**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 “PhoneGuardian” software system assigned as a project of the “Design and Implementation of Mobile Applications” course at Politecnico di Milano.

# Brief introduction

PhoneGuardian is a software system that consists of three parts: CitizenApp, AuthorityApp and a Web server between them. Main purpose of the system is giving real-time information about events that happened. This information can be seen either from a user of CitizenApp (regular user) or by an user of AuthorityApp (authority staff i.e. policeman, fireman or medical staff on the field).

CitizenApp is an Android application used for reporting emergency events or searching among reported events by applying a filter. Main purpose of the application is to offer insight to dangerous zones of cities. User can also follow reported event and current development of the situation through comments posted by other users.

AuthorityApp is an Android application used for receiving reports about events that happened i.e. the application works in real-time. The user of the application (usually authority staff) can accept a job they want to deal with and after checking situation they can write a review about it and notify other users.

WebServer is a medium that connects CitizenApp and AuthorityApp. The server deals with data transfer and connects the applications to 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 easier 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:

* Citizen app (Android application);
* Authority app (Android application);
* Web server (PHP server);
* Database (MySQL);

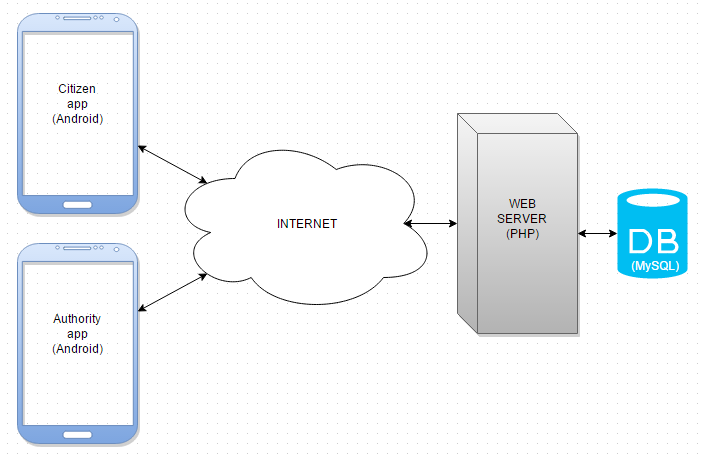


Figure - System architecture

All the data and control logic are centralized on our Web server. Each interaction coming from citizens or authority staff must be followed with server’s acknowledgment. Citizens and Authorities cannot communicate directly (although the server may deliver user’s phone number to authority staff, more info about it will be described later on). In that way we gain more control over our system.



# 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 entities in our system are *User* and *Authority*. Regular user, after completing the procedure of signing up becomes the user of a system. Each *User* can create zero, one or many entities of type *Event*. Thus each *Event* is connected to one *User*, and one *User* can be “the creator” of zero, one or many *Event*s. The relation is identified by the field *phone\_number* which is present in the *User* table. *One-to-many* relation (and also other relations) are presented later in the diagram using Crow’s foot notation.

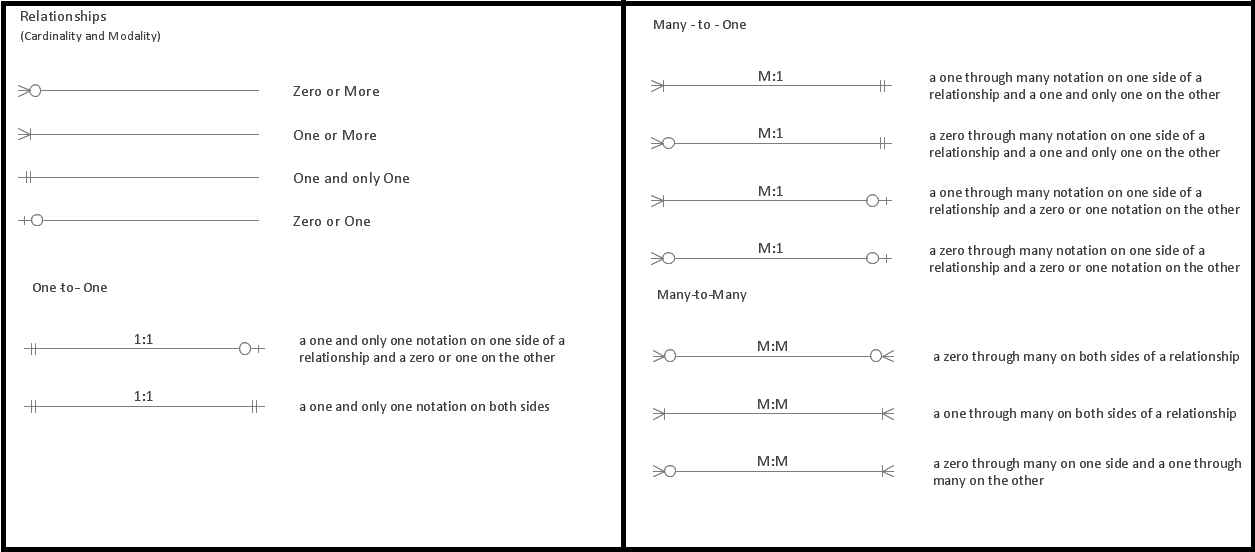


Figure: Crow’s foot notation

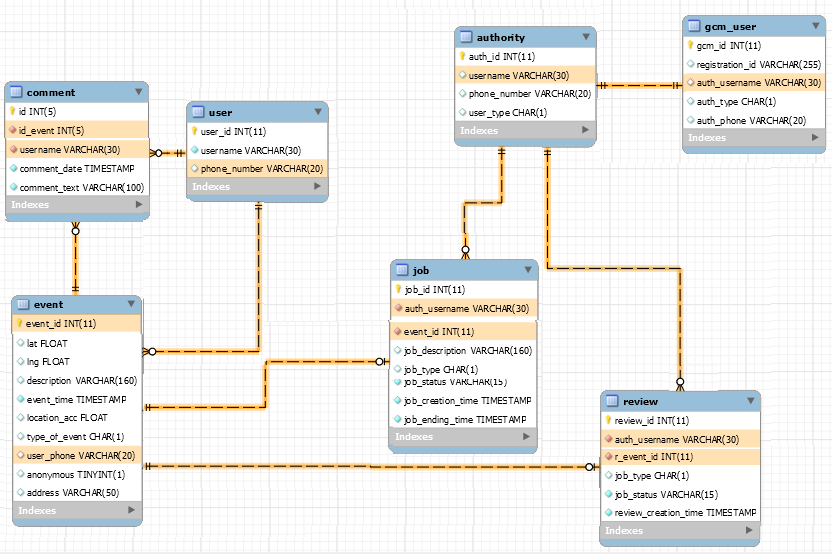
In the following diagram our system’s conceptual design is presented: 

Figure - Database design

Each *User* may post zero or more *Comment*s related to one specific entity Event. Each *Event* may contain zero or many *Comment*s posted by different users, but each Comment has to be connected to its unique creator of type *User*. Hence, the relations are *One-to-many* between *User* and *Comment* and *One-to-many* between *Event* and *Comment*.

On the other side of the schema there is a simple entity called *Authority*. Each *Authority* must also be *Gcm\_user* (Google Cloud Messaging User), and *Gcm\_user* must be identified by a single *Authority* username. Hence, *One-to-one* relation is added between them.

Each *Authority* may take an Event for investigation, and by doing that (s)he creates an entity of type *Job* in the database. After taking the *Job*, *Authority* must review the job, creating an entity *Review* in the database. Hence, *One-to-many* relation between *Authority* and *Job*, and *One-to-many* relation between *Authority* and *Review*.

*Job* and *Review* entities, from the creation time, must be connected to a single *Event*. Hence, *One-to-one* relation is added to the database.

Note: Although an *Authority* may take many *Jobs* and make many *Review*s by the database design, in software implementation we limited the amount of *Job*s to be taken to one. The *Job* needs to be reviewed in order to allow the *Authority* to take another *Job*. In this way we prohibited Authorities from greedy actions, for instance – taking all the jobs for (her)himself.

## 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.

After conceptual design we needed to create a real structure of the database, so we used a forward engineering approach to generate tables from Entity-Relations diagram. In order to achieve that we used a tool *MySqlWorkbench* which supports “Forward Engineering” transformation. In this way the time necessary to create the database was reduced effectively.

# Citizen app

class diagram and its description

## Activities

|  |  |
| --- | --- |
| D:\FAX\I year\Mobile\PhoneGuardian\Documentation\Screenshots\CitizenApp\1 register_activity.png  Figure 1. Register Activity | D:\FAX\I year\Mobile\PhoneGuardian\Documentation\Screenshots\CitizenApp\2 alert_acitivty.png  Figure 2. Main Activity |
|  |  |

Figure 1. presents user register procedure. The user needs to type in username and his phone number. The application picks up his Country code based on his location. After submitting the information, the application tries to verify the phone number. In case of invalid phone number, the user receives a toast message stating that the phone number entered is incorrect.

Figure 2. presents main activity. If the user already has an account for this application, the register activity will not start; the application will check his credentials and immediately start the main activity. From this activity user can either report an emergency event or check the map for events nearby.

|  |  |
| --- | --- |
| C:\Users\milica13629\AppData\Local\Microsoft\Windows\INetCache\Content.Word\3 add_location_activity.png  Figure 3. Add Event Activity | C:\Users\milica13629\AppData\Local\Microsoft\Windows\INetCache\Content.Word\4 pgmap_activity.png  Figure 4. Map Activity |
| C:\Users\milica13629\AppData\Local\Microsoft\Windows\INetCache\Content.Word\5 marker_activity.png  Figure 5. Event Details Activity | |

Figure 3. presents an activity for adding new events. The application suggest to the user to add and event to his current location or to type in an address of the event; application is doing address matching using Google Places API and suggests to the user the best match; also the application is updating the time and date to current time and date. User needs to choose a type of event and type in a description of the emergency situation that he saw. Also he has an option of adding an event as an anonymous user or he could reveal his phone number to the appropriate authorities.

Figure 4. presents a map with pins fitting the screen around the current location of the user. Zooming in and out the viewport reduces or increases the number of pins shown on the map. The user can filter the pins by clicking the icons representing different types of reported events or he can refresh the map clicking on the appropriate button. Tapping on desired pin the user can get insight to more detail information about the reported event (shown on Figure 5.); also user is able to make more complex queries by clicking on filter button which takes him to the Filter Activity (shown on Figure 8.).

|  |  |
| --- | --- |
| C:\Users\milica13629\AppData\Local\Microsoft\Windows\INetCache\Content.Word\6 add_comment_activity.png  Figure 6. Add Comment Activity | C:\Users\milica13629\AppData\Local\Microsoft\Windows\INetCache\Content.Word\7 view_comments_activity.png  Figure 7. View Comments Activity |
| C:\Users\milica13629\AppData\Local\Microsoft\Windows\INetCache\Content.Word\8 search_activity.png  Figure 8. Filter Activity | C:\Users\milica13629\AppData\Local\Microsoft\Windows\INetCache\Content.Word\9 list_of_markers_activity.png  Figure 9. List of events Activity – after applying filter |

Figure 6. presents an activity for posting comments on the event which was selected by tapping a pin on the map. Comments are saved in a database and are connected with the selected event. Any user can check comments posted for any reported event. The list of comments and authority review together with the date of posting and usernames are shown in figure 7.

Figure 8. presents the activity for creating more complex queries for filtering the events. User can filer events based on his current address or he could pick an address, also he can set a radius around the typed address he is interested in. Filter includes choosing a type of event, description of reported event and date span he is interested in. The results of applied filter can be shown on a map or in a list.

Figure 9. presents list of events based on the applied filter created in previous acitivty. In the list user can see basic information about events.

# Authority app

class diagram

description

## Activities

|  |  |
| --- | --- |
| C:\Users\milica13629\AppData\Local\Microsoft\Windows\INetCache\Content.Word\1 register_activity.png  Figure 10. Register Activity | C:\Users\milica13629\AppData\Local\Microsoft\Windows\INetCache\Content.Word\2 main_activity.png  Figure 11. Main Activity |

|  |  |
| --- | --- |
| C:\Users\milica13629\AppData\Local\Microsoft\Windows\INetCache\Content.Word\3 take_job_activity.png  Figure 12. Job List Activity | C:\Users\milica13629\AppData\Local\Microsoft\Windows\INetCache\Content.Word\4 job_details.png  Figure 13. Job Details Activity |

|  |  |
| --- | --- |
| C:\Users\milica13629\AppData\Local\Microsoft\Windows\INetCache\Content.Word\5 map_acitivty.png  Figure 14. Map Activity | C:\Users\milica13629\AppData\Local\Microsoft\Windows\INetCache\Content.Word\6 review_job.png  Figure 15. Review Job Activity |
| C:\Users\milica13629\AppData\Local\Microsoft\Windows\INetCache\Content.Word\7 CitizenApp add_location_activity.png  Figure 16. Add Event Activity – CitizenApp | C:\Users\milica13629\AppData\Local\Microsoft\Windows\INetCache\Content.Word\8 Notification_service.png  Figure 17. Notification Service |

# PHP Server

description

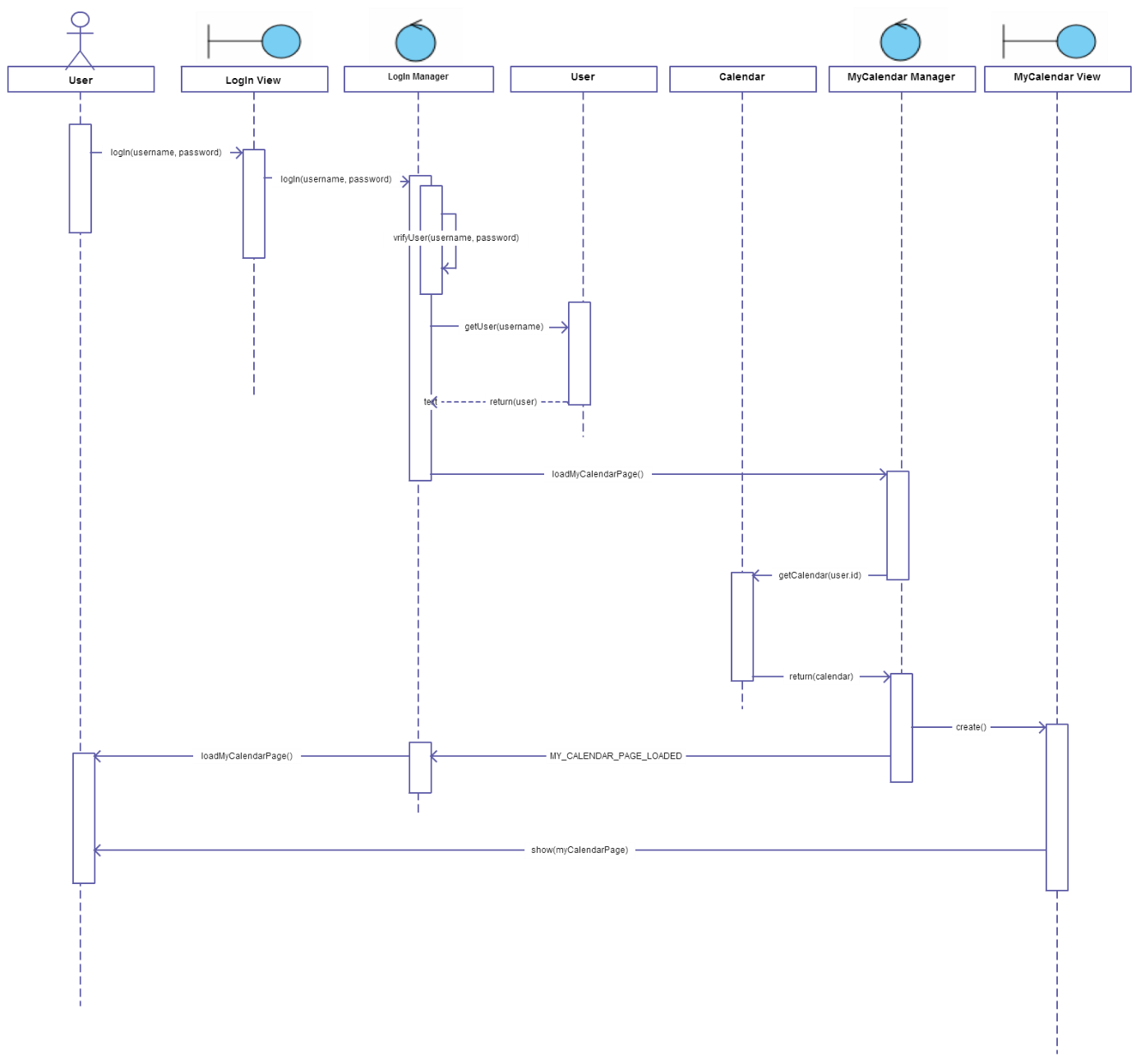
# SEQUENCE DIAGRAMS

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:

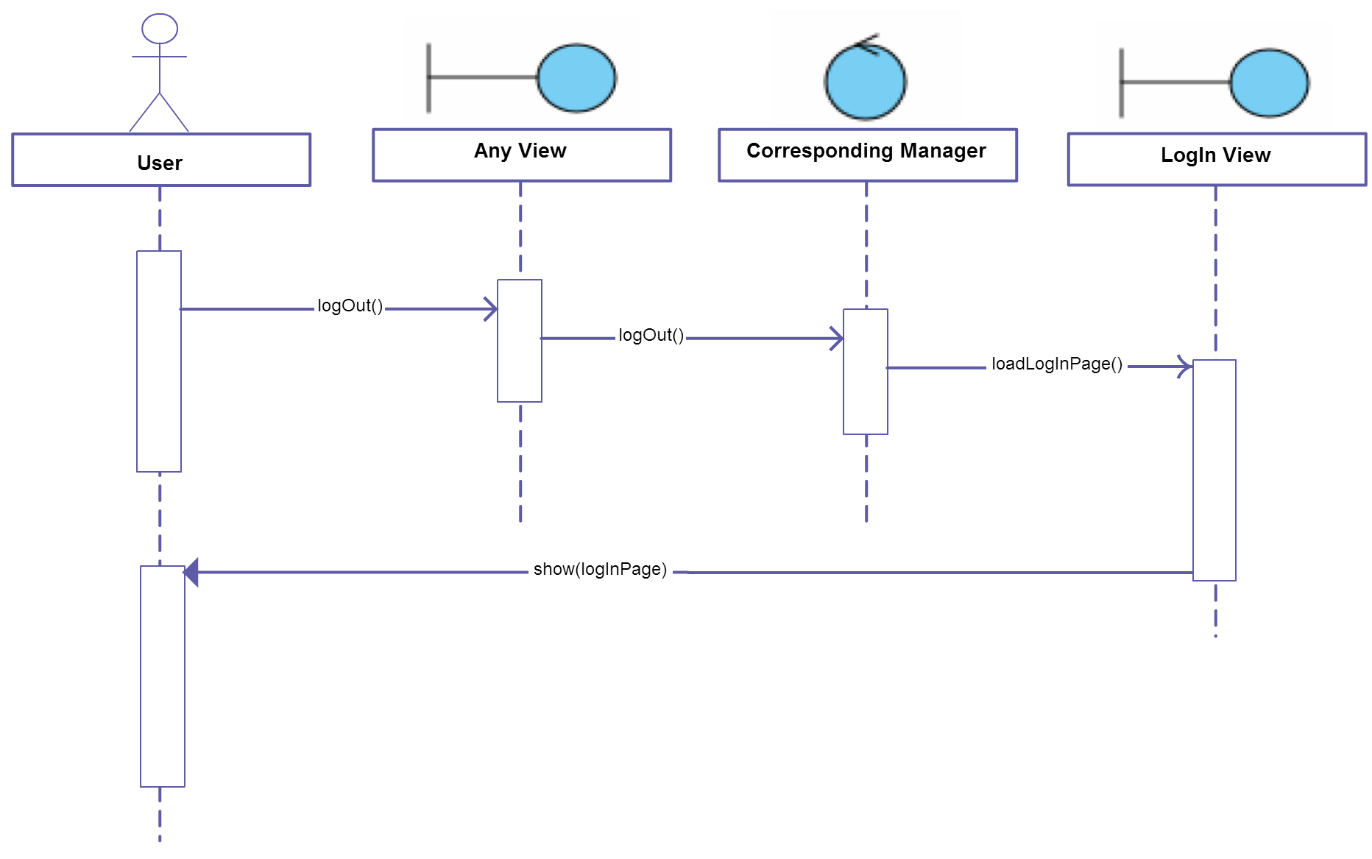
* Logs in.



## User Logs Out

A user:

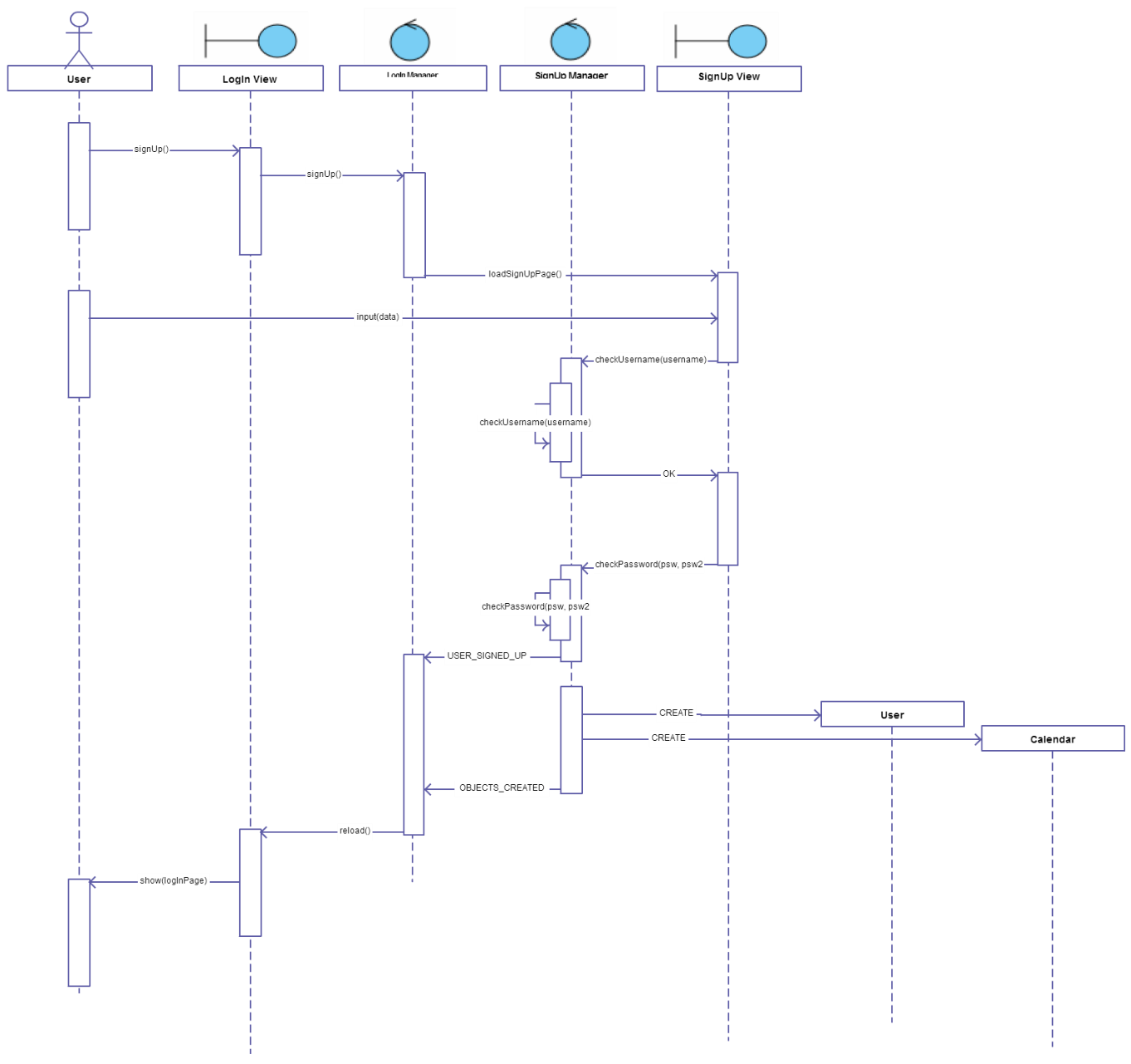
* Logs out of the system



## Sign Up

A user:

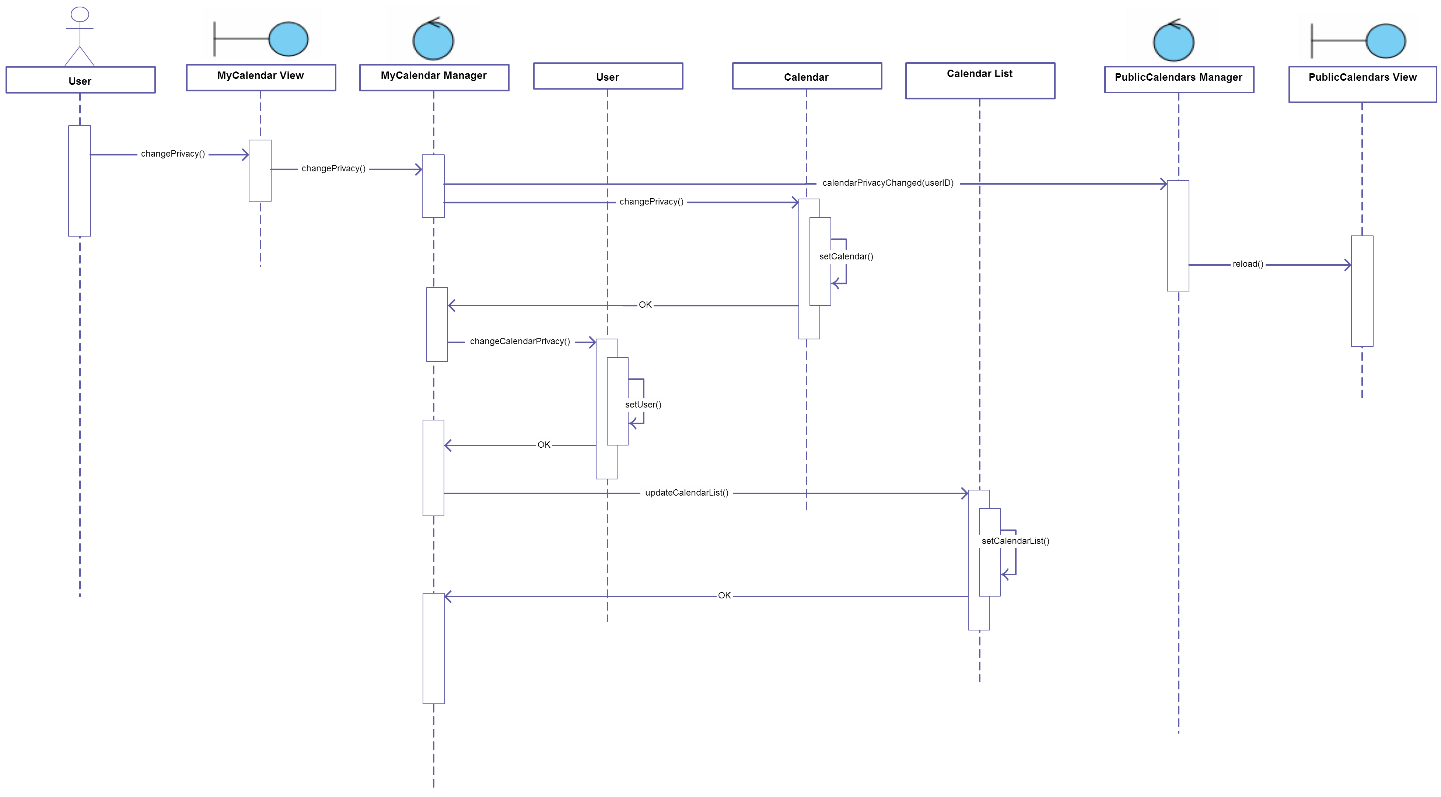
* Signs up as a new user of the system



## Change calendars privacy

A user:

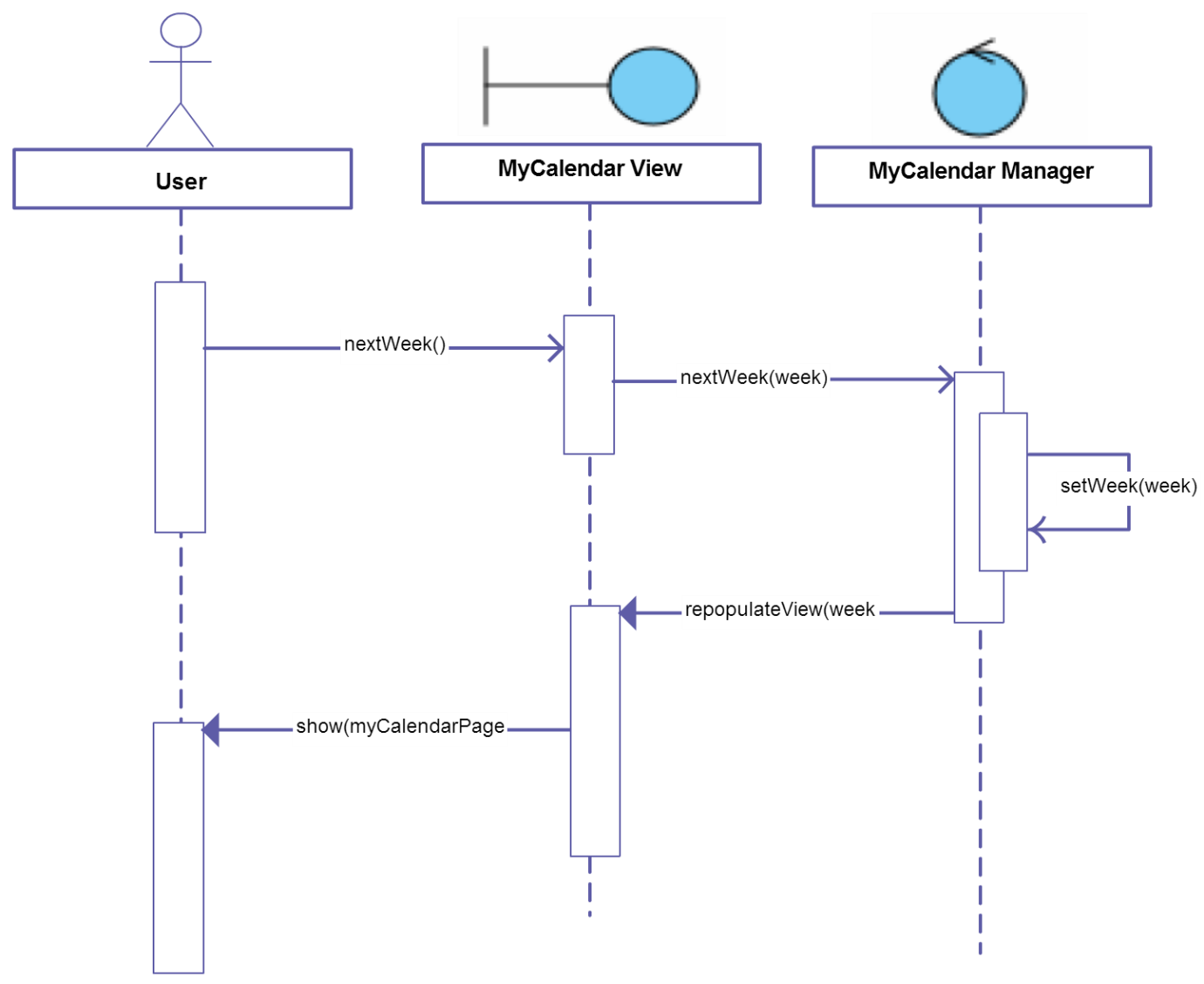
* Changes privacy setting of his calendar



## Next week

A user:

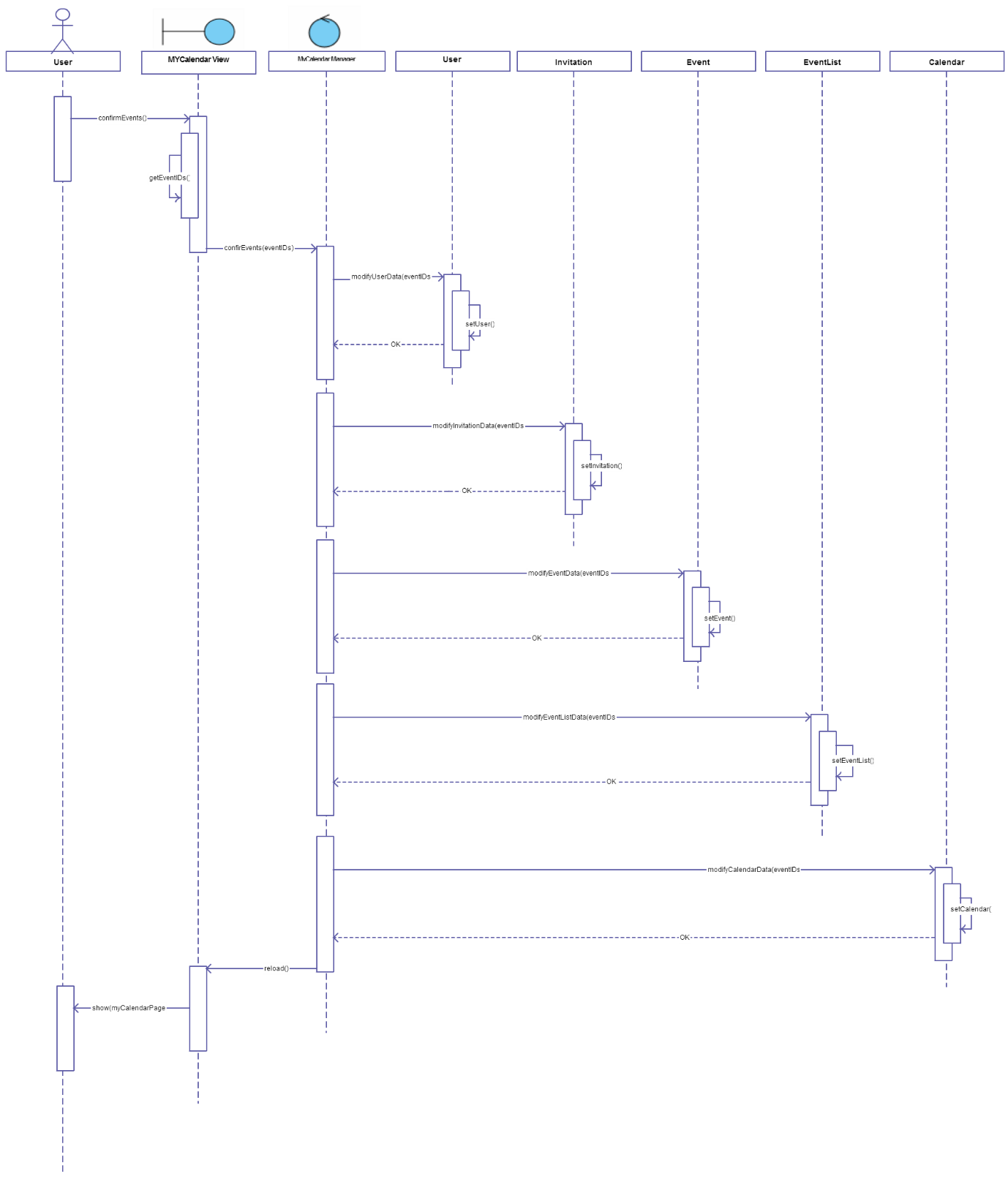
* User navigates through the calendar he is currently seeing



## Accept Invitations

A user:

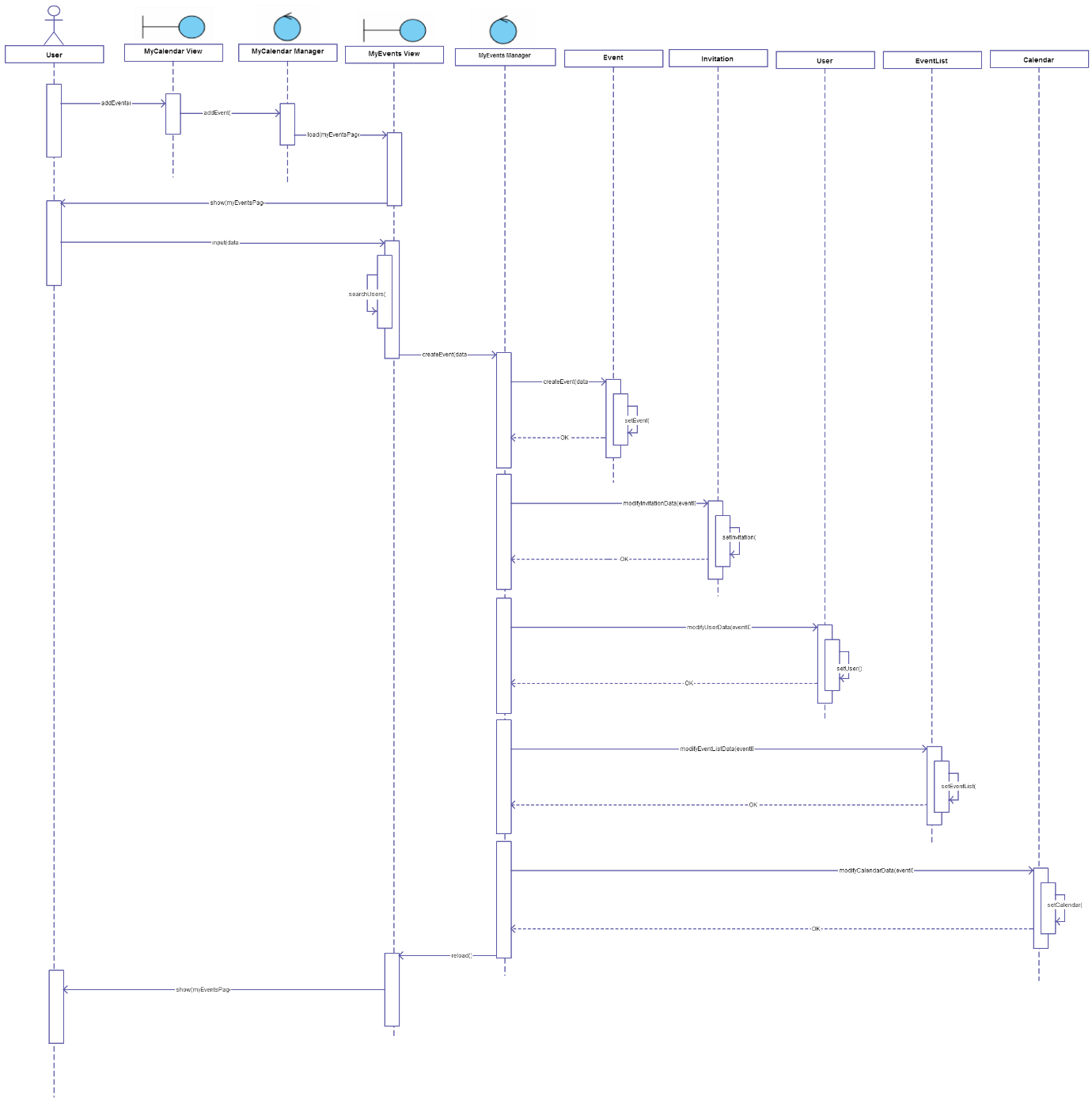
* User accepts invitations to the events he has selected



## Add Event

A user:

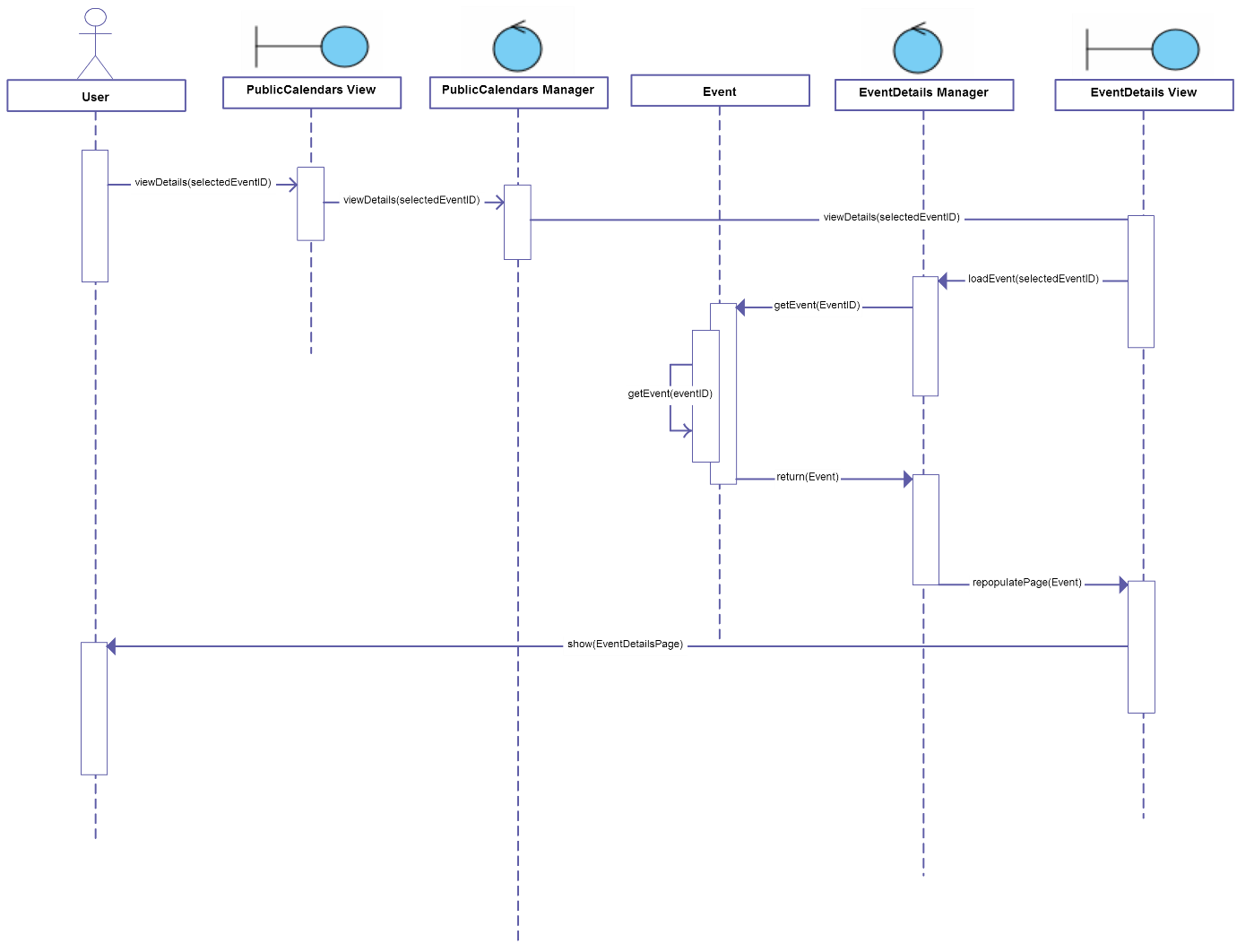
* User creates a new event



## 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 |