Design Document
<MeteoCal>

MATTEO GAZZETTA *837853*

ALESSANDRO FATO *838218*

December 6, 2014

Status: Final

Version: 1.0

Contents

1 Introduction **3**
1.1 Purpose of this document . . . 3
1.2 Scope . . . 3
1.3 Document History . . . 3
1.4 Definitions, Acronyms, Abbreviations . . . 4
1.4.1 Definitions . . . 4
1.4.2 Acronyms . . . 5
1.4.3 Abbreviations . . . 5
1.5 Document References . . . 6
1.6 Overview . . . 6

2 Architecture Description **7**
2.1 Application Architecture . . . 7
2.2 Execution Domain . . . 8
2.3 Application Level Details . . . 9

3 Database Model **10**
3.1 Introduction . . . 10
3.2 Conceptual Design . . . 10
3.2.1 Entity Analysis . . . 12
3.2.2 Relation Analysis . . . 14
3.3 Logical Schema . . . 15
3.3.1 Translation of ER into Logic Model . . . 15
3.3.2 Logic Schema . . . 15
3.3.3 Foreign Key Constraints . . . 17

4 Detailed Software Design **19**
4.1 Navigation Models . . . 19
4.1.1 UX EventOrganizer . . . 20
4.1.2 UX EventParticipant . . . 21
4.2 Design Analysis . . . 22
4.2.1 User Management . . . 24
4.2.2 Event Management . . . 25

1 Introduction

1.1 Purpose of this document

This document describes the design phase of <*MeteoCal*>, the project of the course of Software Engineering 2 at Politecnico di Milano. The document will explain the architectural and structural choice made during the design process that will be developed in the implementation phase. This structure is based on the specification described in the RASD.

1.2 Scope

The MeteoCal system is, as said in the previous document (RASD), an informative system that allow users to organize personal and public events, being aware of weather conditions for the events locations. This document will provide an architectural description of the user interface, business logic, data layer and user experience and application design.

1.3 Document History

Version	Change Description	Author	Date	Released
1.0	Initial Document Creation	MG AF	26.11.2014	07.12.2014

1.4 Definitions, Acronyms, Abbreviations

1.4.1 Definitions

Definition	Explanation
User	Registered and authenticated member of the application
Event	Information about location, date and participants of users appointment
Event Organizer	The User who owns the event
Event Participant	User who participates the event created by the Event Organizer
Invitation	Request of participation to an event send by the Event Organizer to the Event Participants
Calendar	The set of events scheduled for the user
Visibility	The privacy setting (Public or Private) of a calendar or an event of the system
Weather Conditions	Information about weather condition related to event
Notification	A message send from the system to the users

1.4.2 Acronyms

Acronym	**Explanation**
EO	Event Organizer
EP	Event Participant
MG	Matteo Gazzetta
AF	Alessandro Fato
RASD	Requirements Analysis and Specification Document
DBMS	Database Management System
API	Application Programming Interface
OWM	OpenWeatherMap
JEE	Java Enterprise Edition
JPA	Java Persistence API
EJB	Enterprise Java Bean
UX	User eXperience
ER	Entity Relationship
XHTML	eXtensible HyperText Markup Language
JSF	Java Server Faces

1.4.3 Abbreviations

Abbreviation	Explanation
MC	MeteoCal
DB	Database

1.5 Document References

Documents below, related to the current initiative, have been created prior to or in conjunction with the Functional Requirements document and can be referenced for further detail:

Name	Date	Version	Author	Location
MeteoCal Prj 14-15.pdf	24.10.14	1.0	Professors	Deliveries folder
RASD.pdf	16.11.14	1.0	MG AF	Deliveries folder

1.6 Overview

This document is composed by the following parts:

Introduction Gives a short description about what the system will do and describe what is the purpose of this document

Architecture Description The analysis of the system architecture that will be used and for which will be developed the application.

Database Model This section gives the description of the database design process, divided in conceptual and logical design

Detailed Software Design This section gives a detailed description about the design process with different views of the system and how a client should interact with the system using UX diagrams.

2 Architecture Description

This section describes the application architecture and the execution domain in which will be effectively executed.

2.1 Application Architecture

<*MeteoCal*>is a web application and will be developed using the Java Enterprise Edition platform and the relative technologies for the software development. The architectural style of the application that has been chosen is the Client-Server architecture. Also will be used the MVC pattern in the software.

The application is structured on three logical levels:

Presentation Level This is the level for the front-end of the application and this visualize the information and handle the user interaction with the application

Application Level This level had the job to handle the operations that can be done on the system and give to the user the functionality.

Data Level This level handle the information data of the system.

This three logic level are mapped into two physical level (tier) using the Client-Server architecture with two level:

- Client: This tier is represented by the user's machine where is mapped the presentation level of the application
- Server: This tier instead is represented by the server machine where it will be hosted the system that maps the other two logical layer (application layer and data layer)

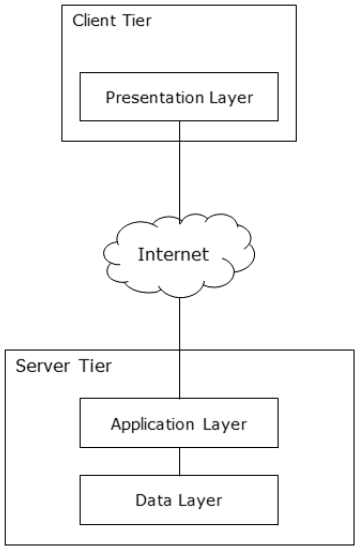

Figure 1: <*MeteoCal*>Architecture

In the client, the interaction with the user and the visualization of the application is made possible by a web browser. In the server instead the application level is managed by the application server Glassfish, while the data level runs on a MySQL DBMS. Following a more detailed analysis of the server tier.

2.2 Execution Domain

The Server domain that will execute the application have the follow characteristics:

DataBase Server	
Nome	MySQL Community Server
Versione	5.6.21
Site	http://www.mysql.com/

Application Server	
Nome	Glassfish
Versione	4.1
Site	https://glassfish.java.net/

The application runs in every operating system for which is available the application server and the database server.

2.3 Application Level Details

The application level is also structured in levels that communicate with each others. The organization of the levels in the server, that will be discussed later, depends on the implementation choice, in particular we chose the software platform of Java EE. Levels:

- **Web Component Layer**: There are XHTML pages which provides to the user a graphical interface of <*MeteoCal*>. Every page using JSF has a Named Beans associated with it, provided by the Java EE architecture. In order to manage the input from the user and to return the results to him, named bean has several methods, which are used to collect requests from the XHTML pages, compute the response and return the results to the client. In order to process requests, named beans can invoke business components when it's needed.
- **Business Component Layer**: In this component is provided all the business logic of the application and all the functionalites required by the MC application.This layer is implemented with EJB components of JEE architecture. This component receives requests from the web level and interacts with the persistence one, in order to retrieve the data from the DB and interact with the social and weather forecast API to retrieve the needed information.
- **Persistence Component Layer**: This component is useful to access the data stored in the DB by the business component. Entity beans from JEE architecture are used in order to implement this component and JPA technologies are used to map the EntityBean on scheme of the relational DB. The main purpose of this layer is mapping the relational structure of the DB in an object-oriented one to let the application easily manage these data.
- **Database Layer**: It is the DB to which the entity beans are mapped. This DB is stored in a DBMS, which is MySQL. Since it is the real DB, not a map of it, here there are all the tables in which the data are actually organized, as they are designed according to system specification.

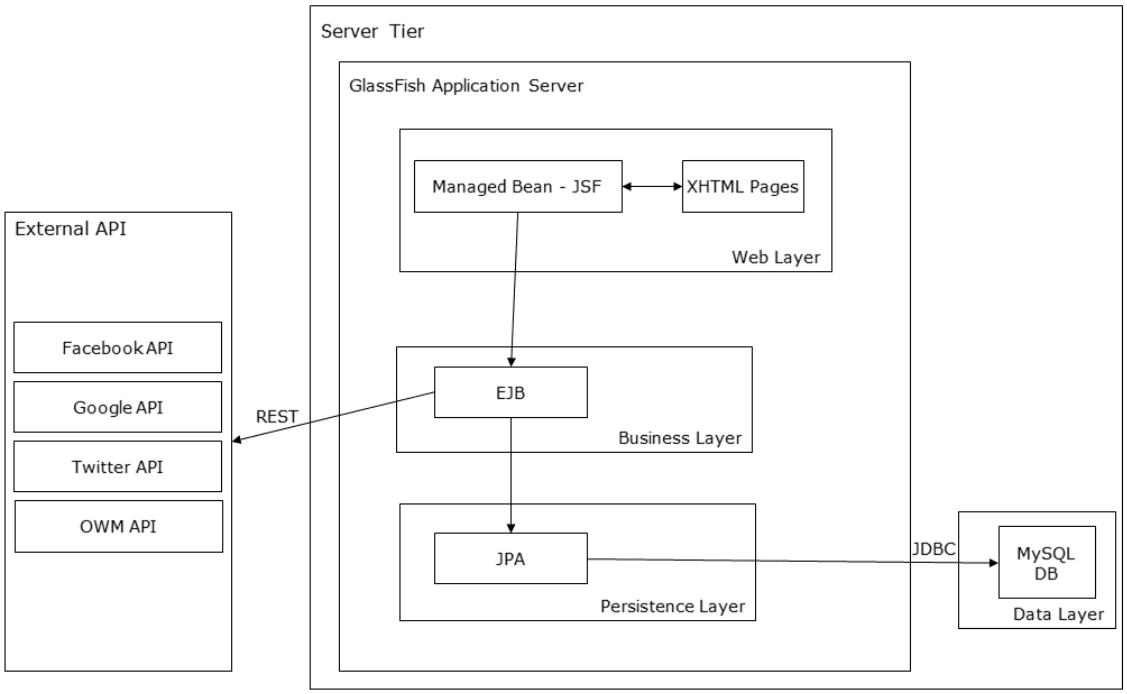

Figure 2: <*MeteoCal*>JEE Application Detail

3 Database Model

3.1 Introduction

For the structure of application data we will use a database based on a relational model. Will follow the discussion of the data structure that will be used by the application.

3.2 Conceptual Design

Conceptual design allows to start thinking about the data we want to store and about the relations between them. So the conceptual model presented is based on the Entity-Relationship schema where the rectangles represent entities, ellipses attributes and rhombus associations. It's important to note that the labels on the arcs represent the cardinality. The convention adopted is to represent the cardinality with the following meaning: Every generic package can generate 0 to N custom packages (Where N is any number), while a package customized is generated by one single generic package.

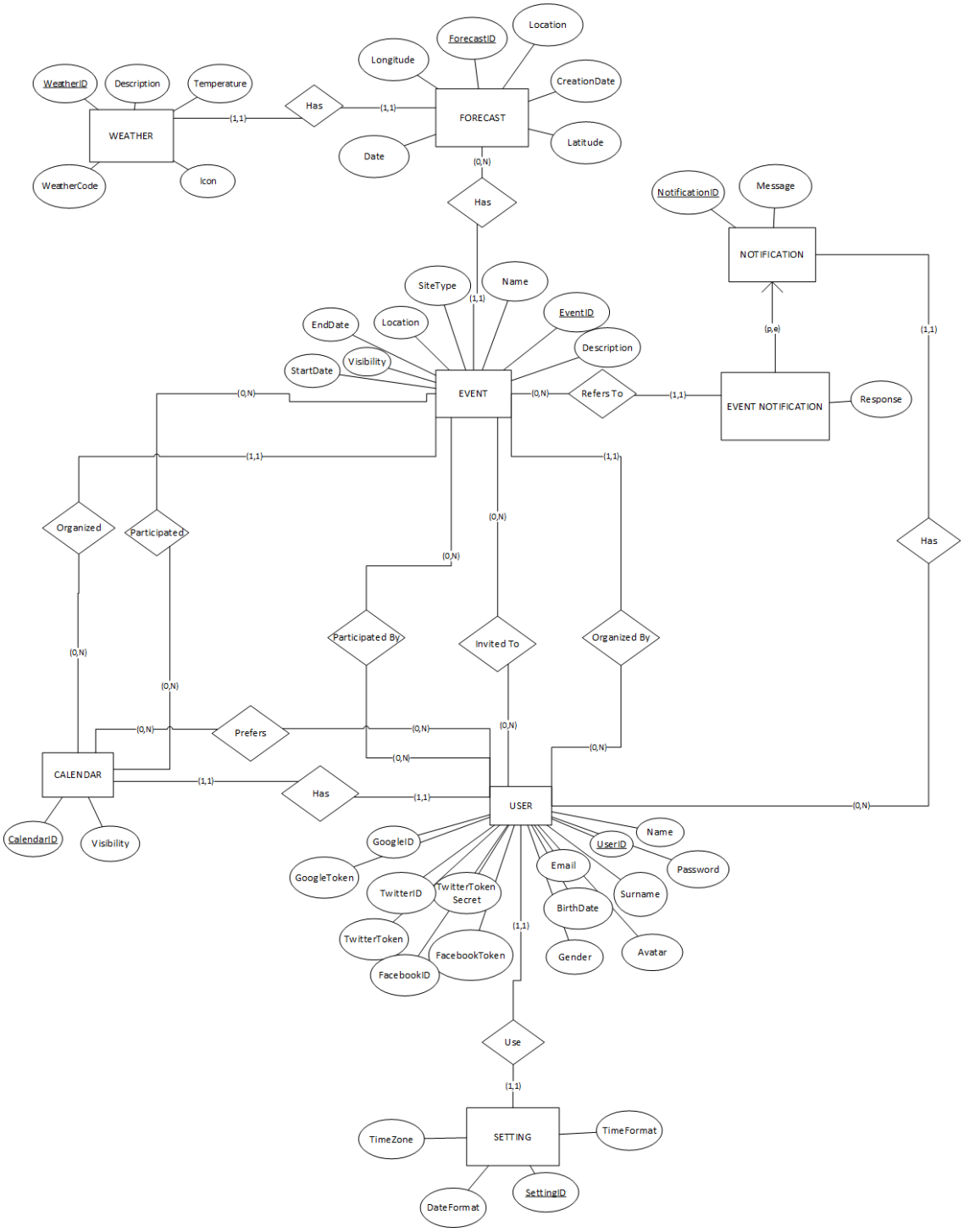

Figure 3: <*MeteoCal*>Entity-Relation Diagram

3.2.1 Entity Analysis

- **User** This entity stores all the information of the user. Its attributes are:
  - *UserID*: All the users are identified by an unique ID code
  - *Email*: The email of the user, needed to login in the application
  - *Password*: The user password, needed to login in the application
  - *Name*;
  - *Surname*;
  - *BirthDate*;
  - *Gender*: It's an enum (MALE, FEMALE) in the application.
  - *Avatar*: It's the URL of the user avatar.
  - *FacebookID*;
  - *FacebookToken*;
  - *GoogleID*;
  - *GoogleToken*;
  - *TwitterID*;
  - *TwitterToken*;
  - *TwitterTokenSecret*;
- **Calendar** This entity contains all the event of the user separated in two sets. It's attributes are:
  - *CalendarID*: All the calendars are identified by an unique ID code
  - *Visibility*: It's an enum (PUBLIC, PRIVATE) in the application and specifies if the calendar is in the public or private domain.
- **Setting** The settings of the User. Its attributes are:
  - *SettingID*: All the settings are identified by an unique ID code
  - *DateFormat*: It's an enum (DEFAULT, LONGDATE, SHORTDATE) in the application, DEFAULT: 20-Feb-2005; LONGDATE : February 20, 2005; SHORTDATE : 2/20/200.
  - *TimeFormat*: It's an enum (DEFAULT, AMPM) in the application, DEFAULT : 24h; AMPM : AM/PM
  - *TimeZone*: It's like the java.util.TimeZone in the application.
- **Event** this entity represents an event that has been created by a user in the application. Its attributes are:
  - *EventID*: All the events are identified by an unique ID code
  - *Name*;
  - *Description*;

  - *Location*: A valid location name where the event will take place
  - *SiteType*: It's an enum (INDOOR, OUTDOOR) in the application and specifies if the location of the event is outdoor or indoor
  - *StartDate*: The date and time when the event starts
  - *EndDate*: The date and time when the event ends
  - *Visibility*: It's an enum (PUBLIC, PRIVATE) in the application and specifies if the event is in the public or private domain.
- **Notification** This entity represents the notification of the application. Its attributes are:
  - *NotificationID*: All the notifications are identified by an unique ID code
  - *Message*: The message text of the notification
- **EventNotification** This entity represents notifications related to an event. Of course this is a specialization of a Notification, and so all attributes are inherited. It's attribute are:
  - *Response*: It's an enum (ACCEPTED, DECLINED, PENDING) in the application and specifies the response of the user and the participation status to an event.
- **Forecast** This entity represents the forecast information gathered from the OWM API. Its attribute are:
  - *ForecastID*: All the forecasts are identified by an unique ID code
  - *Location*: A valid location name which the forecast refers to
  - *Latitude*: The latitude of the location
  - *Longitude*: The longitude of the location
  - *Date*: The date which the forecast refers to
  - *CreationDate*: The date time when the forecast information are gathered from the OWM API.
- **Weather** This entity represent the weather information of a location in a specific forecast. Its attribute are:
  - *WeatherID*: All the weathers are identified by an unique ID code
  - *Description*: Natural language description of the weather condition
  - *WeatherCode*: The OWM Code for the specified weather condition
  - *Temperature*;
  - *Icon*: The URL of an icon that shows the weather condition.

3.2.2 Relation Analysis

- **HasWeather**: This is a relation between the Forecast entity and the Weather entity. The cardinality is (1,1) in both ways (in fact Weather it's a weak entity).
- **HasCalendar**: This is a relation between the User entity and the Calendar entity. The cardinality is (1,1) in both ways. In fact, a user can have only one calendar in the application.
- **HasForecast**: This is a relation between the Event entity and the Forecast entity. The cardinality is (1,1) from event to forecast and (0,N) in the other way. In fact, an event can only have one forecast information, but a forecast information ca be used for many events taking place in the same date and place.
- **HasNotification**: This is a relation between the User entity and the Notification entity. The cardinality is (1,1) from notification to user and (0,N) in the other way. In fact, a user can have many notification but a notification is related to only one user.
- **Participated By**: This is a relation between the entities User and Event. It's useful to tie the events to which a user participate. The cardinality is (0,N) from user to event and also (0,N) in the other way. In fact, a user can participate to different events;
- **Organized By**: This is a relation between the entities User and Event. It's useful to tie the events that a user has organized. The cardinality is (0,N) from user to event and (1,1) in the other way. In fact, an event can be organized by only one users;
- **Invited To**: This is a relation between the entities User and Event. It's useful to tie the events that a user is invited but doesn't have response to an event notification or has declined it. The cardinality is (0,N) from user to event and also (0,N) in the other way. In fact, a user can be invited to different events;
- **Prefers**: This is a relation between the entities Calendar and User. The cardinality is (0,N) from User to the Calendar and also (0,N) in the other way. This relation ties the preferred users Calendar;
- **Participated**: This is a relation between the Calendar entity and the Event entity.It's useful to tie the events to which a user is participating in its own calendar. The cardinality is (0,N) from calendar to event and also (0,N) in the other way. In fact, a event can be participated by different users so can be in different calendars;
- **Organized**: This is a relation between the Calendar entity and the Event entity.It's useful to tie the events organized. The cardinality is (0,N) from calendar to event and (1,1) in the other way. In fact, an event can be organized by only one user and can be only in the EO calendar;
- **Use**: This is a relation between the User entity and the Setting entity. The cardinality is (1,1) in both ways (in fact Setting it's a weak entity).

3.3 Logical Schema

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.

3.3.1 Translation of ER into Logic Model

First of all, we have to deal with the hierarchies. We have 1 of them:

- Notification: The Notification entity is specialized in EventNotification entity. So is possible to merge this specialization in Notification, because they doesn't have too many specific characteristics. So we had a new type attribute to identify the class of Notification.

Moreover, all the relations, which have (1,1) cardinality on one side, don't become tables in the logic schema, because they can easily be represented by adding an attribute to the entity whose side is the one with (1,1) cardinality. These relations are:

- HasWeather
- HasCalendar
- HasForecast
- HasNotification
- Organized By
- Organized
- Use

3.3.2 Logic Schema

The ER Diagram translation led to this schema:

User (ID, *CalendarID*, *SettingID*, FirstName, LastName, BirthDate, Email, Password, Gender, Avatar, FacebookID, FacebookToken, GoogleID, GoogleToken, TwitterID, TwitterToken, TwitterTokenSecret, Role)

User_PreferedCalendar (*UserID*, *CalendarID*)

User_Notification (*UserID*, *NotificationID*)

Calendar (ID, Visibility)

Calendar_OrganizedEvent (*CalendarID*, *OrganizedEventID*)

Calendar_PartecipatedEvent (*CalendarID*, *PartecipatedEventID*)

Event (ID, Name, Description, *EO*, StartDate, EndDate, Location, Site, Visibility, *ForecastID*)

Event_Participants (*EventID*, *ParticipantUserID*)

Event_InvitedUser (*EventID*, *InvitedUserID*)

Weather (ID, Description, Icon, Temperature, WeatherConditionCode)

Forecast (ID, CreationDate, Date, Latitude, Longitude, Location, *WeatherID*)

Notification (ID, Dtype, *UserID*, *EventID*, Message, Status)

Setting (ID, DateFormat, TimeFormat, TimeZone)

Where the underlined attributes are the primary keys and the italics one are the foreign keys .

3.3.3 Foreign Key Constraints

- User.CalendarID → Calendar.ID
- User.SettingID → Setting.ID
- User_PreferedCalendar.UserID → User.ID
- User_PreferedCalendar.CalendarID → Calendar.ID
- Calendar_OrganizedEvent.CalendarID → Calendar.ID
- Calendar_OrganizedEvent.OrganizedEventID → Event.ID
- Calendar_PartecipatedEvent.CalendarID → Calendar.ID
- Calendar_PartecipatedEvent.PartecipatedEventID → Event.ID
- Event.EO → User.ID
- Event.ForecastID → Forecast.ID
- Event_Participants.EventID → Event.ID
- Event_Participants.ParticipantUserID → User.ID
- Event_InvitedUser.EventID → Event.ID
- Event_InvitedUser.InvitedUserID → User.ID
- Forecast.WeatherID → Weather.ID
- Notification.UserID → User.ID
- Notification.EventID → Event.ID

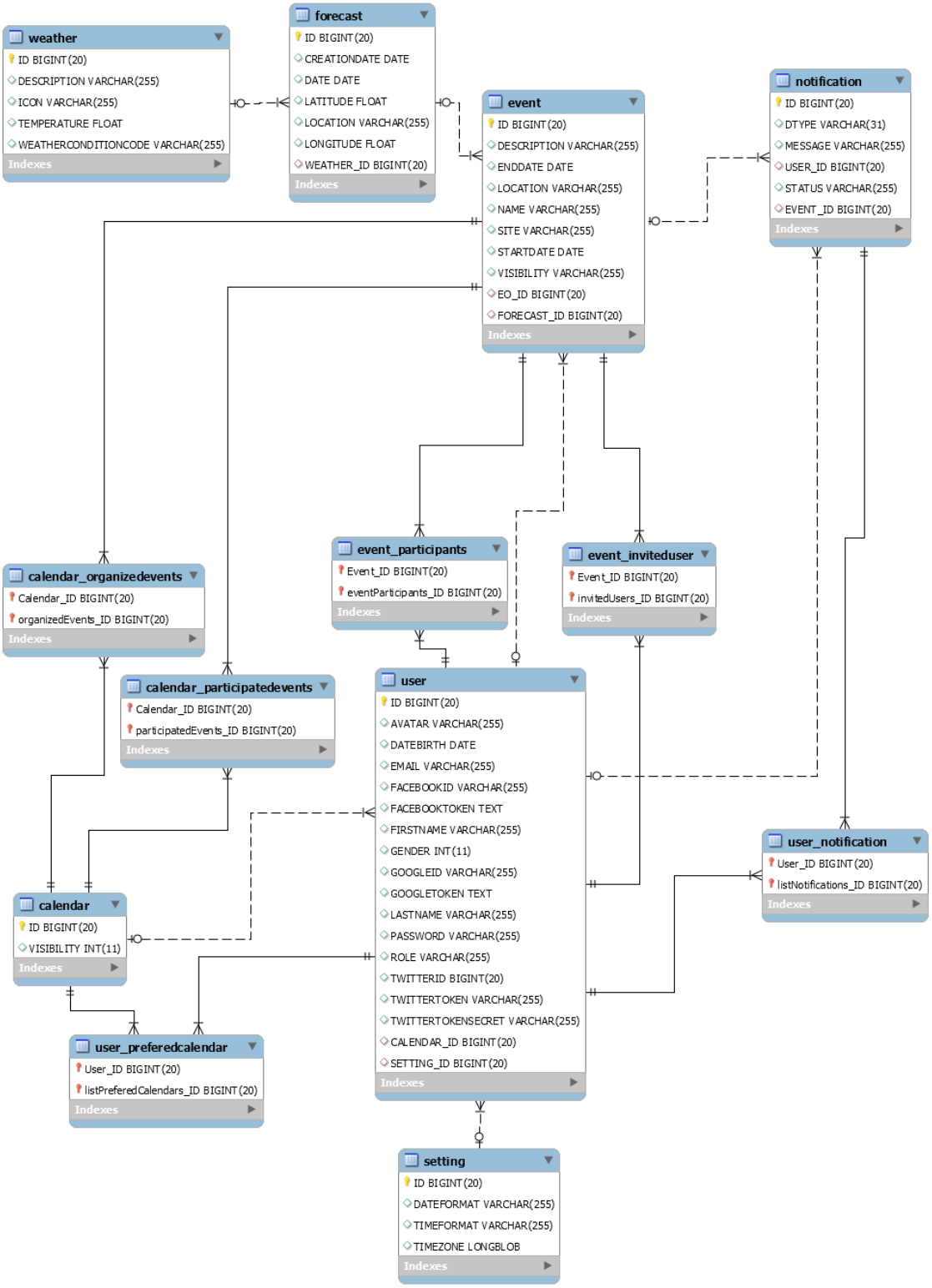

Figure 4: <*MeteoCal*>Logic DB

4 Detailed Software Design

4.1 Navigation Models

In this paragraph we want to describe the User Experience (UX) given by our system to users. We used a Class Diagram with appropriate stereotypes «screen», «screen compartment» and «input form»s and normal Classes to let understand how our Experience was thought. While «screen» represents pages which can have data and methods that the user can call, «screen compartment» represents parts of the page that can be shared with others and is included in a «screen». «input form», eventually, represents some input fields that can be fulfilled by a user (this information will be submitted to the system clicking on a button). There is also classes that doesn't have a stereotype which represent the entity containing the data that will be represented. The name of that entities refers to the entities described in the previous section of this document.

4.1.1 UX EventOrganizer

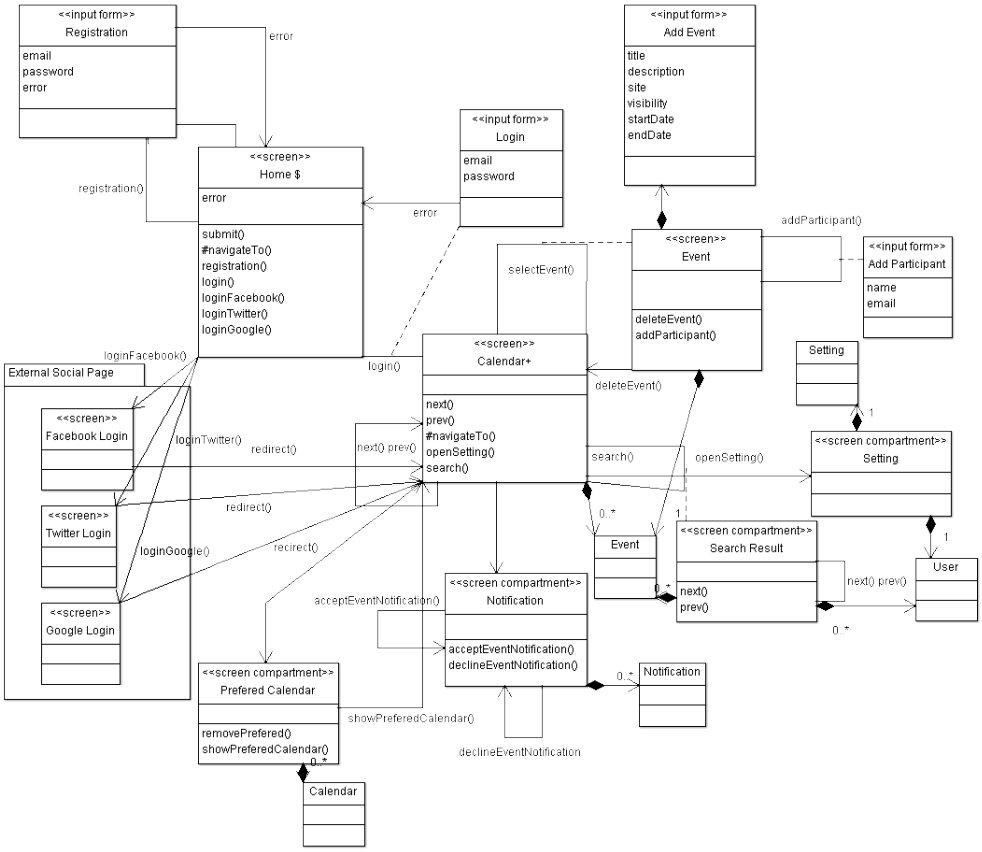

Figure 5: UX EO

4.1.2 UX EventParticipant

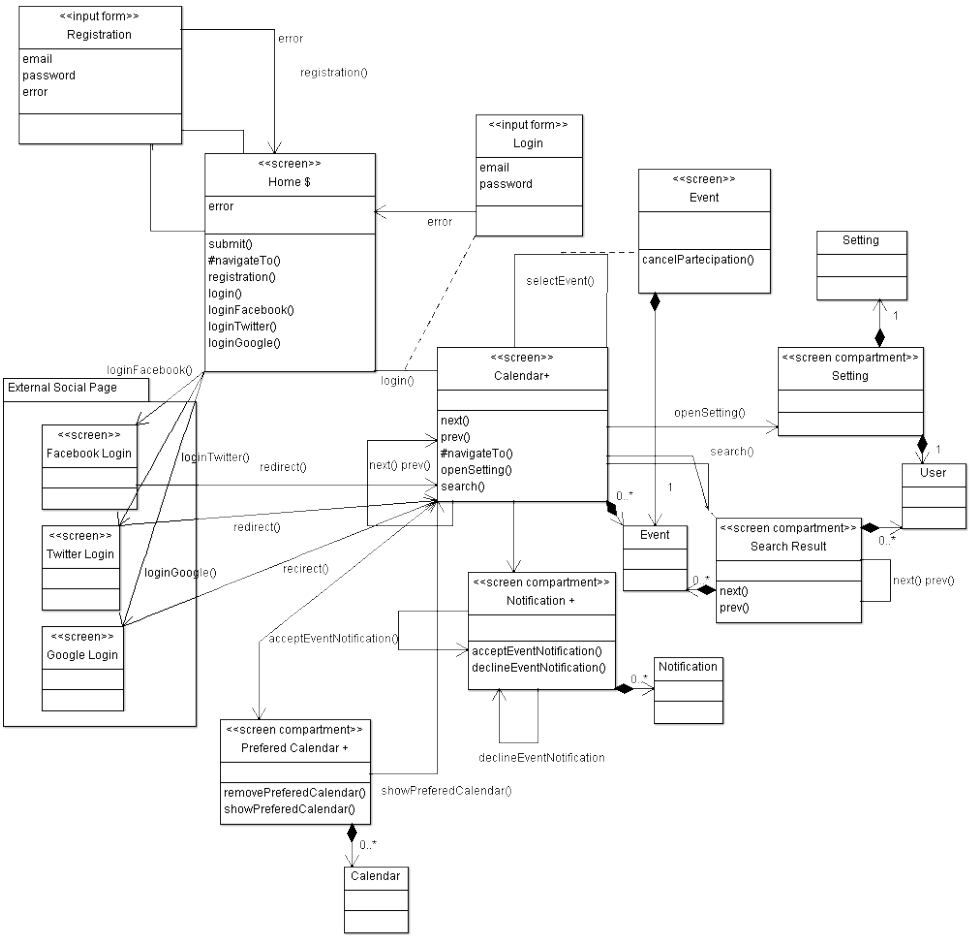

Figure 6: UX EP

4.2 Design Analysis

In order to represent the system design and structure, we use an UML diagram which shows the beans we will use in MC and how they interact. We decide not to use Boundary-Control-Entity diagram because it's useful to design software according to MVC pattern, but it's at a too high level for what we need. As you can see, after our analysis of JEE architecture we realize that MVC is already implemented in this architecture, in fact every layer can be easily put in Model, View or Controller part of the software. Briefly recalling our analysis on MVC, we see that client layer are the view, web and business ones compose the controller, and the persistence plus the database are model. Moreover, each layer has its own type of modules:

- Client $\rightarrow$ XHTML Pages
- Web $\rightarrow$ Named Bean
- Business $\rightarrow$ Session Bean
- Peristence $\rightarrow$ Entity Bean
- Database $\rightarrow$ Tables

Thanks to this fact, we can refer to a bean type and exactly know which is its place in the MVC pattern. Since the client has been already defined in the UX diagrams, and the database is largely analyzed in this document, the missing parts are the Beans. The interaction among the Beans is always the same:

1. The Named Bean collect the input from the client, validate them and send them to the session bean by an adhoc interface;
2. The Session Bean, implementing the interface methods, receives the data. After that, it realizes the business logic of the specific functionality and, if needed, it updates the DB by using the Entity Bean methods and types of data.

In some functionalites, it's required to retrieve data from the DB, not to store them. In order to implement it, the interface methods return particular objects, called DTO. They are quite complex objects in which are stored all the information of a record of a specific table. As you can see, a DTO is associated to the table whose records are represent by the DTO itself. Moreover, if you have to return a set of records, the interface methods return a collection of DTO, not a single object. So, DTO are a sort of contract between the Named Bean and the session one in order to exchange data. In fact, they are also used by the Named Bean to send data to the Session Bean. It's important to remark that DTO don't affect the implementation of the data in the single beans: in fact, in a bean you can model and work with the data as you want, the only matter is converting this representation to the DTO before communicating with others beans through the defined interface.

Moreover, all the beans dedicated to a specific functionality are more or less independent from the other ones, so we design our system as a set of interactions between beans modeled as the aforementioned schema. Of course, it doesn't mean that there is no exchange of information between the modules which implement dif-

ferent functionalites. Simply, this communication is not done among the beans, but by using the DB or the client layer. All the data are stored and retrieved from the DB, so it's quite obvious that every bean can easily access, edit or delete data written by other beans. For what concerning the data which are not stored in the DB, they can easily passed through the client layer, by associating the same XHTML page with two or more named beans. This information can be collected by the XHTML page from the named bean in which they are stored and displayed to the user. This design model can work because all the information in MC are either generated by the user or retrieved by the DB.

Moreover, since the beans are specific for every functionality and the interfaces are between these beans, of course also interfaces are exclusively dedicated to a single functionality. Otherwise, we would have a lot of methods not used in all the session beans and with a very trivial implementation. These design choices can simplify the MC structure and have several advantages:

1. We decrease the coupling between the modules, splitting them by functionalites in which they are more or less independent;
2. The design of the beans is not too complex;
3. The cohesion is ensured without adding too dependencies between the modules;

These are the design principles we use in <*MeteoCal*>. We think that such a detailed design can help us a lot in the developing step.

Here some details about the diagram conventions:

- We avoid showing all the getter and setter, of course every bean has them but they are trivial, so we hide them in order to decrease the visual complexity;
- We omit the attributes of the bean, because we are interested in the interaction among the modules, not in so specific implementation details;
- The entity beans have no methods because they are a simply mapping of the DB tables, and so their methods are simply method to interact with the DB (they are not involved in design choices, they are standardized way of JEE to access the DB);
- The DTO definitions aren't show because they are a trivial mapping of attributes of a table, and in this phase we aren't interested in their actual implementation. We define their conceptual structure, we define how to use them, but we don't need to know if an attribute is represented by an integer or a string.

4.2.1 User Management

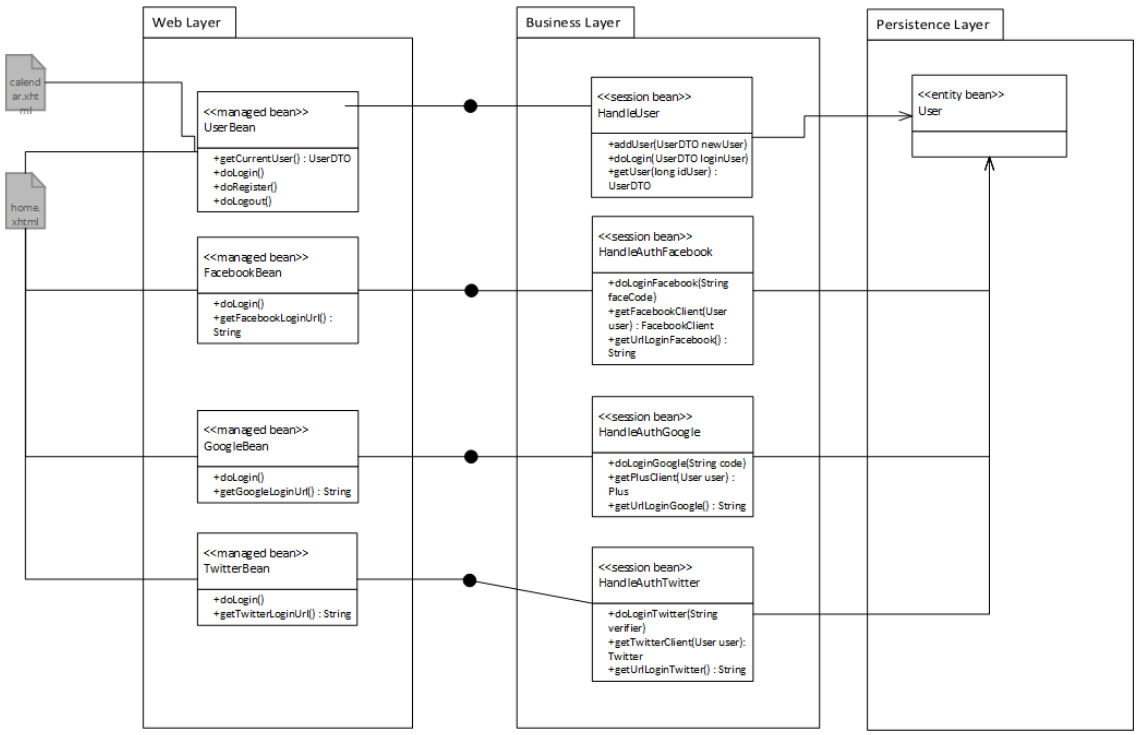

Figure 7: User Management

4.2.2 Event Management

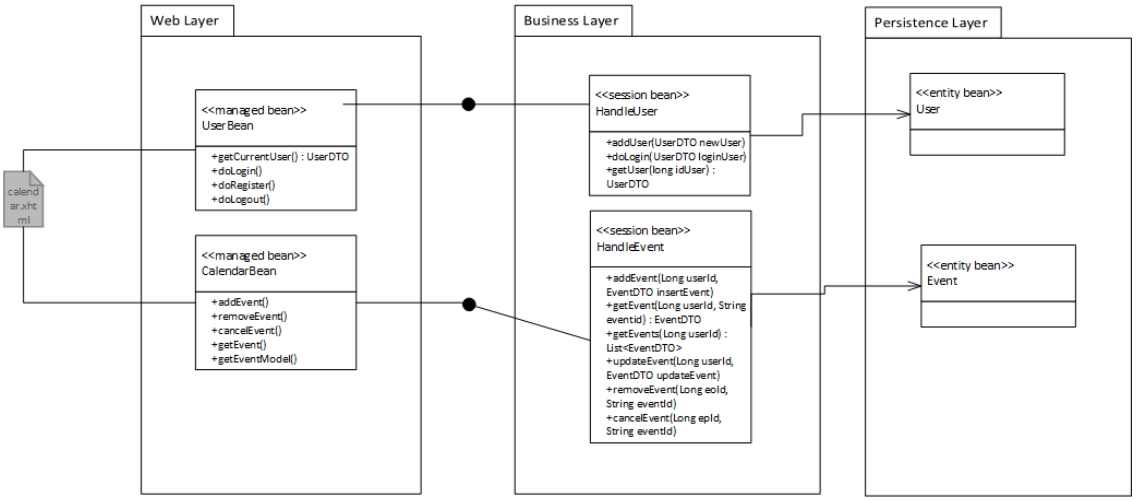

Figure 8: Event Management