Project Zilean
A Timekeeping Application

An Ha Lele Fang Thijs Voncken

June 27, 2014

Contents

1 Foreword

The purpose of this document is to provide information on the design and development process of the Zilean Timekeeping Application (Zilean) project. It will start off with a brief description of the application, describing the functionality and specifications of the product. Towards the end we will reflect upon the design, development and the eventual product itself, indicating the parts which went better or worse.

This project was done in association with the Radboud University located in Nijmegen, the Netherlands.

2 Description

2.1 Introduction

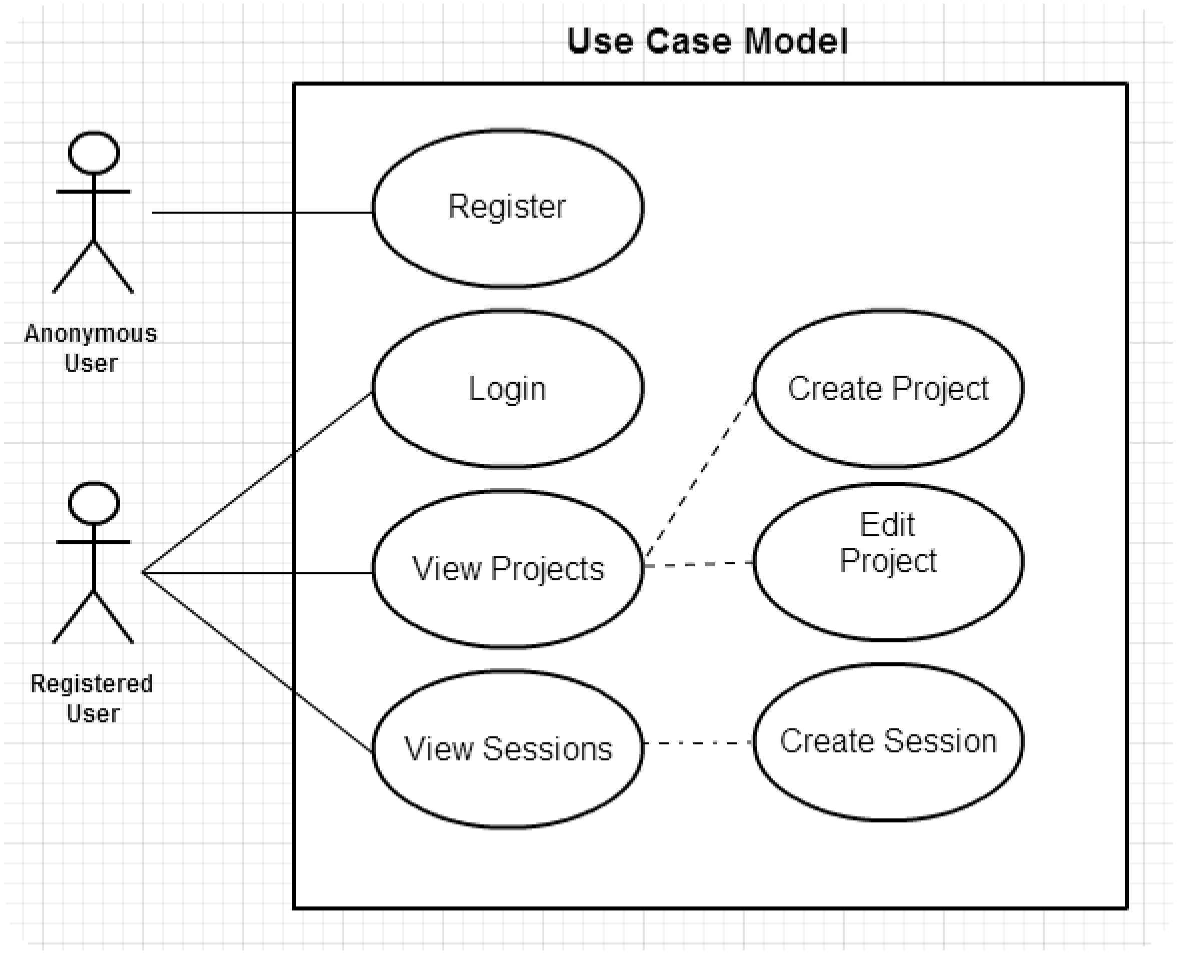
Zilean is an application focusing on improving both the workflow and the administration of teams. The Zilean application in its current state registers the amount of time the user has spent on a certain project by recording the starting time and ending time of a work-session. This data is stored on a centralised server which requires an account to access.

2.2 Product motivation

Zilean was designed to give users an easy way to log their time spent on a certain project. This could also be recorded with the average office tool such as Excel or simply with pen and paper. Using office tools is however highly inefficient, especially for mobile workers. A user also does not always have access to a paper or the file containing the session records, which results in a scattering of data across easy to forget files. Zilean provides a tool which is always available to the user since a mobile phone is a requirement for every working person when working mobile.

2.3 Specifications

2.3.1 Use Case Model



2.3.2 Non-functional Requirements

- Accessibility
- Availability
- Usability
- Privacy

2.3.3 Use Cases

Description	Create Project
Basic course of Events	1. User clicks on Projects in the side bar menu. 2. Application shows all current projects. 3. User clicks on the plus sign. 4. Application shows the form for a new project. 5. User submits forms by clicking 'Create'.
Preconditions Postconditions	Server is online and phone is connected to the internet. User has created a new project.

Description	Create Session
Basic course of Events	1. User clicks on Sessions in the side bar menu. 2. Application shows all current sessions. 3. User selects a project at the bottom of the screen and clicks start. 4. Application starts a timer. 5. User clicks on the stop button. 6. Application asks for confirmation of the session. 7. User clicks accept. 8. Application registers session.
Preconditions Postconditions	Server is online and phone is connected to the internet. User has created a new session.

3 Design

3.1 Global design

As specified in the specifications the application needs to be able to perform the following operations:

- Server connection
  - Register a new user
  - Authenticate
  - Receive data
  - Send data
- Application
  - Display the data
  - Handle user input

In order to accommodate these requirements, the application is comprised of the following components:

- Connection manager
  - Network Thread
  - Data Handler
- Application user interface

The application user interface displays the data received from the server and catches the data input from the user. From there, it can send this data over an established connection to the server by making use of the connection manager. The connection manager has two major components: The network thread, which handles the requests to the server, and the data handler, which adds the user input data to the requests and extracts the data from the server responses.

3.2 Detailed design

A detailed overview of the classes described in the following sections can be found in Appendix A, a description of client/server communication is included in Appendix B.

3.2.1 User interface

The project uses three main activities to handle visual display and user interaction. These are the LoginActivity, the RegisterActivity and the MainActivity. These activities also host the instances of the services required for the connection between the server and application, which will be covered in an upcoming paragraph. The MainActivity also hosts a number of fragments which are used as view elements, and handles the navigation between these fragments.

3.2.2 Server connection

There are a number of classes behind the scenes which manage the section between the application and the server. The most important ones, the DataService and NetworkHandlers, handle the transfer of data between the server and client as described in Appendix A. By using the DataService which is persistent throughout a user session, the activites requiring the connection are able to send and receive data from the server. The DataService does this through the NetworkHandler, which sends the requests and waits for a response, which then is sent back to the DataService for processing. The DataService then sends the processed data back to the Activity and, depending on the type of request and response, the view of the application is changed.

3.3 Design motivation

In the application design only three activities are used. The LoginActivity for logging in, the RegisterActivity for registering and the main activity for the activity itself.

Within the main activity fragments were used to produce different views for different actions. We chose to use fragments because we wanted to make the view contain compartments which we could reuse and also to have a persistent element in the view. Using fragments was the ideal approach to achieve this and using activities for this would have been harder because we would otherwise have needed to hardcode the different compartments in each activity. By using fragments the android API these processes were automated, including dynamically combining these fragments into one view.

Secondly, we chose to use a service to connect to the database. This allowed us to make a single connection to the database which we used for all communication with the server, which reduces the server load and modularizes our code by grouping together network code and separating it from any other code used in our application.

The server and application use HTTP requests and JSON objects to communicate. Since our application only updates in a few occasions and live notifications to the user were not needed an HTTP connection was ideal for the task. As for the choice of using JSON, it is a simple but powerful data format, which we thought would fulfill our needs.

4 Reflection

We are not content with our work. There are numerous reasons for this:

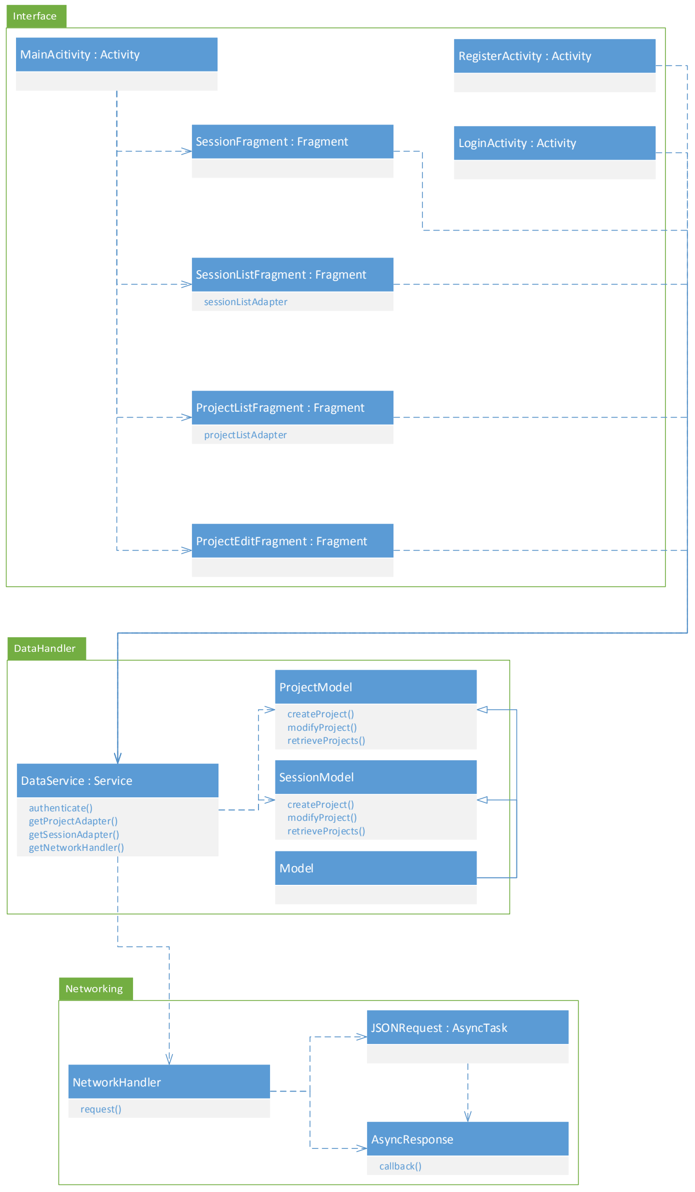
- The designs could have contained more room for scaling and new functions.
- We would have liked to have decent security.
- The application functionality is too limited now, we would have liked to see more features.

On top of that our organization has not been great throughout the project. We wasted a lot of time doing nothing while we should have been working, which resulted in rushing to finish the application towards the end and not really thinking about proper and robust deign. Through these issues this project has also shown us the value of time management and planning, had we done taken the time to plan our application properly we probably would have been far more content with the result.

4.1 What did we learn

Throughout this project we got familiar with the android API and learned basic python. We learned to appreciate the time spent designing an application and underlying systems much more and will certainly do this more often and more thoroughly in the future. By doing this we hope to make future projects go smoother and future end products to be easier to create and maintain.

A Appendix A - Application class design



B Appendix B - Server communication interface

B.1 General

All requests can at all times return the following response:
← 500 Internal Server Error:

```
{
    "error" : <Non-human readable error response>
}
```

B.2 User login/registration

Login Request → POST /authenticate

```
{
    "username" : <username>,
    "password" : <password>
}
```

← 200 OK:

```
{
    "id"              : <user id>
    "token"           : <token>
    "name"            : <user display name>
}
```

← 401 Unauthorized:

```
{
    "error" : "NO_AUTH"
}
```

← 403 Forbidden:

```
{
    "error" : "INVALID_AUTH"
}
```

← 419 Authentication Timeout:

```
{
    "error" : "EXPIRED_AUTH"
}
```

Registration Request → PUT /user

```
{
    "name" : <user display name>
    "username" : <username>,
    "password" : <password>
}
```

← 200 OK:

```
{
    "id" : <user id>
    "name" : <user display name>
    "username" : <username>,
}
```

← 400 Bad Request:

```
{
    "error" : "INVALID_FIELDS",
    "field_errors" :
    {
        "name" : <"TOO_SHORT"
                  | "TOO_LONG"
                  | "INVALID_CHARACTERS">,
        "username" : <"TOO_SHORT"
                      | "TOO_LONG"
                      | "INVALID_CHARACTERS">,
        "password" : <"TOO_SHORT"
                      | "TOO_LONG"
                      | "INVALID_CHARACTERS">
    }
}
```

← 409 Conflict:

```
{
    "error" : "DUPLICATE_USERNAME"
}
```

B.3 Authorized requests

All following request types require autentication, which means three additional response have to be added to every set of responses. The authentication token will have to be provided through the header X-Auth-Token.

← 401 Unauthorized:

```
{
    "error" : "NO_AUTH"
}
```

← 403 Forbidden:

```
{
    "error" : "INVALID_AUTH"
}
```

← 419 Authentication Timeout:

```
{
    "error" : "EXPIRED_AUTH"
}
```

B.4 Project management

Project retrieval request → GET /project[/<id>]
If the optional <id>is specified, only that single project will be returned in the array, otherwise all projects the user has access to are returned.

← 200 OK:

```
{
    "projects" :
    [
        {
            "id" : <project id>
            "name" : <project name>,
            "description" : <project description>,
            "owner" :
            {
                "id" : <project owner id>
                "name" : <project owner name>,
                "type" : <"USER" | "GROUP">
            }
            "users" :
            [
                {
                    "id" : <user id>,
                    "name" : <user display name>
                },
                ...
            ]
        },
        ...
    ]
}
```

← 200 OK:

```
{
    "id" : <project id>
    "name" : <project name>,
    "description" : <project description>,
    "owner" :
    {
        "id" : <project owner id>
        "name" : <project owner name>,
        "type" : <"USER" | "GROUP">
    }
    "users" :
```

```
    [
        {
            "id" : <user id>,
            "name" : <user display name>
        },
        ...
    ]
}
```

← 404 Not Found:

```
{
    "error" : "PROJECT_NOT_FOUND"
}
```

Project creation request → PUT /project

```
{
    "name" : <project name>,
    "owner" : <project owner id>
    "description" : <project description>
}
```

← 200 OK:

```
{
    "id" : <project id>
    "name" : <project name>,
    "description" : <project description>,
    "owner" :
    {
        "id" : <project owner id>,
        "name" : <project owner name>,
        "type" : <"USER" | "GROUP">
    }
}
```

← 400 Bad Request:

```
{
    "error" : "INVALID_FIELDS",
    "field_errors" :
    {
        "name" : <"TOO_SHORT"
                 | "TOO_LONG"
                 | "INVALID_CHARACTERS">,
        "owner" : "OWNER_NOT_FOUND",
        "description" : <"TOO_LONG"
                        | "INVALID_CHARACTERS">
    }
}
```

Project deletion request → DELETE /project/<id>

← 200 OK:

```
{
    "message" : "OK"
}
```

← 404 OK:

```
{
    "error" : "PROJECT_NOT_FOUND"
}
```

Project alteration request → PATCH /project/<id>
The users field in the request body is supposed to contain all users the project still has after the alteration request.

```
{
    "name" : <project name>,
    "description" : <project description>,
    "users" : [<user id>, <user id>, ...]
}
```

← 200 OK:

```
{
    "id" : <project id>
    "name" : <project name>,
    "description" : <project description>,
    "owner" :
    {
        "id" : <project owner id>,
        "name" : <project owner name>,
        "type" : <"USER" | "GROUP">
    }
    "users" :
    [
        {
            "id" : <user id>,
            "name" : <user display name>
        },
        ...
    ]
}
```

← 400 Bad Request:

```
{
    "error" : "INVALID_FIELDS",
    "field_errors" :
    {
        "name" : <"TOO_SHORT"
                 | "TOO_LONG"
                 | "INVALID_CHARACTERS">,
        "description" : <"TOO_LONG"
                        | "INVALID_CHARACTERS">,
        "users" :
        [
            {"id" : <"USER_NOT_FOUND"
```

```
                              | "DUPLICATE_MEMBER">}
            ]
      }
}
```

← 404 Not Found:

```
{
      "error" : "PROJECT_NOT_FOUND"
}
```

B.5 Group management

Group retrieval request → GET /group[/<id>]
If the optional <id>is specified, only that single group will be returned in the array, otherwise all groups the user has access to are returned.

← 200 OK:

```
{
    "groups" :
    [
        {
            "id" : <group id>
            "name" : <group name>,
            "description" : <group description>,
            "owner" :
            {
                "id" : <group owner id>
                "name" : <group owner name>
            }
            "users" :
            [
                {
                    "id" : <user id>,
                    "name" : <user display name>
                },
                ...
            ]
        },
        ...
    ]
}
```

← 200 OK:

```
{
    "id" : <group id>
    "name" : <group name>,
    "description" : <group description>,
    "owner" :
    {
        "id" : <group owner id>
        "name" : <group owner name>
    }
    "users" :
    [
        {
            "id" : <user id>,
            "name" : <user display name>
        },
        ...
    ]
}
```

← 404 Not Found:

```
{
    "error" : "GROUP_NOT_FOUND"
}
```

Group creation request → PUT /group

```
{
    "name" : <group name>,
    "description" : <group description>
}
```

← 200 OK:

```
{
    "id" : <group id>
    "name" : <group name>,
    "description" : <group description>,
    "owner" :
    {
        "id" : <group owner id>,
        "name" : <group owner name>
    }
}
```

← 400 Bad Request:

```
{
    "error" : "INVALID_FIELDS",
    "field_errors" :
    {
        "name" : <"TOO_SHORT"
                 | "TOO_LONG"
                 | "INVALID_CHARACTERS">,
        "description" : <"TOO_LONG"
                        | "INVALID_CHARACTERS">
    }
}
```

Group deletion request → PUT /group

```
{
    "name" : <group name>,
    "description" : <group description>
}
```

← 200 OK:

```
{
    "message" : "OK"
}
```

← 404 Bad Request:

```
{
    "error" : "GROUP_NOT_FOUND"
}
```

Group alteration request → PATCH /group/<id>
The users field in the request body is supposed to contain all users the group still has after the alteration request.

```
{
    "name" : <project name>,
    "description" : <project description>,
    "users" : [<user id>, <user id>, ...]
}
```

← 200 OK:

```
{
    "id" : <group id>
    "name" : <group name>,
    "description" : <group description>,
    "owner" :
    {
        "id" : <group owner id>,
        "name" : <group owner name>
    }
    "users" :
    [
        {
            "id" : <user id>,
            "name" : <user display name>
        },
        ...
    ]
}
```

← 400 Bad Request:

```
{
    "error" : "INVALID_FIELDS",
    "field_errors" :
    {
        "name" : <"TOO_SHORT"
                | "TOO_LONG"
                | "INVALID_CHARACTERS">,
        "description" : <"TOO_LONG"
                        | "INVALID_CHARACTERS">,
        "users" :
        [
            {"id" : <"INVALID_ID"
                    | "DUPLICATE_MEMBER">}
        ]
    }
}
```

← 404 Not Found:

```
{
    "error" : "PROJECT_NOT_FOUND"
}
```

B.6 Session management

Session fetching → GET /session/<user |group |project>/<id>
This request returns all sessions per user, group or project. The id passed is the user, group or project id you want to get all sessions for.

← 200 OK:

```
{
    "sessions" :
    [
        {
            "user" :
            {
                "id" : <user id>,
                "name" : <user display name>
            },
            "project" :
            {
                "id" : <project id>,
                "name" : <project name>,
                "description" : <project
                   description>
            }
            "timestampEnd" : <timestamp end>,
            "timestampStart" : <timestamp start>
        },
        ...
    ]
}
```

← 404 Not Found:

```
{
    "error" : <"USER_NOT_FOUND"
                | "PROJECT_NOT_FOUND"
                | "GROUP_NOT_FOUND">
}
```

Session registration → PUT /session

```
{
    "project" : <project id>,
    "timestampStart" : <starting timestamp>,
    "timestampEnd" : <ending timestamp>
}
```

← 200 OK:

```
{
    "timestampStart" : <starting timestamp>,
    "timestampEnd" : <ending timestamp>,
    "owner" :
    {
        "id" : <owner user id>,
        "name" : <owner user name>
    },
    "project" :
    {
        "id" : <project id>,
        "name" : <project name>,
        "description" : <project description>
    }
}
```

← 400 Bad Request:

```
{
    "error" : "INVALID_FIELDS",
    "field_errors" :
    {
        "project" : "PROJECT_NOT_FOUND",
        "timestampStart" : "
            TIMESTAMP_START_AFTER_END",
        "timestampEnd" : "
            TIMESTAMP_END_BEFORE_START"
    }
}
```

B.7 User management

User fetching → GET /user/<username>

← 200 OK:

```
{
    "user" :
    {
        "id" : <user id>,
        "name" : <user display name>
    }
}
```

← 404 Not Found:

```
{
    "error" : "USER_NOT_FOUND"
}
```