.

Password Retrieval In the event of password lost by the user, it is possible to request a reset code that will be sent to the user's email address. The code is generated randomly and it's stored in a table with a timestamp. For every not terminated new request the user will receive the same reset code, until the time limit is reached. When the process ends, the entry in the table is deleted and the password resetted.

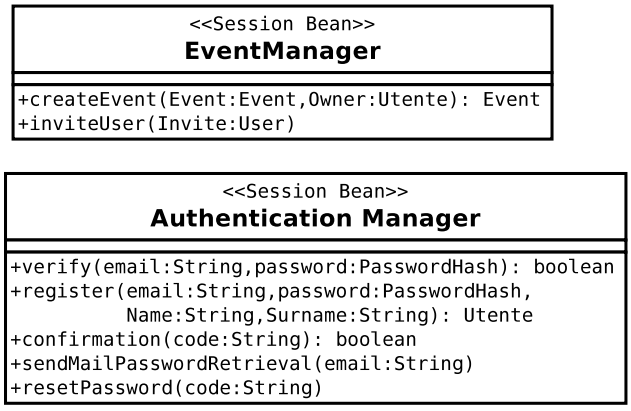

Figure 18: Diagramma UML degli EJB di TM

11 Hours Count

Table 14: Hours invested in tasks

Task	Ferrai	Gabbianelli	Grazioso
Introduction	1	1	1
Conceptual Design	5	7	5
Logical Design	2	8	2
Physical Design (Mysql)	4	10	1
UX writing	10	0	1
UX revision	3	1	2
BCE diagrams	3	1	10
Entities diagram writing	1	1	5
Application Logic (EJB)	1	1	3
Total	30	30	30

References

[DBS] *Database Systems - Concept, Languages and Architectures*
Paolo Atzeni, Stefano Ceri, Stefano Paraboschi and Riccardo Torlone
http://dbbook.dia.uniroma3.it/

[OWM] OpenWeatherMap
http://openweathermap.org/api

[WD] OpenWeatherMap | Weather Data
http://openweathermap.org/weather-data

[WC] OpenWeatherMap | Weather Conditions
http://openweathermap.org/weather-conditions