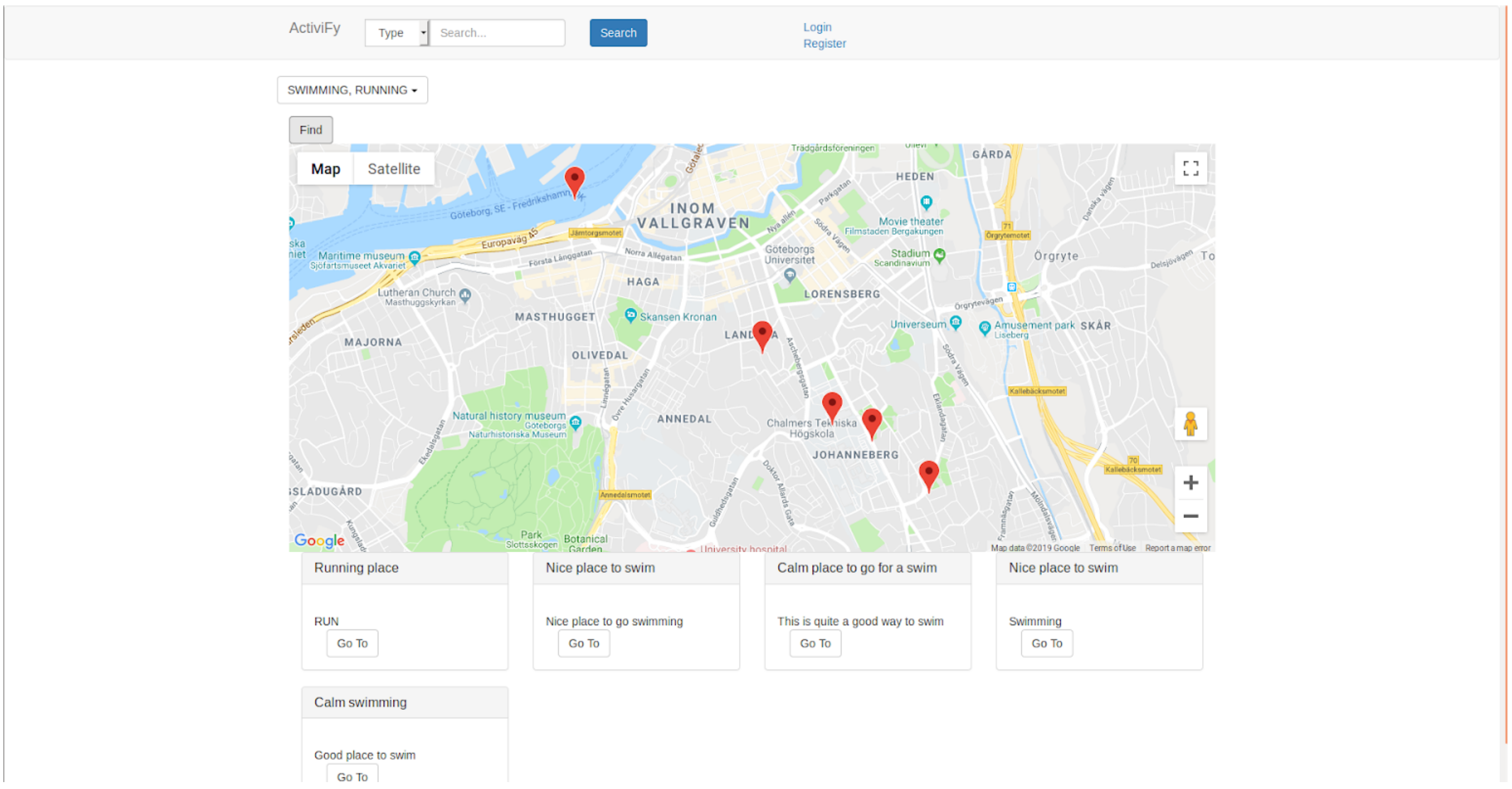
Activify

Group 12

DAT076 / DIT126 web applications



Running place

RUN

Go To

Nice place to swim

Nice place to go swimming

Go To

Calm place to go for a swim

This is quite a good way to swim

Go To

Nice place to swim

Swimming

Go To

Calm swimming

Good place to swim

Go To

Adam Grandén
Gustav Albertsson
Robin Hekmatara
Johan Svennungsson

1. Introduction

The webapp is a community driven web based platform for sharing and discovering activities. Users can create their own accounts and submit locations with activities such as places to go jogging and other outdoor activities. It's a tool that help users find places to perform their activities or discover new activities in their area. Users can search for activities in different ways, either by city or by the type of activity.

The repo for the webapplication is: https://github.com/DukeA/Dit126-app

2. A list of use cases

- Add activity - The user should be able to add an activity
- Show activity (detailed view) - The user should be able to get a detailed view of a single activity
- Remove activity - The owner of an activity should be able to remove it
- Edit activity - The owner of an activity should be able to edit it.
- Search activity by place - Users should be able to search for a city and get the activities in that city
- Search activity by type - Users should be able to search for a type of activity and get the activities with that type
- List all activities by type of activity - Users should be able to only get a list of activities with the prefered types.
- Login - The users should be able to log into the system
- Register - The users should be able to register an account
- Logout - The users should be able to logout

3. User manual

- Add place/activity - When a user is logged in he/she can press "Add activity" in the top menu. When pressing "Add activity" the user is presented with a map where a pin should be placed to indicate where the activity is. Pressing "Next" will give the user input fields that indicates what the activity is such as title, description and activity type. Pressing "Add" will now save the activity with the logged in user as its owner. The user will now get redirected to a page that displays detailed information about the activity.

- Show activity (detailed view) - When searching for an activity or when filtering for activities the user can get a better view of the information that activity has by pressing "Details" or "goto" when looking at the results of a search/filtering.

- Remove activity - A delete button is displayed when a user is logged in and viewing the detailed view of an activity he/she owns. Pressing this button will permanently delete the activity. The user will be redirected to the list activity view afterwards.

- Edit activity - The owner of an activity should be able to edit it.
  When a user is logged in he/she can edit her own activities. To do this the user must first find the activity that the user wants to edit. Going into detailed view for the activity will present the user with an edit button only if they are the owner of that post. Pressing “Edit” will present the same 2 stage process as the Add activity. Where the user can change the information needed in order to fit their needs. The information that can be changed while editing an activity is the same as when adding an activity

- Search activity by type - To find activities based on a certain type, e.g. “swimming” the user can search for Type in the MultiMenu next to the search field in the navbar.

- Search activity by city - Same goes for searching by city, but choosing City in the MultiMenu instead.

- List all activities by type of activity - To list activities by types, a user have to press “Activities” link in the navbar to get to the list-activities page. Once a user is at the list-activities page he/she needs to select the desired types from a drop down box. When the types of activities has been selected the user needs to press the find button which will update the map to show all the activities for the chosen types. Boxes with activity information will be displayed under the map, each containing a button. Pressing the “Go To” button will redirect the user to show activity view.

- Login - In order for functions such as editing your own activity, it’s necessary for the user to login. The user presses the login button which does only appear in the navbar when the user is not logged in. The user then inputs it’s username and password and clicks the login-button, if the username exists and the password is correct the user will be logged in.

- Register - To be able to log in to the application the user must be able to register into the website. By pressing “register” in the navbar will bring the user to a register page. Here the user is then able to input the username and the password, the username needs to be unique. So there can’t be two users with the same username.

- Logout - When the user is logged in a button appears in the navbar that allows the user to logout.

4. Design

4.1 Frameworks

The application is built using Java EE and JSF. The database is PostgreSQL. We use Maven to build the project and bundle it in a war, which is then deployed to a running Payara Server. To be able to quickly make the frontend look reasonably well and make it responsive we have chosen to use BootsFaces, which allows us to use Bootstrap 3 on our website in an easy way. The application uses Spring Security to hash and to verify the users passwords. Spring is an established framework used by the industry and we trust their implementation of the security is sufficiently good.

4.2 Software architecture

The web application is built using the MVC architectural pattern where the .xhtml files is the **V**iew that handles the display logic. The **C**ontroller are the beans that handles the interaction between the view and the model. This layer is also responsible to contact any classes in the util package that is needed. The **M**odel part of the application consists of entities and facades.

4.3 Testing

The testing of the code is done using JUnit and Mockito, the choice to use Mockito is because it can easily mock things. For instance when tests are done to the AddActivity class things needs to get mocked in order to ensure that the correct calls to the facades are made. We cannot test it without mocking since entity managers are used and we require a database to use them properly.

4.4 External APIs

Two parts of the application uses the Google Maps API. The view part uses the API in order to display maps to the user that they can interact with, this is done via the Google Maps JavaScript API. The other part of the application which uses the Google Maps API is when adding/ editing an activity the city will get automatically retrieved based on the activities location. In order to keep this external API separate the from the rest of the code and make it easy in the future to switch API this connection was put in the */util/HttpRequest* package that uses an factory pattern in order to abstract away any internal implementation.

4.5 Database

The database as mentioned earlier is a PostgreSQL database. This database consists of 3 tables. app_user, activity and location.
The tables in the database consists the of the following attributes:

Location:

- location_id, integer, primary key
- latitude float
- longitude float
- city varchar

The Location table holds the information about a location on the map, that is the longitude, latitude, and the city that it is in. The reason to save the city and not look it up at from coordinates everytime is to allow more efficient search by city and not having to do more calls then necessary to the Google Maps API.

App_user:

- user_id, integer, primary key
- user_name, varchar
- user_password, varchar

The App_user table holds the information about a user, the only thing needed at this point in time is the username and the password. This could in future be extended to emails or other attributes needed to reflect the user.

Activity:

- activity_id, integer, primary key
- title, varchar
- type, varchar
- description, varchar
- location_id, references Location.location_id
- user_id, references App_user.user_id

The activity table contains information about an activity. An activity has some attributes that describe it such as the title, type and description. It also has foreign keys to a location and a user. The user foreign key are used to find the owner of the activity. The location foreign key is used to find the location that the activity is located at.

4.6 Data validation

The data that the user enters on different fields on the website is validated on the server and often a helpful error message is displayed on the front end, for example logging in with the wrong password will display an error message to the user saying that the login information is wrong. This is done throughout the application and the data from the user is never trusted. When editing an activity the first thing that happens is that the user gets validated to see if they are really the owner of the activity they are trying to edit. If they are not the owner they will get redirected to the index page. To have extra security from malicious behaviour the backend also checks that the user is really the owner before updating the entities.

5. Application Flow

Figure[1] shows how Register works for the Application , Figure [2] shows how search for an activity works in the application and Figure[3] shows how the User is able to add to activity into the database.

Register

When a user is at the register screen he/she should enter a username and a password. These are then sent to the class Register, which is a named bean. The first thing that is done is to check that the user has entered a username and a password, if not the user will get an error message. When the username and password is validated the Register class checks if the user name is already taken, this is done by calling checkUsername in the RegisterFacade. RegisterFacade then creates a query and retrieve the information needed from the database. Based on the response of this the Register controller either allows the registration or disallows the registration. If the registration is allowed (i.e. no user exists with that username) the password is hashed, the user is added to the database, and the user gets redirected to the login page. If the registration is not allowed (i.e. a user already exists with that username) the user will not be added to the database and the user will get redirected back to the register page.

Add activity

When a user is adding an activity on the add activity page, he/she would first enter the coordinates of the activity on the screen and then the Activity information. When pressing “Add”, the data would get sent to the AddActivity controller. If the user is not logged in the add method will return null and prevent the adding of an activity. If the user is logged in the data for the ActivityEntity would be set, these values would be title, description and type. The other data values would then be set for the activities location and user which posted the activity. To get the location of the activity being created, the Add activity sends a request to the Google Maps API with the latitude and longitude of the city. The API will then respond with the nearest city, this will be saved together with the longitude and latitude in the LocationEntity. Then the ActivityEntity is created in the database using the ActivityFacade.

Search by type/city

When searching for an activity the user selects if they are searching for an activity type or a city. These gets sent to the search bean which holds the input. The search method then gets called in the searchResults controller. The controller then retrieves the information that the user has typed in. If the user is searching for a type then FindByType is called on the ActivityFacade otherwise FindByCity is called. Both of these methods return a list of ActivityEntity which are then displayed on the results page.

6. Responsibilities

The responsibilities have been split based on the use cases, so each of us have a set of use cases that we have implemented.

Adam Grandén
Show activity (detailed view) - Implemented a way to show an activity
Register - Implemented a register system

Gustav Albertsson
Add place/activity - Implemented the entire use case
Edit activity - Implemented the entire use case
Login - Made a first unsave version to serve as a basis for future development
Logout - Made the logic for logging out

Robin Hekmatara
Remove activity - Implemented removing activity, bean part in ShowActivity, view in ShowAct.xhtml
List all activities by type of activity - Implemented listing activities by type of activity; ListActivity, index.xhtml, display-activities.js

Johan Svennungsson
Search activity by place - Implementing searching of activity by city
Search activity by type - Implemented searched of activity by type of activity
Login - Made the login secure using bcrypt
Logout - Made the logout available to the user when logged in

7. The code on GitHub

Since parts of the code is auto generated from Netbeans/Intellij so some have gotten “free” lines of code, the code that is auto generated are the classes in the model package. The entities and the facades are auto generated however changes have been made to these files for example add a custom query, so one cannot simply discard these files when looking at the contributions.

We did a restructure of the code late in the project to remove some redundant folders, this caused the “contributors” of a file to only show one person as the author of it. To know who wrote which file one can browse the repository before the restructuring was done. This is the

commit right before the restructure. However *gitinspector* still seems to show the correct lines of code.
https://github.com/DukeA/Dit126-app/tree/fe9625ef94b1a7207cee655161d88c13d3b13336

The code on GitHub will NOT work as it did during the presentation, the reason for this is that the web application uses the Google Maps API that needs a valid API key to be able to get the city names. When running the unit tests for the application some tests will also fail because of this reason with the exception "**No API-key supplied**". In order to fully test the web application an API key has to be supplied. An API key can be gotten at https://cloud.google.com/maps-platform/#get-started. To get an API key a credit card has to be linked to the Google account. The api key should be placed in ApiKey class which can be found in */util/HttpRequest/.*

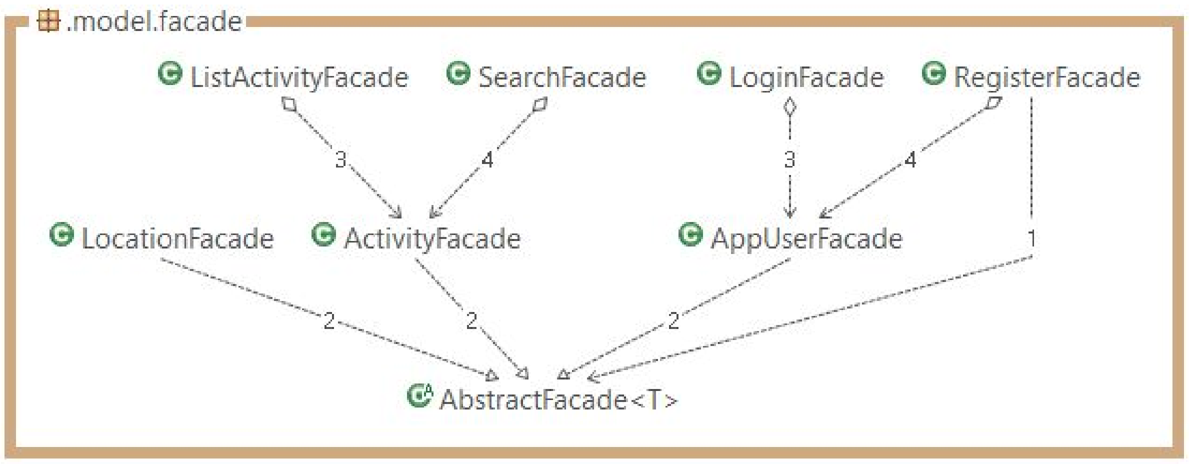
8. Early testing

There are a few branches on Github (other than develop and master) that is left. These are early work during the first week of the project where we tested different frameworks and languages that we wanted to use. E.g. JAX-RS, Spring, React, google-maps-react, JWT. Ultimately we settled for Java EE and JSF because we were unsure how to handle sessions for logged in users and didn't want to spend too much time finding potential solutions.

We have left this in order to show what we did during the first week since we put quite a lot of work into trying different technologies.

9. Further improvements

The design of the facades in the *model/facade* package are not structured in an optimal way. At the moment there is an auto generated AbstractFacade which contains some basic functionality and then we have 3 basic facades, one for each entity that extends the AbstractFacade and lets us use the basic methods from the AbstractFacade. We then wanted to have more facades that extends these basic facades with more functionality for example one facade that contains for example the methods for searching. The reason for this is that we only want to inject facades with a minimal amount of methods needed to the different controllers. We don't want to give the AddActivity controller access to the searching methods. However we encountered problems when doing this because of the injection. We then decided to still have the facades that have more functionality then the basic facades but instead use the Delegation pattern to still be able to access parts of the basic facade. This works well but it required us to make the entity manager public instead of protected. The image below displays how the facades structure ended up as.



Another improvement that could be done is to enable HTTPS, at the moment the passwords are hashed and salted when they are added into the database but the communication is still done without encryption over HTTP.

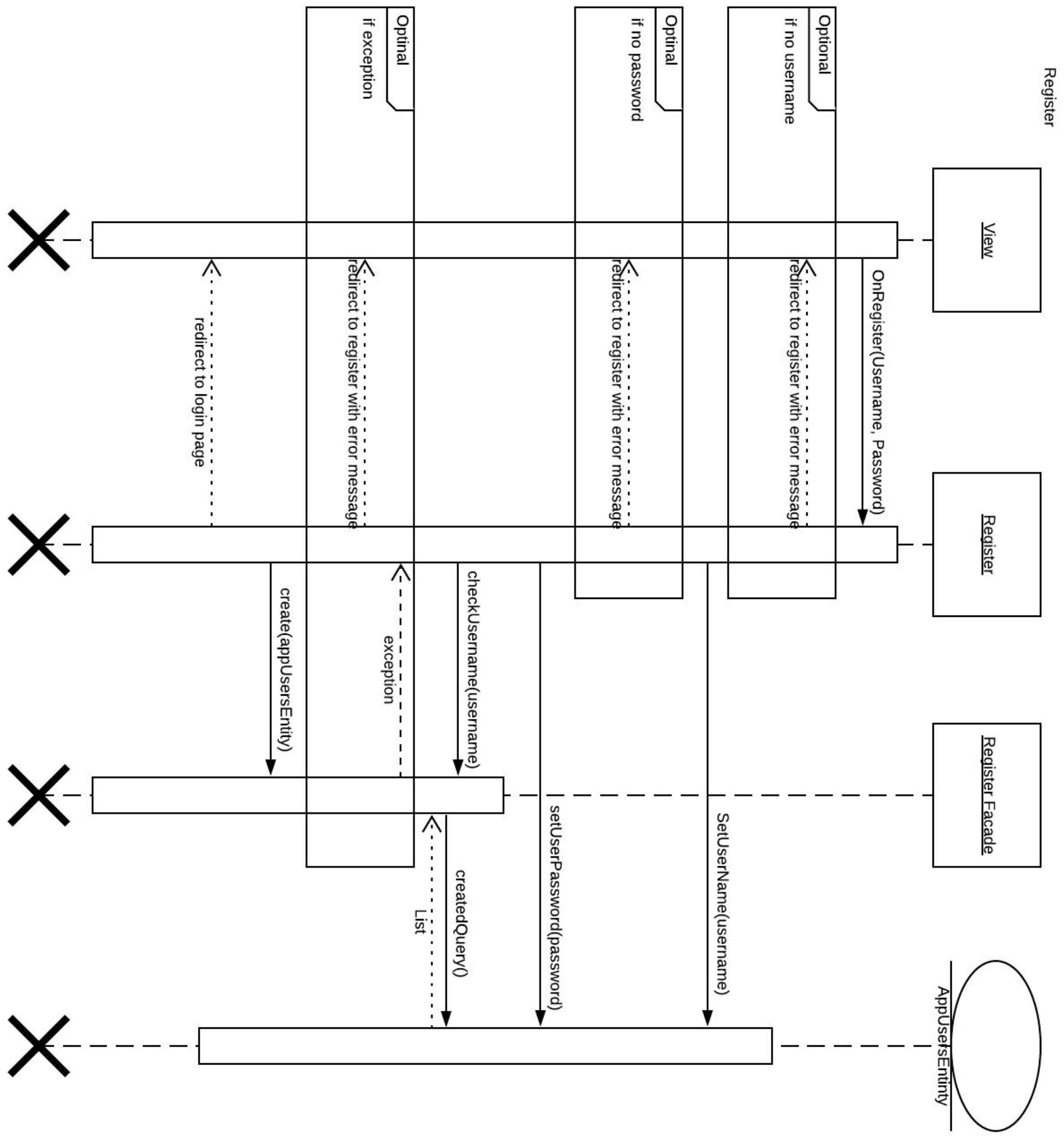
Another improvement is that although we don't save any incorrect/corrupt information to the database the messages displayed to the user are not always as helpful as they could be.

10. Credit

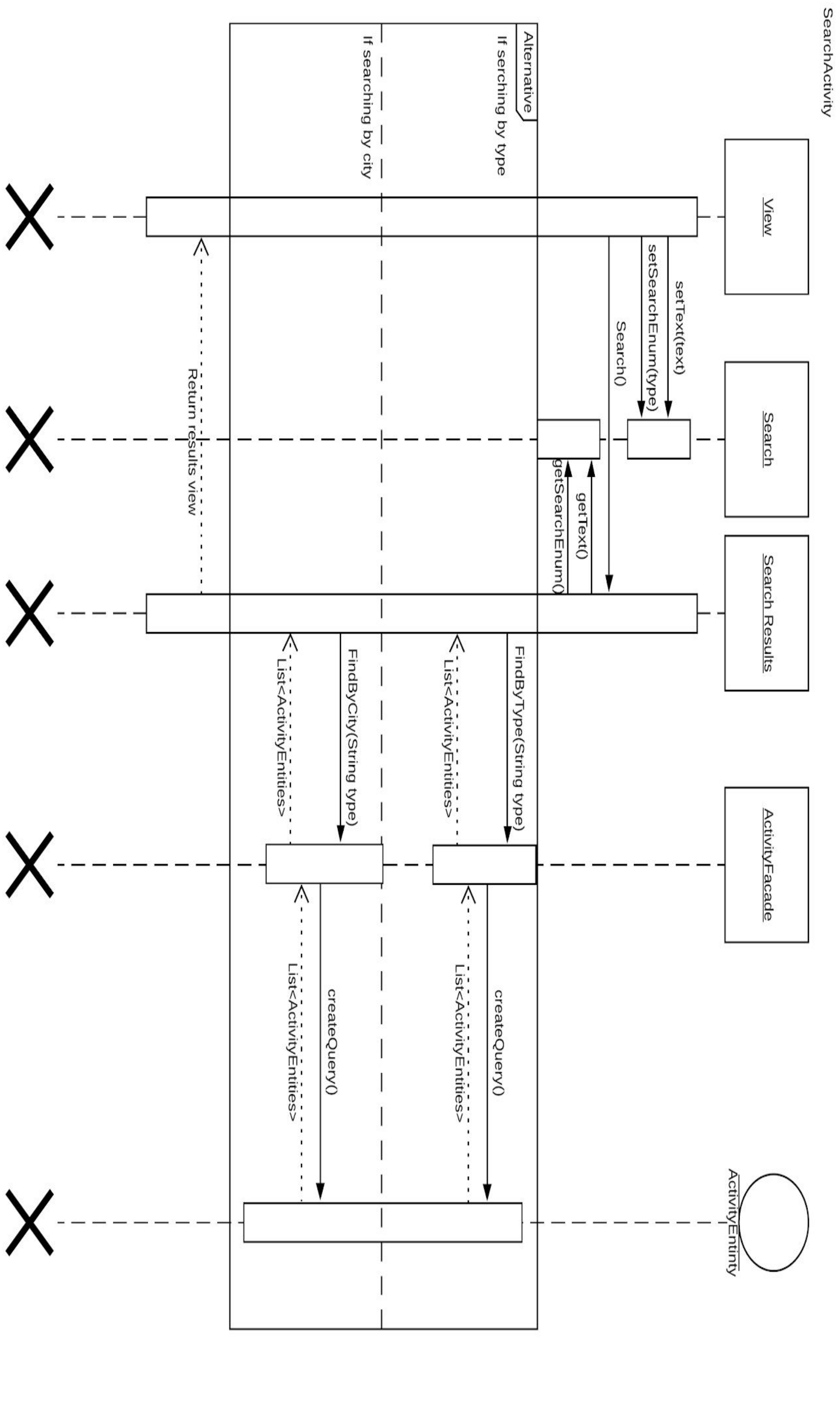
In the beginning of the project we had major issues getting the code to connect to the PostgreSQL database until we found an example from a Payara developer on Github.

https://github.com/MattGill98/Payara-Examples/blob/4b70f90b61eaecf8c451d90b96e0383441e18600/Payara-Micro/jpa-datasource-example/src/main/webapp/WEB-INF/web.xml

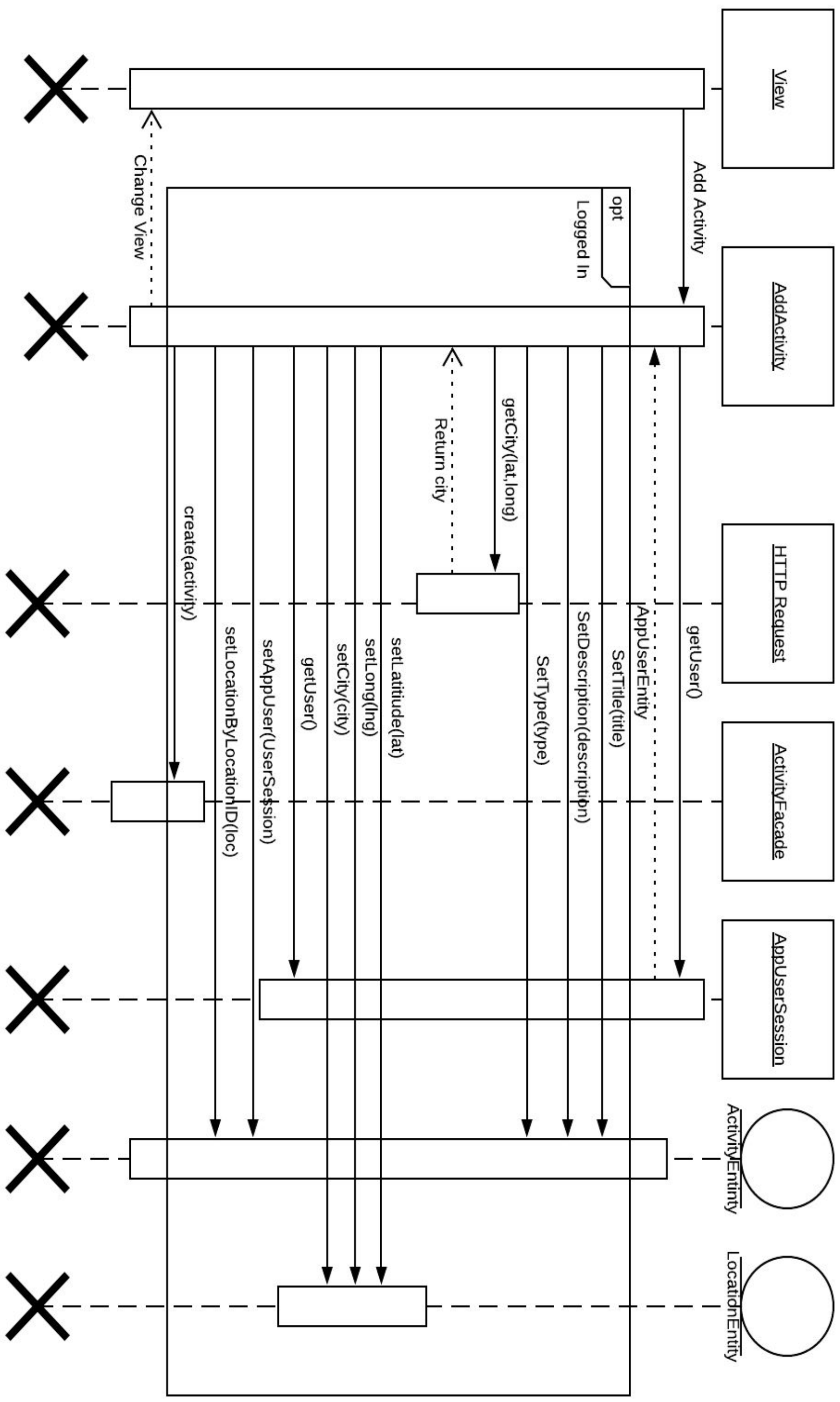
Which means our web.xml-file is more or less copy-pasted straight from this example.



Figure[1]: Sequence diagram for registering a user



Figure[2]:Sequence diagram for searching for an activity



Figure[3]: Sequence diagram for adding an activity