**Politecnico di Milano**

**5th School of Engineering**



PhoneGuardian

Design and Implementation of Mobile Applications

# **D**esign **D**ocument

**23th May 2015**

Contents

[**D**esign **D**ocument 1](file:///D:\Net%20downloads\Android%20SDK\workspace\Documentation\Design%20Document.docx#_Toc420181238)

[1. Purpose 3](#_Toc420181239)

[2. Brief introduction 3](#_Toc420181240)

[3. Conceptual design 3](#_Toc420181241)

[4. Architectural design 3](#_Toc420181242)

[4.1. IDENTIFYING SUB-SYSTEMS 3](#_Toc420181243)

[5. PERSISTENT DATA MANAGEMENT 6](#_Toc420181248)

[5.1. CONCEPTUAL DESIGN 6](#_Toc420181249)

[5.2. LOGICAL DESIGN 8](#_Toc420181250)

[5.2.1. TRANSLATION TO LOGICAL MODEL 8](#_Toc420181251)

[6. Used tools 11](#_Toc420181252)

[7. Working Hours 11](#_Toc420181253)

# Purpose

This document aims to describe the design and prototyping steps taken for “Big Gym” web application assigned as part of a project of the “Hypermedia Applications (Web and Multimedia)” course at Politecnico di Milano.

# Brief introduction

Good description of an app

The document provides detailed web application development procedure. Starting from the conceptual design, upgrading it to logical design and, finally, page design. After page design is introduced, it is going serve as the basis for interactive mockup development and will be described at the end of this document. In order to declutter and keep the document on point, we have left out the legend i.e. the meaning of all the symbols used in schemas and decided to treat it as a given. Explanation of all the symbols used can be found in the course's lesson slides.

# Conceptual design

Entity relation schema of the database

# Architectural design

## Identifying sub-systems

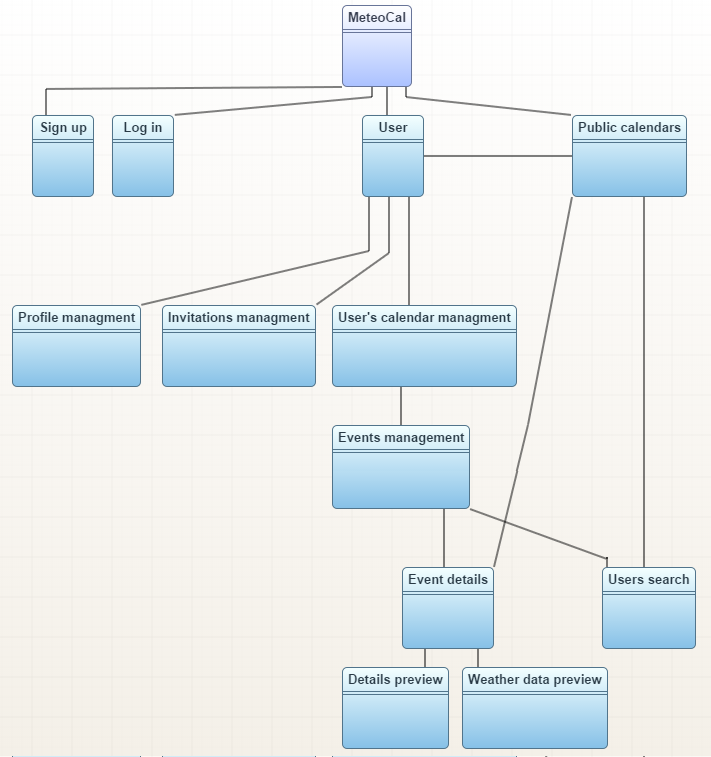
We decided to adopt a top-down approach at least at this point of the project. Maybe, once defined the sub-systems, we will adopt a bottom-up approach in order to create more reusable components.

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

We separate our systems into these sub-systems:

* Sign up subsystem;
* Log in subsystem;
* User subsystem;
  + Profile management subsystem;
  + Invitations management subsystem;
  + User’s calendar management subsystem;
* Public calendars subsystem;

Picture of architectural design scheme (Android app1, Android app2, phpServer, Database)



Explain how components communicate



# PERSISTENT DATA MANAGEMENT

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

## CONCEPTUAL DESIGN

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

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

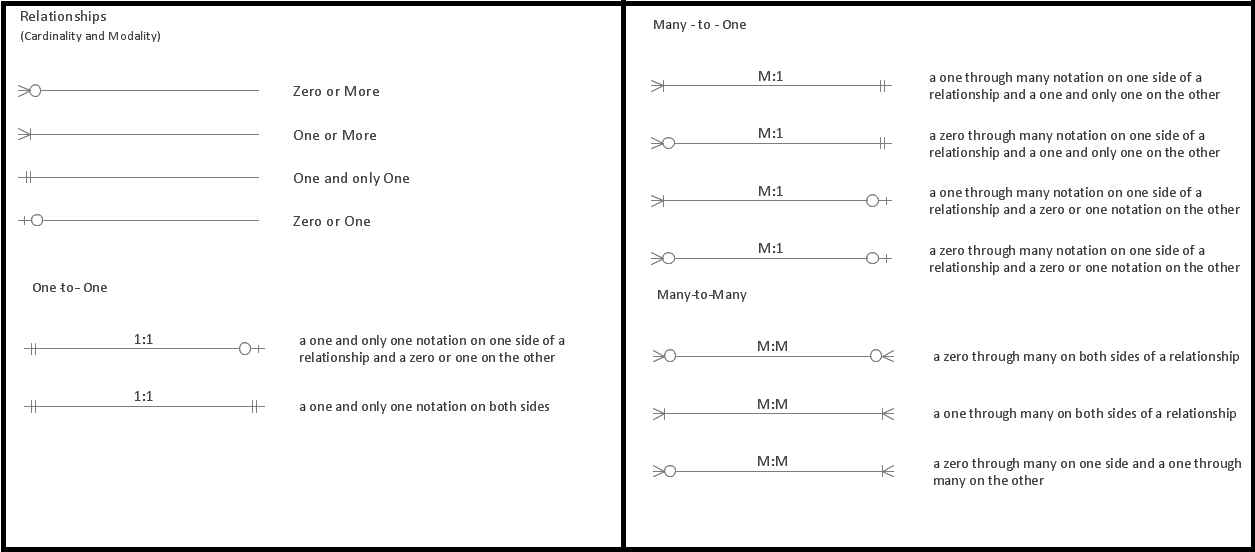
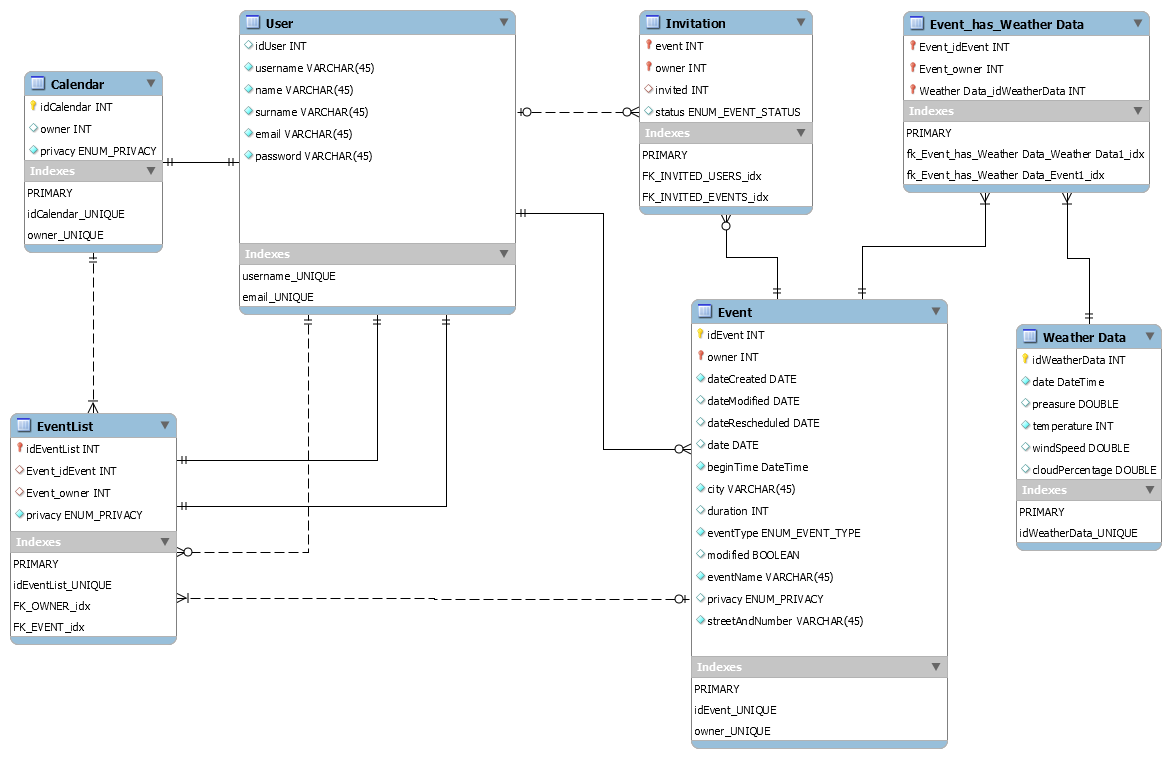


Figure: Crow’s foot notation

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

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

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

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

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

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

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

## LOGICAL DESIGN

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

### TRANSLATION TO LOGICAL MODEL

1. User table: Relation ‘Invited’ is a n:m relation so it is modeled trough the Invitation table, relation between User and Invitation table is 1:n relation that maps idUser(User:PK)->invited(Invitatio:FK). Relation ‘Owns’ is a 1:n relation that maps idUser(User:PK)->owner(Event:FK). Relation ‘OwnedEvents’ is a 1:1 relation that maps idUser(User:PK)->idEventList(EventList:PK). Relatiom ‘OtherEvents’ is a 1:1 relation that maps idUser(User:PK)->idEventList(EventList:PK). 1:n relation between User and EventList table is a part of a n:m relation which is modeled through EventList table, it maps idUser(User:PK)->Event\_Owner(EventList:FK). Relation ‘CalendarOwner’ is a 1:1 relation that maps idUser(User:PK)->idCalendar(Calendar:PK).
2. Calendar table: Relation ‘AppearsIn’ is a 1:n relation that maps idCalendar(Calendar:PK) + owner(Calendar:FK) -> Event\_idEvent(EventList:FK) + Event\_owner(EventList:FK) which is used to map events owned by multiple users that are a part of a public calendars and are also themselves marked as public.
3. Event table: Relation 1:n between Event and Invitation is a part of n:m relation that connects users and events through invitations. It maps (event+owner)(Invitation:CK)->idEvent(Event:PK) + owner(Event:FK). Relation 1:n between Event and Event list is a part of n:m relation that maps users and events they are participants of. It maps idEvent(Event:PK) + owner(Event:FK) -> Event\_Owner(EventList:FK) + idEventList(EventList:PK). Relation between Event and Event\_Has\_WeatherData is a part of n:m relation between Event and WeatherData, it maps idEvent(Event:PK) + owner(Event:FK) -> (Event\_idEvent+Event\_owner)(Event\_Has\_WeatherData:CK).

The final model has the following physical structure:

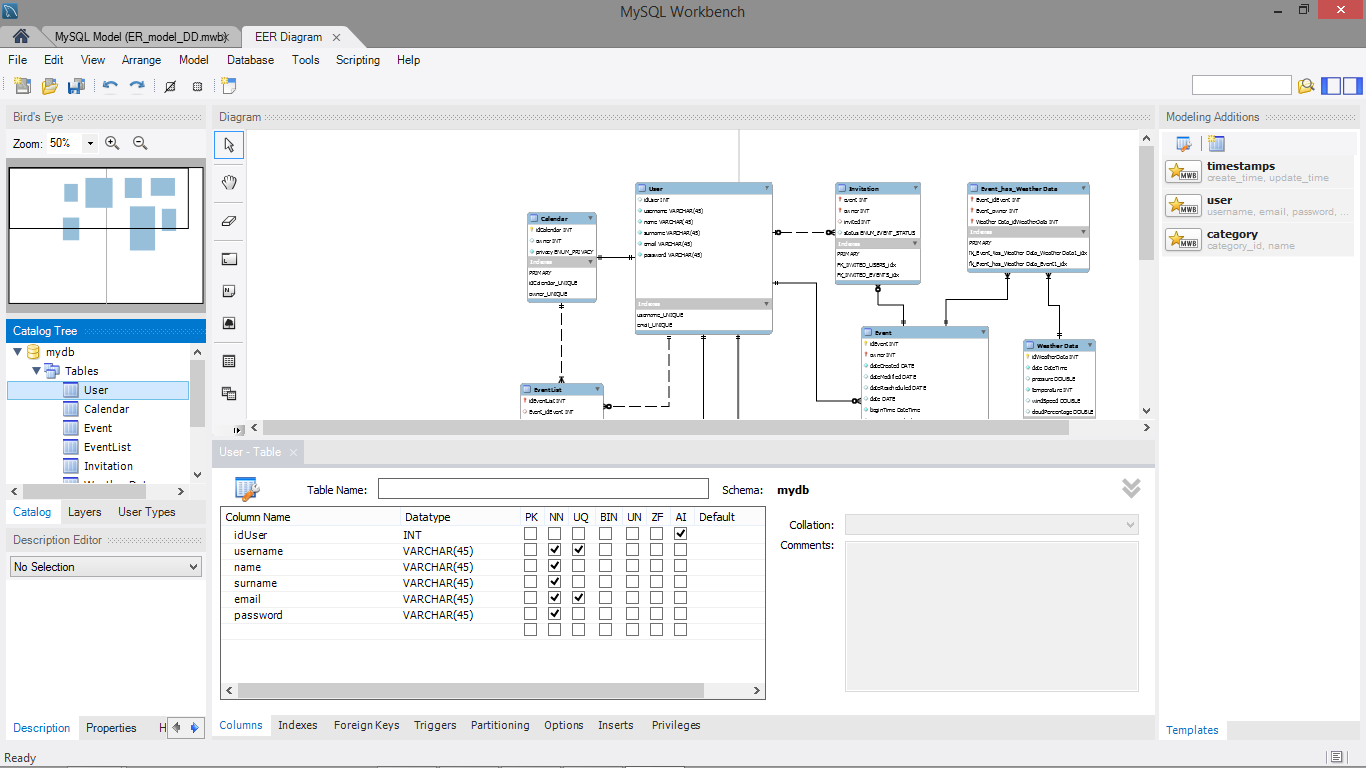


Figure 1: User Table

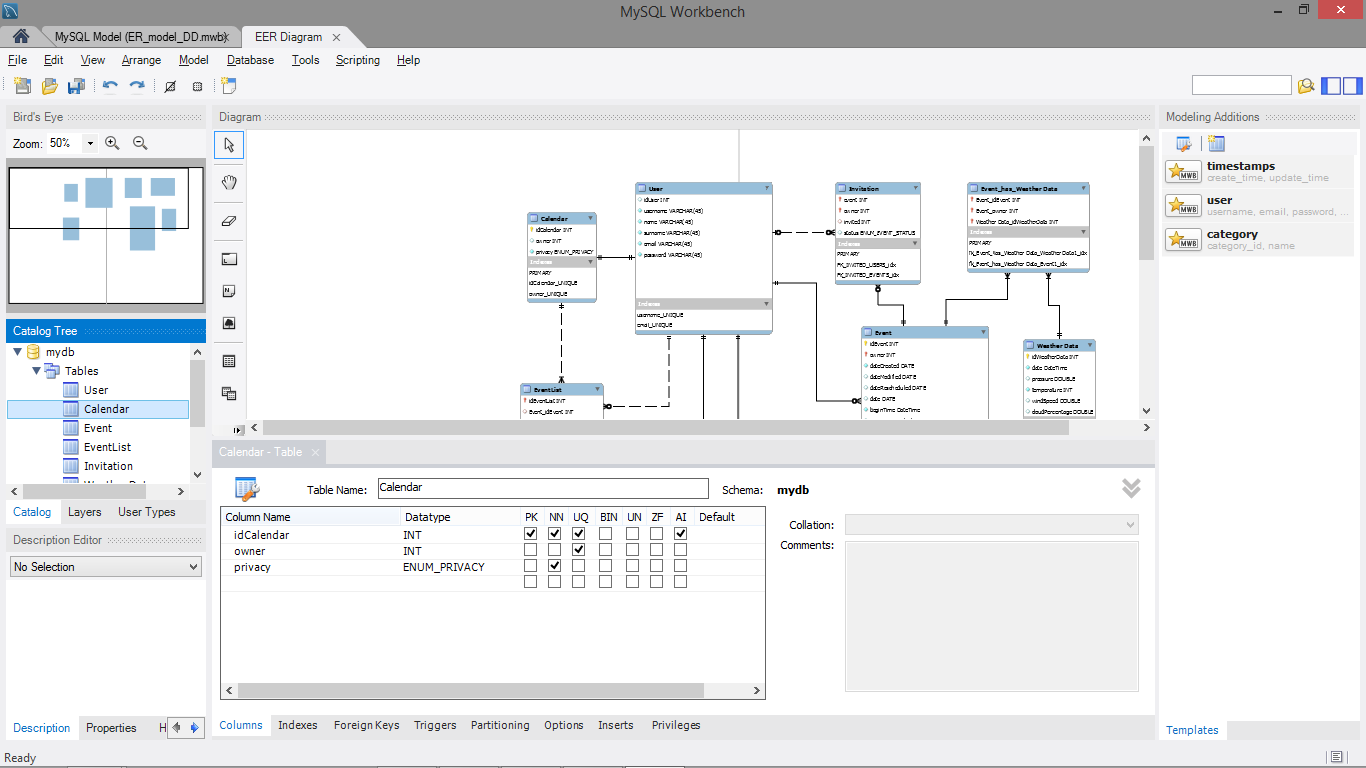


Figure 2: Calendars Table

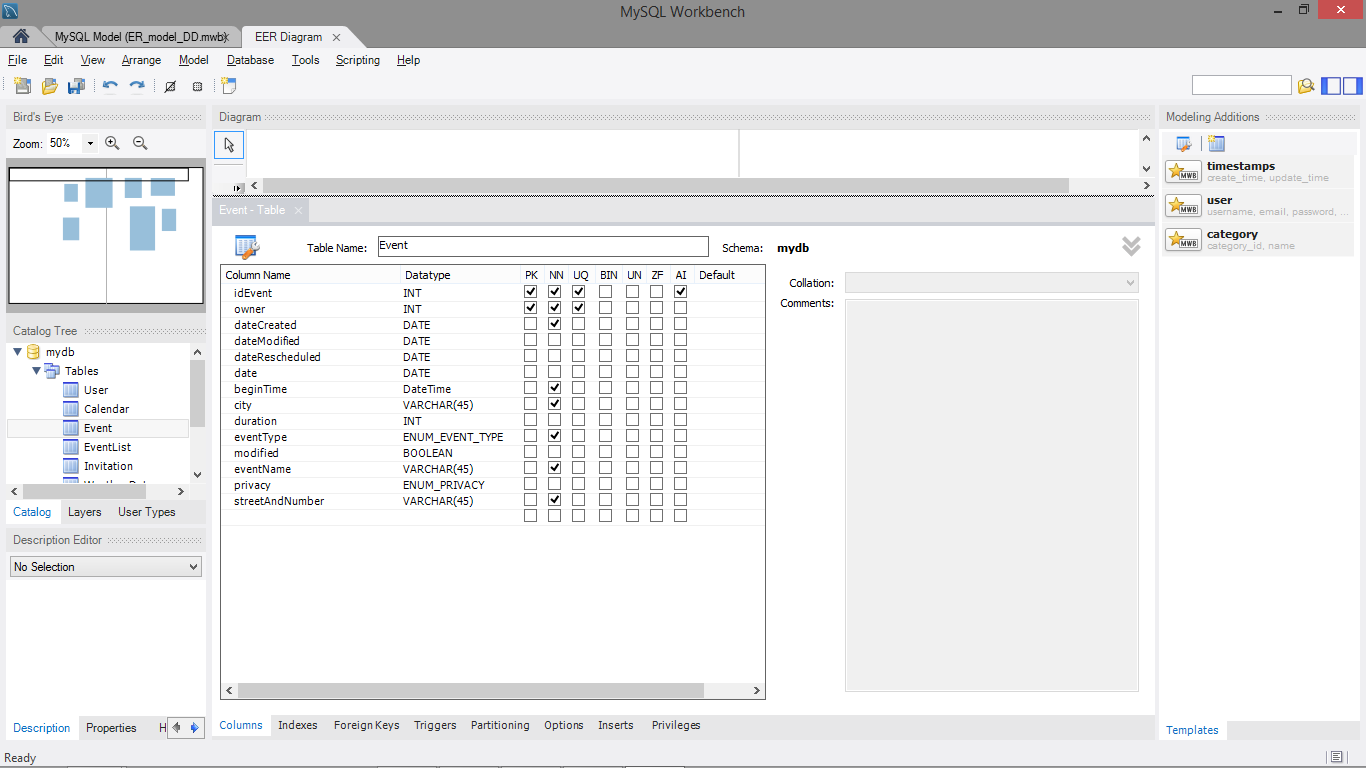


Figure 3: Event Table

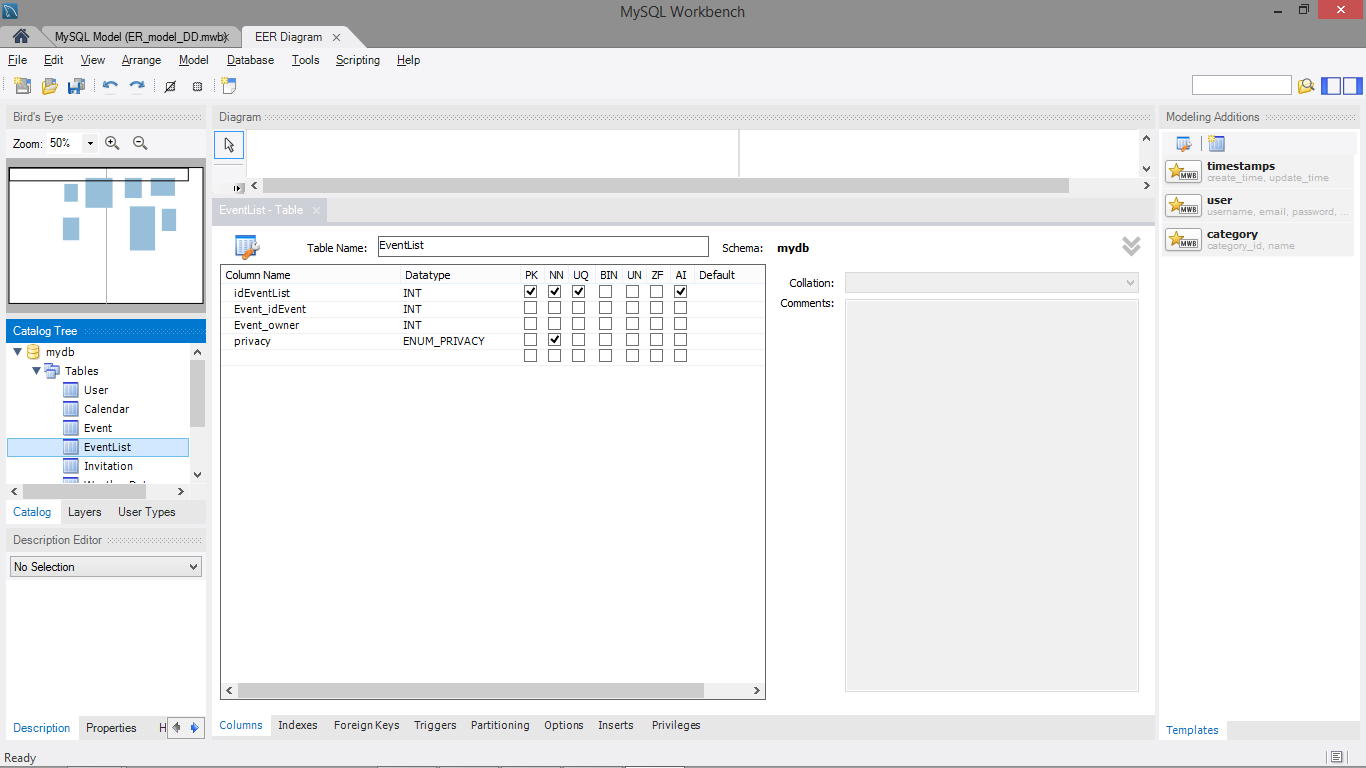


Figure 4: EventList Table

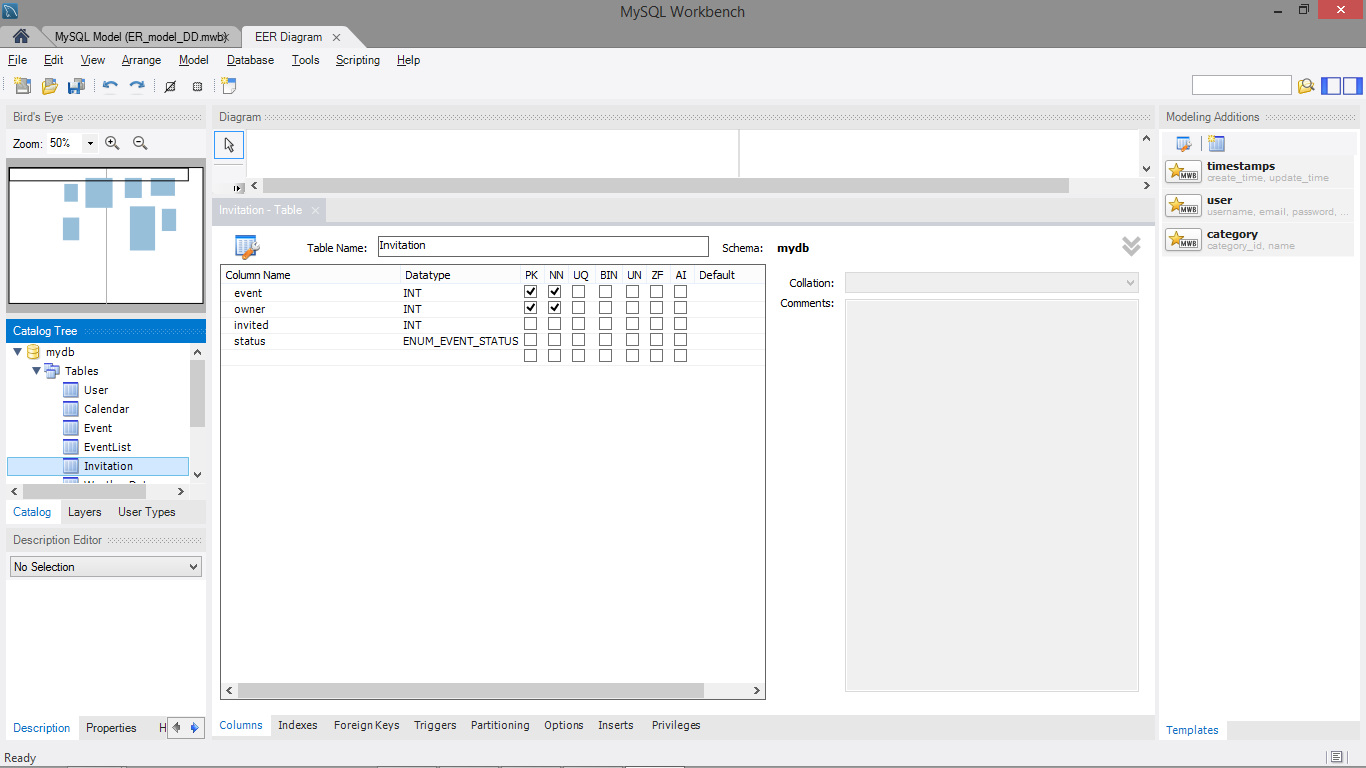


Figure 5: Invitation Table

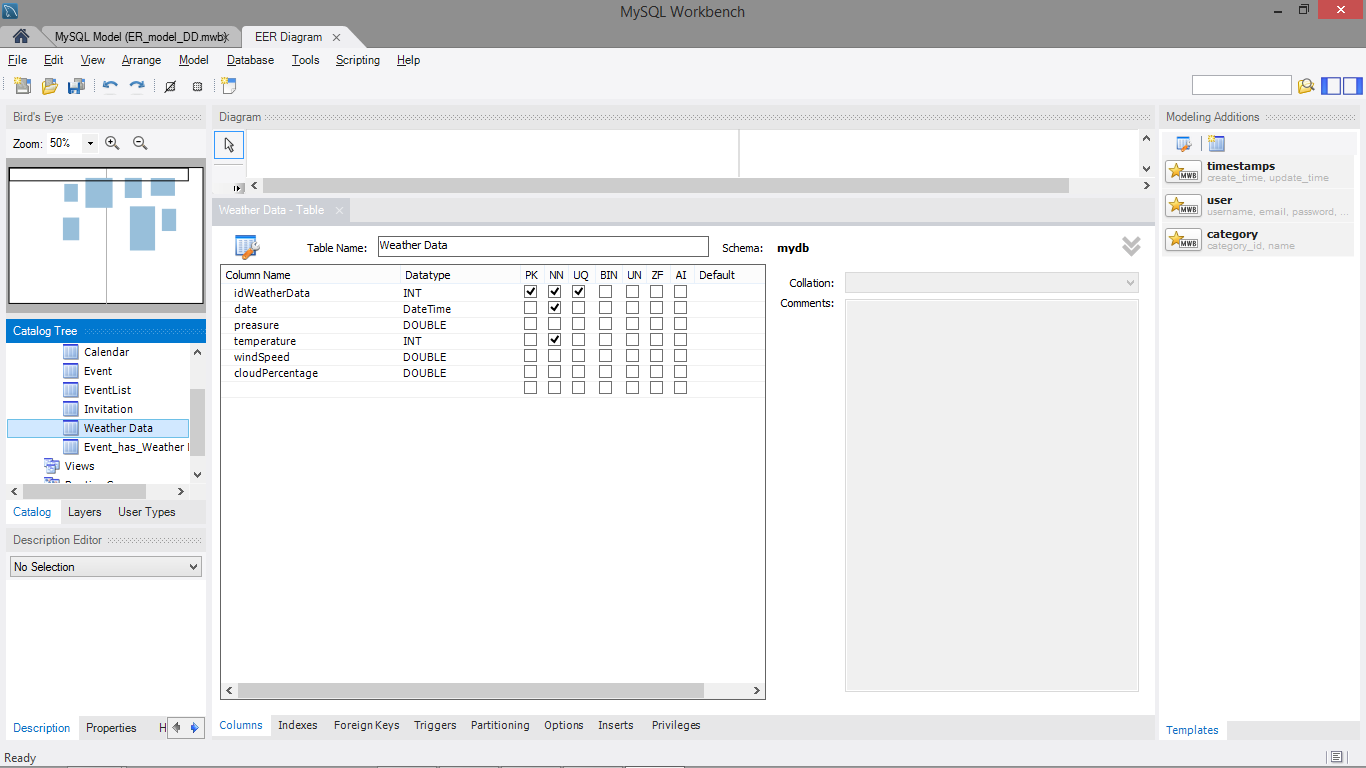


Figure 6: WeatherData Table

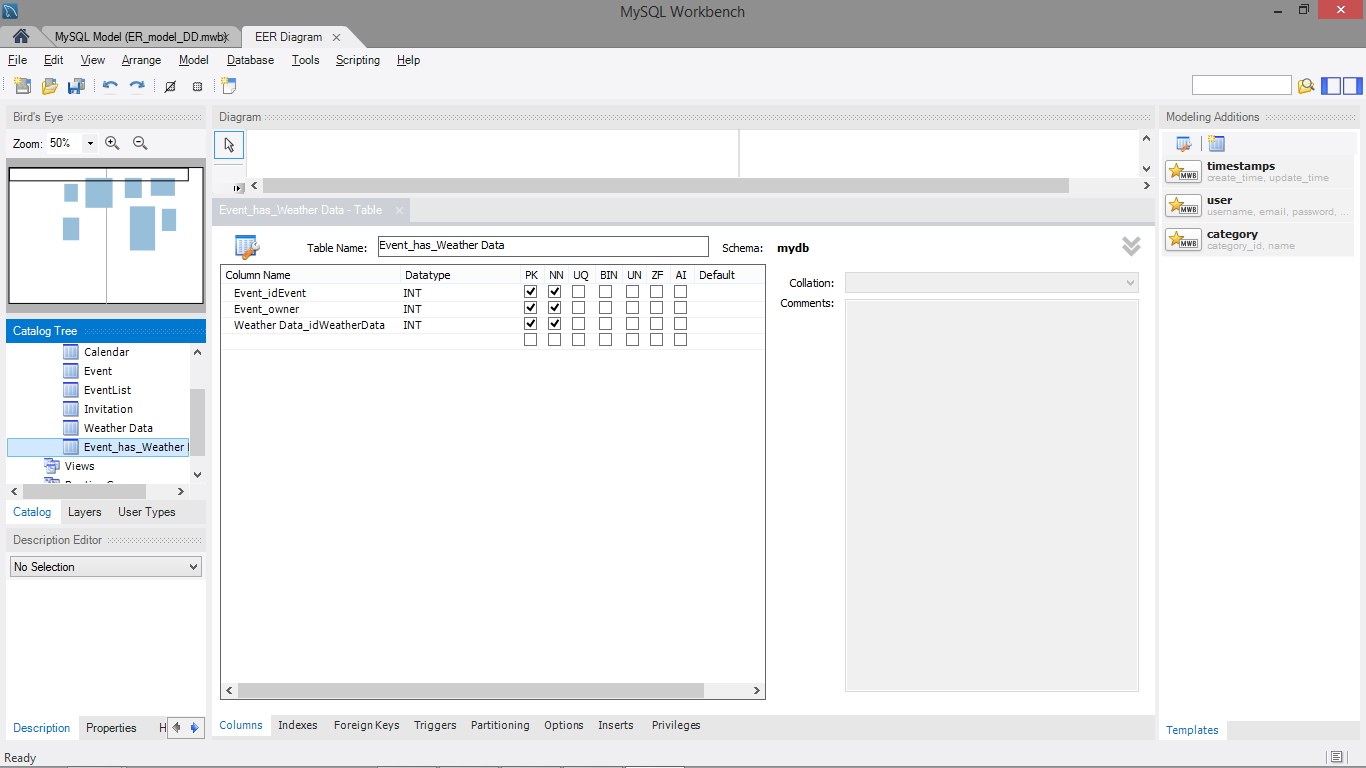


Figure 7: Event\_Has\_WeatherData Table

# Citizen app

class diagram

description

# Authority app

class diagram

description

# PHP Server

description

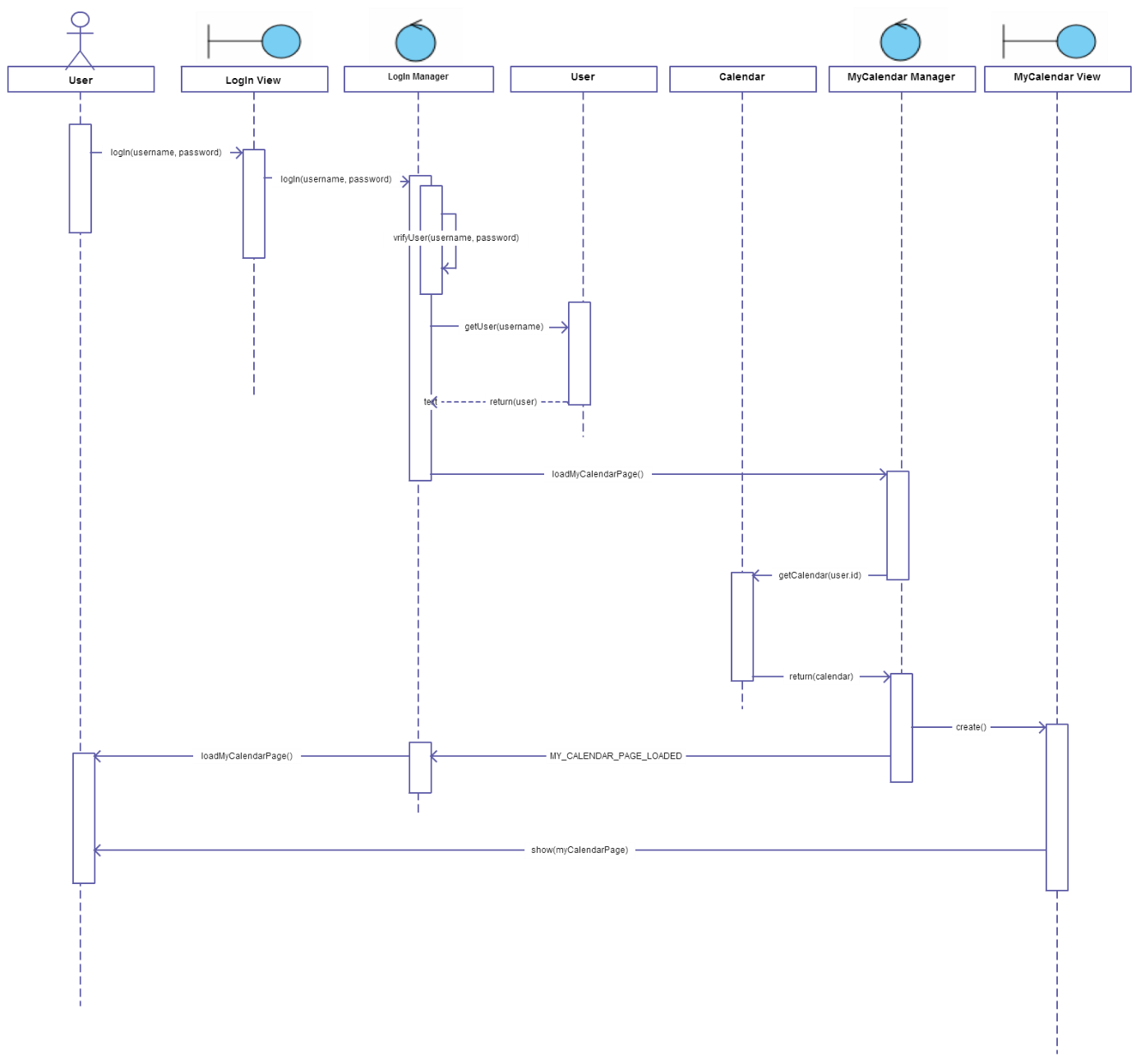
# SEQUENCE DIAGRAMS

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

## Log In

A user:

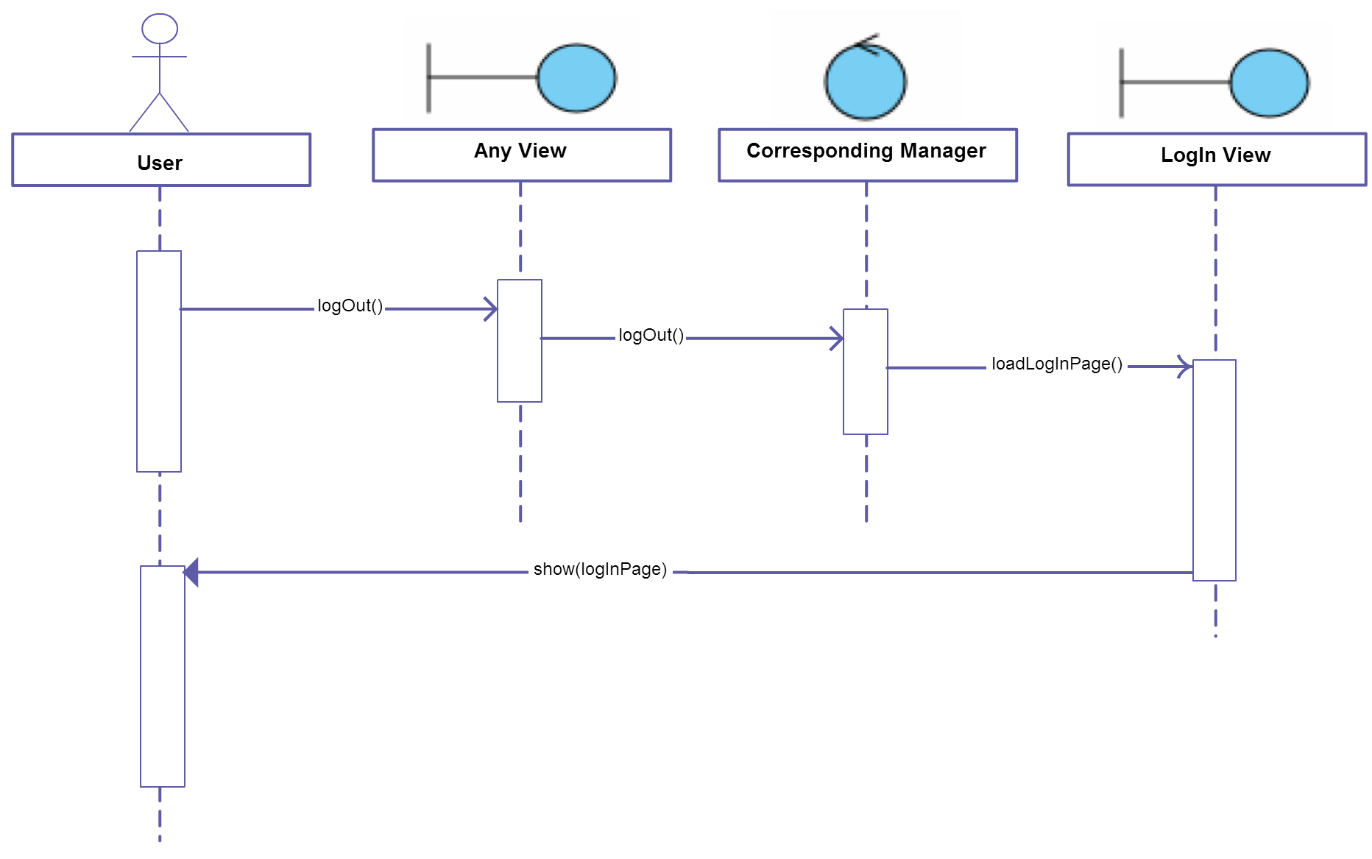
* Logs in.



## User Logs Out

A user:

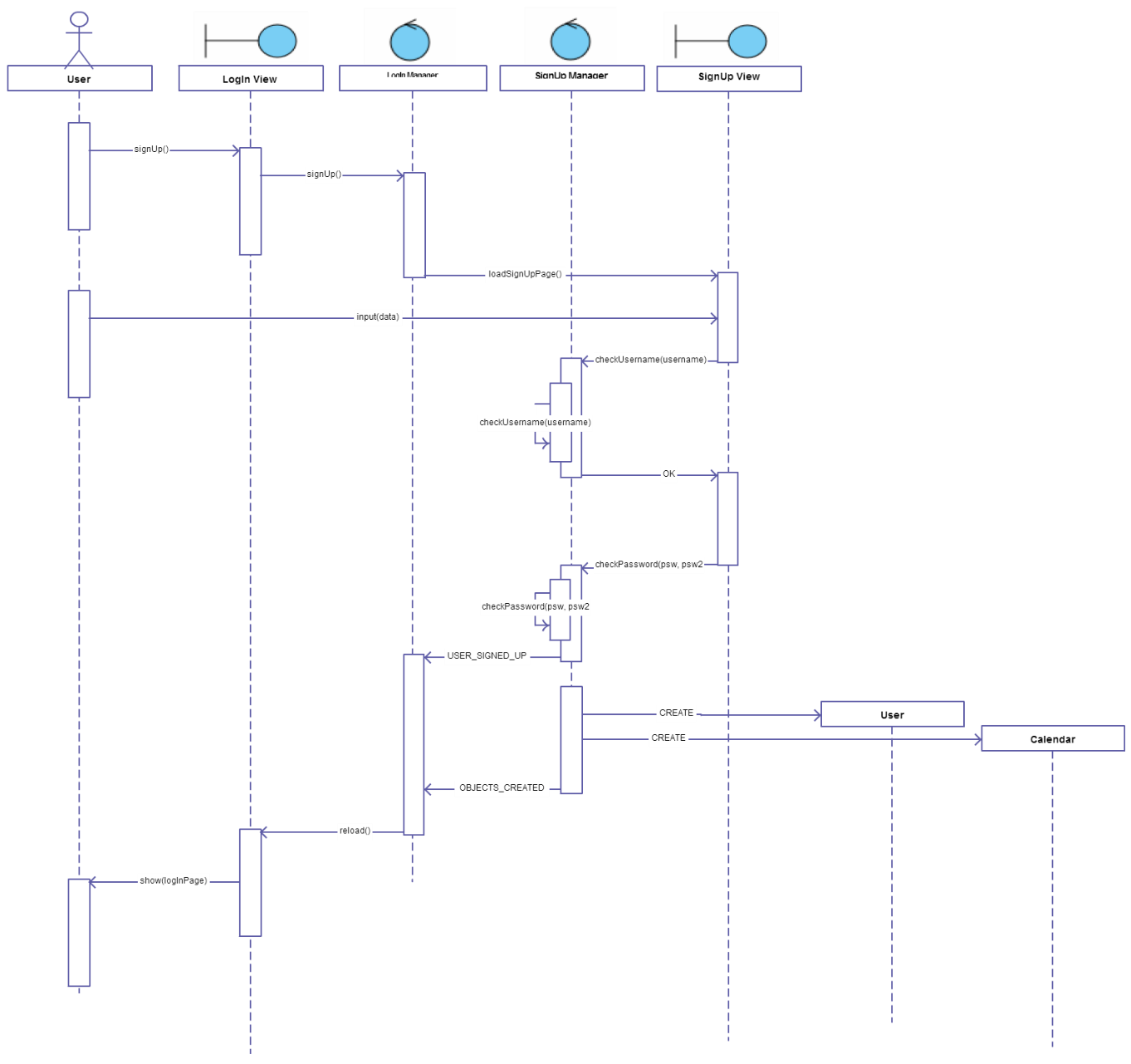
* Logs out of the system



## Sign Up

A user:

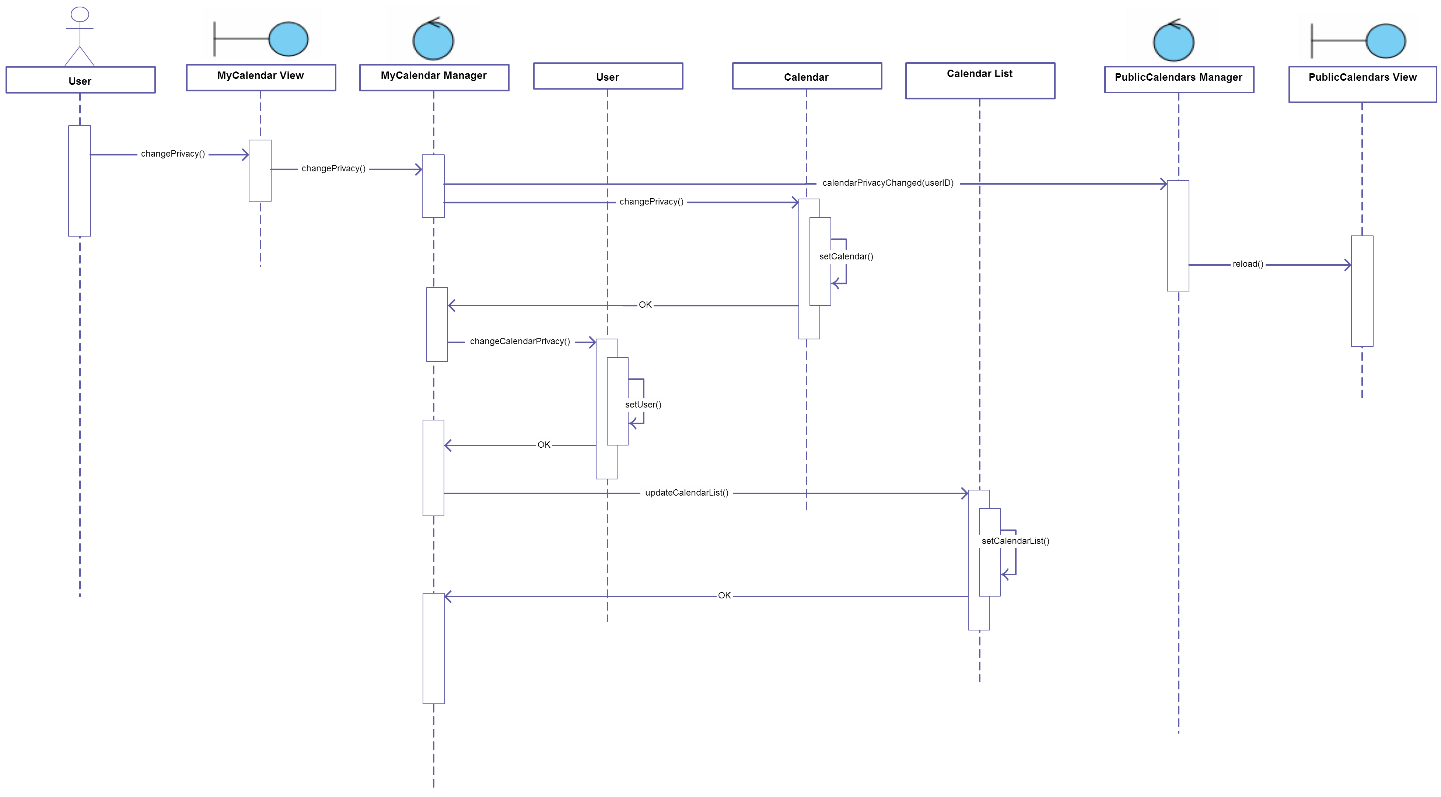
* Signs up as a new user of the system



## Change calendars privacy

A user:

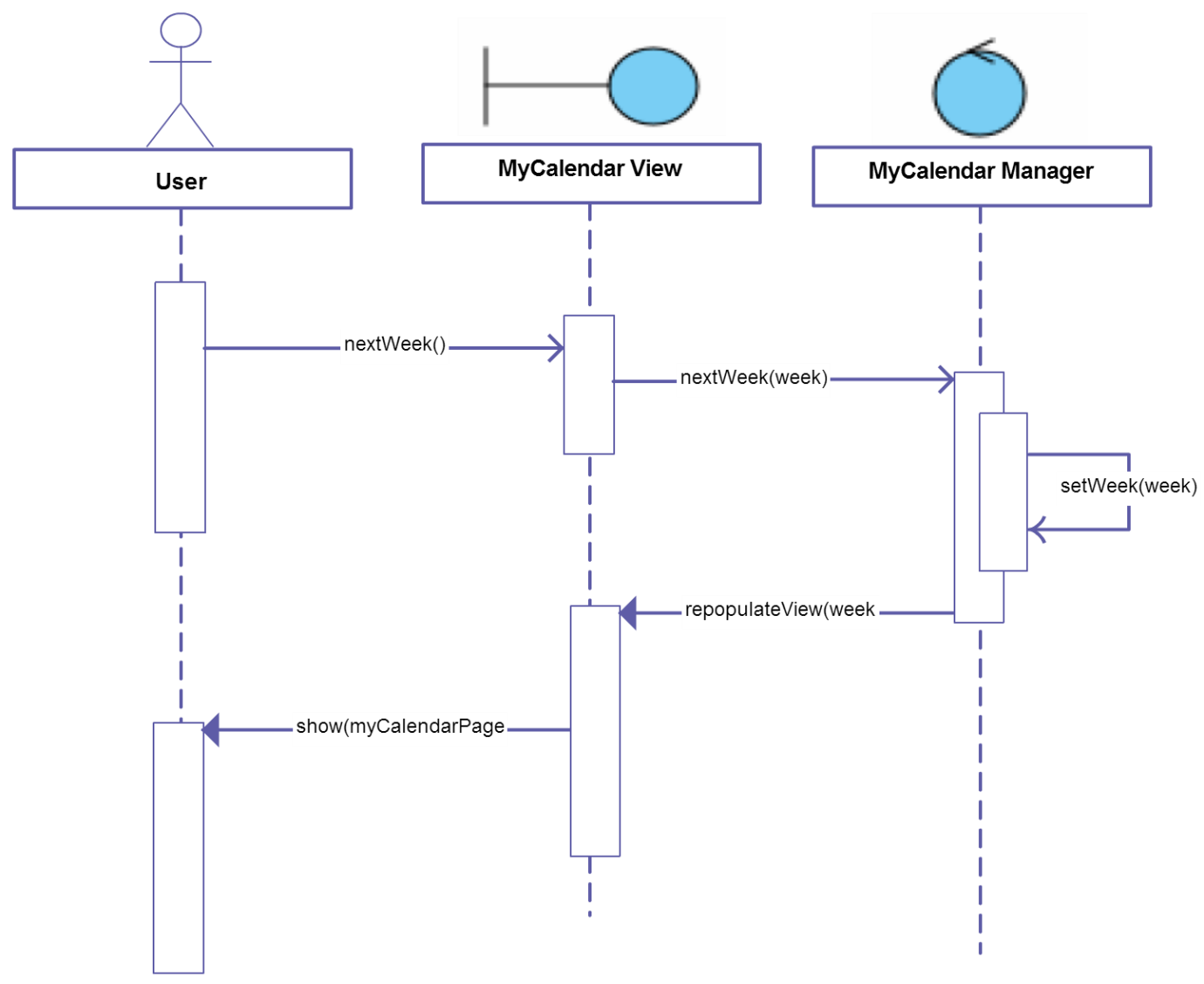
* Changes privacy setting of his calendar



## Next week

A user:

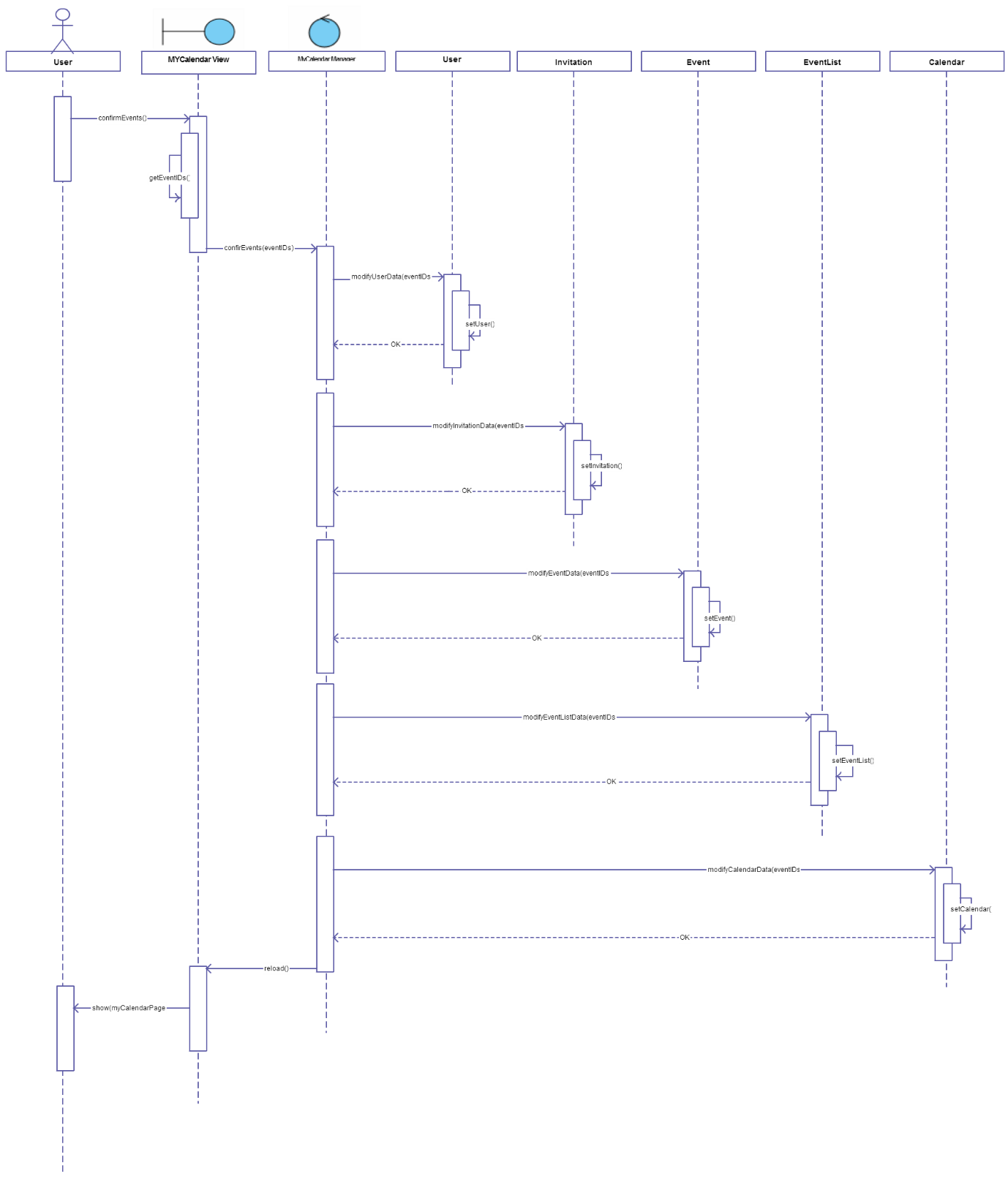
* User navigates through the calendar he is currently seeing



## Accept Invitations

A user:

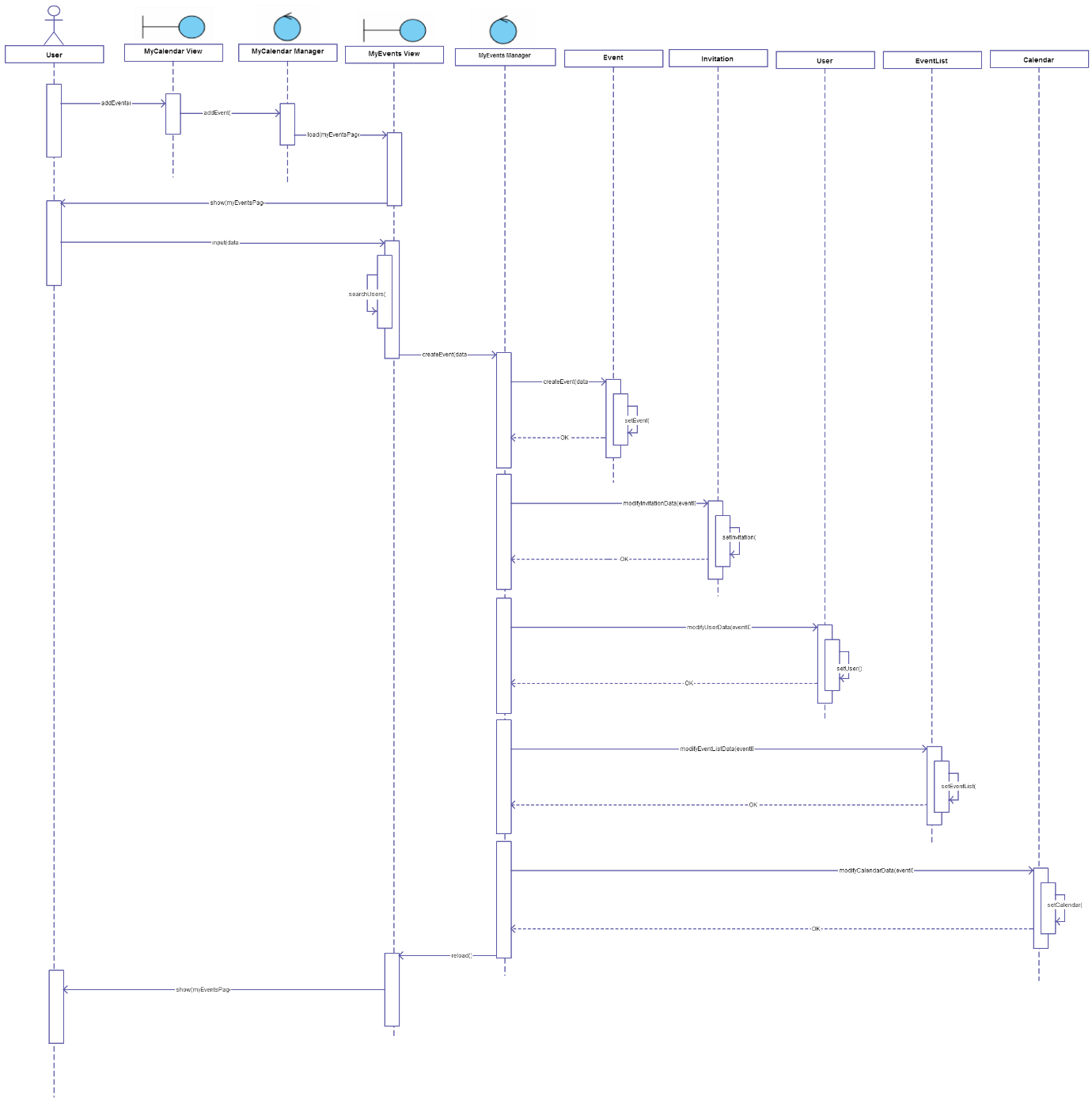
* User accepts invitations to the events he has selected



## Add Event

A user:

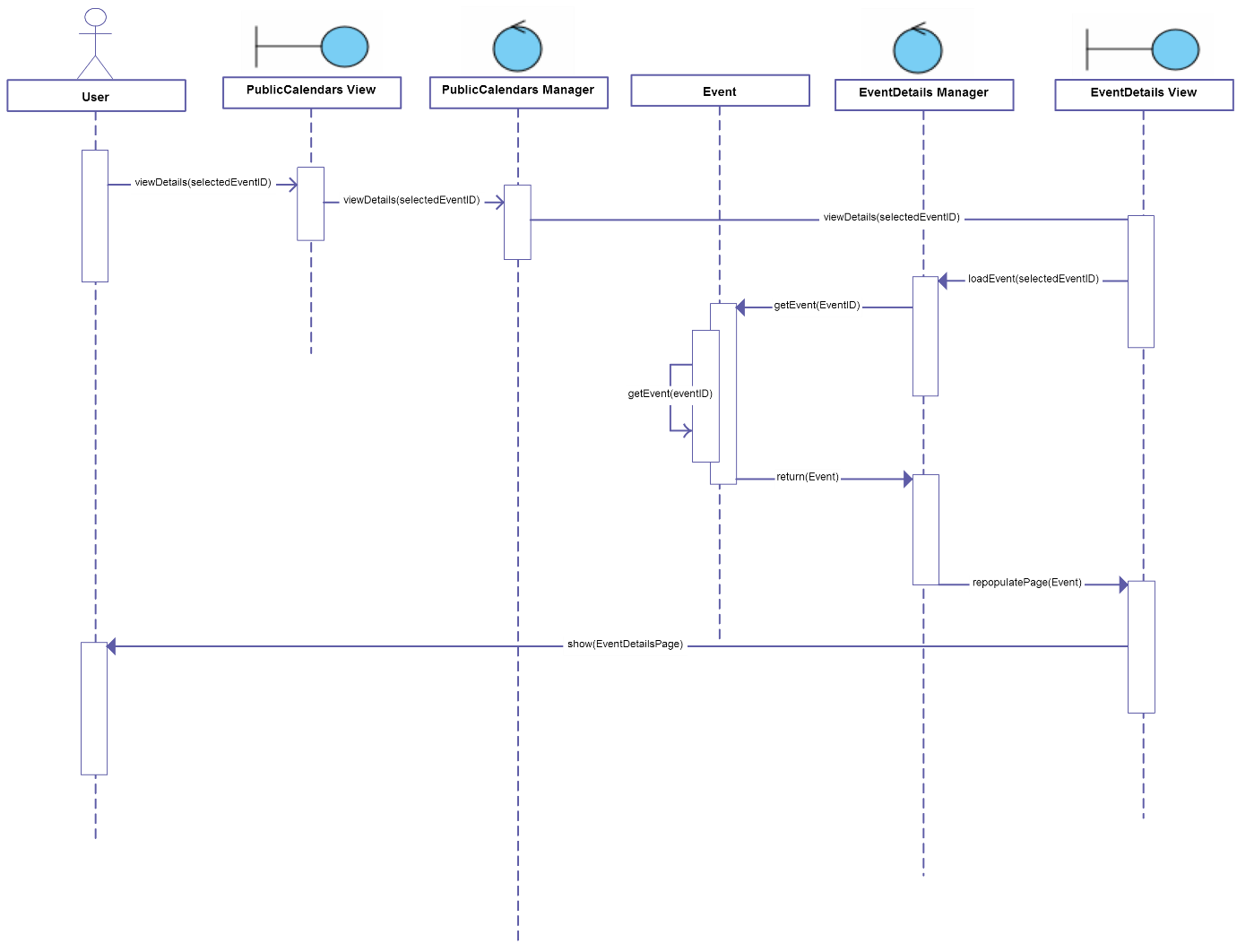
* User creates a new event



## View Events Details

A user:

* User previews details of the event he has selected



# FINAL CONSIDERATIONS

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

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

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

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

# Used tools

* Microsoft Office Word
* Adobe Illustrator
* Evolus Pencil

# Working Hours

|  |  |
| --- | --- |
| **Name** | **Working hours** |
| Nemanja Stolic | 20 hours |
| Mirjam Skarica | 20 hours |
| Milica Jovanovic | 20 hours |