Conference Management System (CMS)

**Group 924**

**Team members:**

**Jurl Flaviu**

**Irimia Cristiana**

**Luca Laura**

**Marcu Daniel**

Table of Contents

1 OVERVIEW …………………………………………………………………………… **3**

2 TOOLS USED …..…………………………………………………………………….. **4**

*2.1 Modeling tools ……………………………………………………………………………...* 4

*2.2 Programming languages …………………………………………………………………. 5*

*2.3 Frameworks ………………………………………………………………………………... 5*

*2.4 Libraries …………………………………………………………………………………….. 7*

*2.5 Database …………………………………………………………………………………… 7*

3 MODELING … …..…………………………………………………………………….. **8**

*3.1 Early Stage ……………………………………………………….…………………........ 8*

*3.2 Class Modeling …………………………………………………………………………… 11*

*3.3 Sequence Diagrams …………………………………………………………………….. 13*

4 SCHEDULING ……….…………………………………………….………………… **20**

5 IMPLEMENTATION….………………………………………………………………. **21**

*5.1 Main layout and Project Diagrams ……………………….…………………................ 21*

*5.2 Testing …………………………………………………………………………………….. 31*

5 FINAL THOUGHTS ….………………………………………………………………. 33

Overview

This documentation provides an understanding of the stages our team went through while designing and implementing the task at hand. The objective was to learn about different useful tools for designing and application and apply them in a project. Artifacts such as class diagrams use case diagrams etc. are also provided in this paper, as they were crucial for the implementation of the task.

This document also provides information regarding the frameworks used, as well as reasoning behind our choices.

Tools Used

This section of the paper covers the tools used to model the task given, as well as different languages and frameworks used.

**Modeling Tools**

In order to model the specifications given by the professor our team used **Visual Paradigm** and **StarUML.**

**Visual Paradigm** is a UML CASE Tool which supports UML2, SysML and Business Process Modeling Notation from the Object Management Group. This tool was used to identify important aspects of the task given, such as Actors and Use Cases. Moreover, we designed the first Class Diagram for our base entities using Visual Paradigm.

**StarUML** is a UML tool owned by MKLab. We used this tool to later re-model our diagrams when needed.

**Programming Languages**

For our application we chose two main programming languages, namely **Java** and **Typescript**.

**Java** is a programming language that is class-based and object-oriented. We used this programming language for the Backend side of our application, as all the members from our team was familiar with it.

**Typescript** is an open-source programming language. It is a strict syntactical superset of JavaScript. This programming language was used to develop the Frontend side of our application.

**Frameworks**

The main frameworks used for implementing the software are: **Spring, Hibernate ORM** and **Angular.**

**Spring** is an application frameworks and inversion control container for the Java platform. This framework provided different modules to help us manage the implementation of the application.

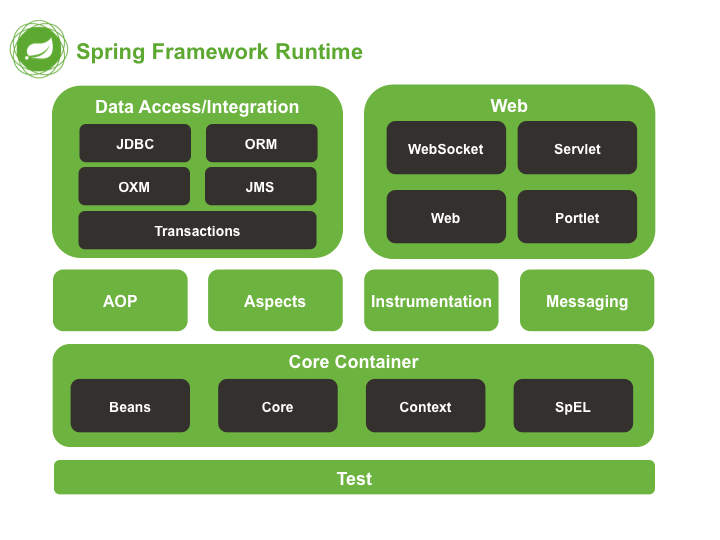


Figure 1 : Spring Modules

**Spring** was used to:

* **Inject dependencies** in objects and **manage the object lifecycles**, such as creating them, calling their initialization methods, and configuring the objects by wiring them together.
* **Access data** from the database with the help of **Hibernate,** by supporting resource management, exception handling, transaction participation, resource unwrapping and abstraction for binary large object and character large object.
* **Manage requests from the Web** using Spring MVC, which contains interfaces such as Controller (which manages incoming requests and redirects to proper response), HandlerAdapter, HandlerMapping, LocaleResolver, MultipartResolver, View and ViewResolver.

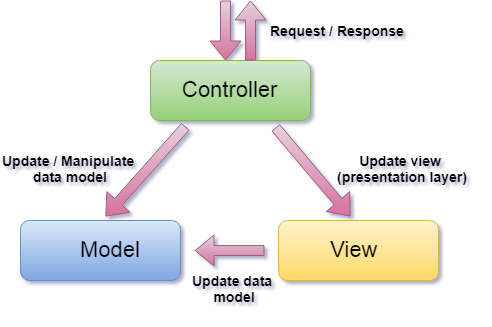


Figure 2: MVC representation

To manage the Server part of our application, our team used **Spring Boot** to build and launch the application faster.

**Hibernate ORM** is an object-relational mapping tool for Java programming languages. It provides a framework for mapping an object-oriented domain model to a relational database.

This tool was used to handle object-relational impedance mismatch problems by replacing direct, persistent database accesses with high-level object handling functions. Its primary task is to map Java classes to database tables and their fields to database columns. Hibernate also provides data query and retrieval facilities.

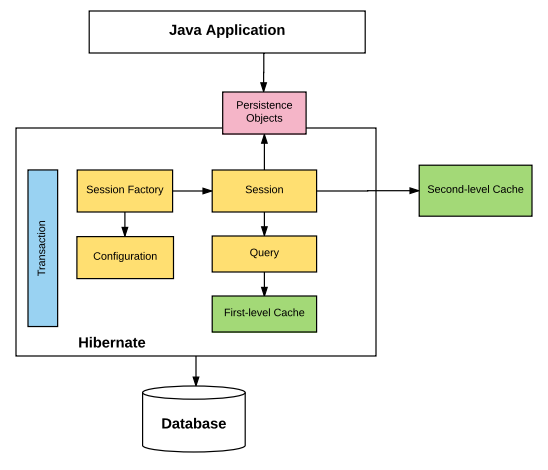


Figure 3: Hibernate Architecture

**Angular** is a Typescript-based open-source web application framework, used to design and implement the Frontend side of the application. This framework helped us design the user interface in a well-organized manner.



Figure 4: Angular Architecture

**Libraries**

**Project Lombok** is a java library which uses annotations to generate code, such as getters and setters. We used this library to shorten the amount of code written for less clutter in our classes.

**Database**

**MySQL** is an open-source relational database management system, which we used to store the data for our application.

Modeling

**Early stage**

In the early stage, we began by analyzing the text given by the professor and identifying the critical point in the requirements in order to better understand and divide the text into tasks and functionalities.

First, we identified the Actors and the Use Cases and we designed the first Use Case diagram using Visual Paradigm. This diagram was handed in to the laboratory teacher for feedback.

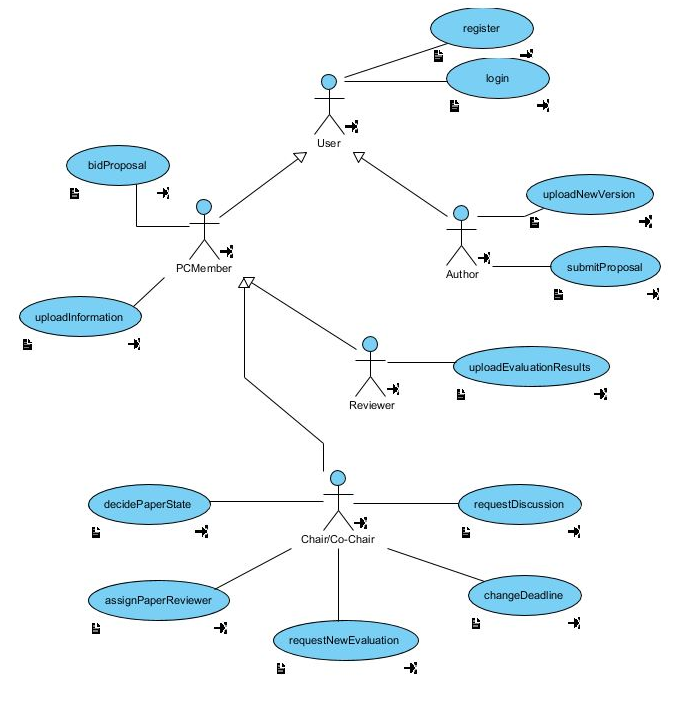


Figure 5: First Use Case Diagram, Part 1

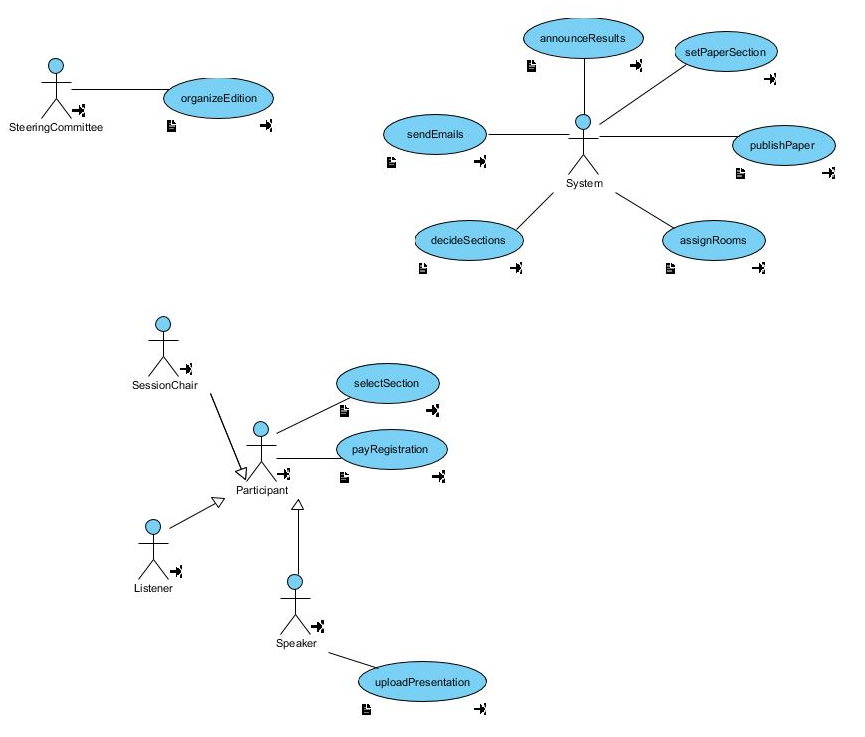


Figure 6: First Use Case Diagram, Part 2

However, later we decided to re-design the diagram using StarUML.

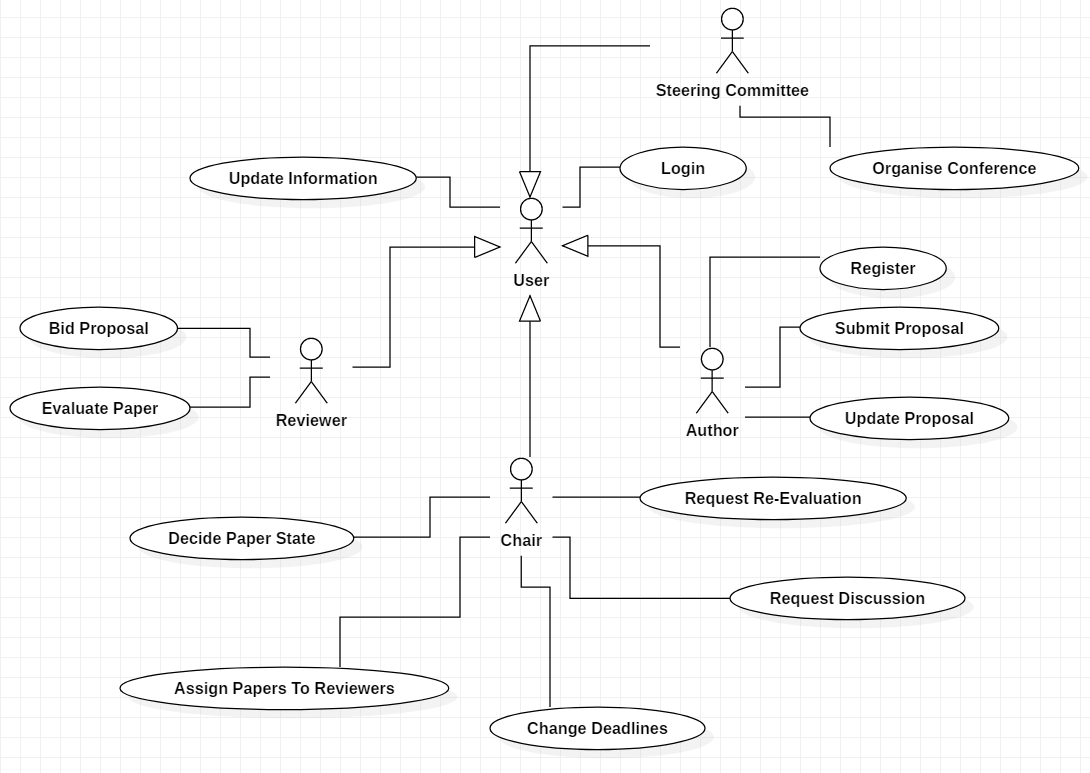


Figure 7: Second Use Case Diagram, Part 1

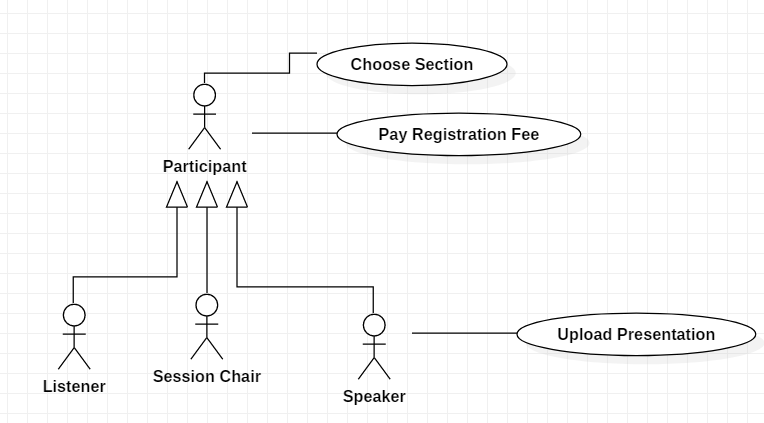


Figure 8: Second Use Case Diagram, Part 2

These diagrams were used as a guideline for the next steps, playing a major role in our understanding of the requirements. By modeling the requirements in Use Case diagrams, we managed to identify the main entities in our application, as well as their roles and options regarding what they might or might not be allowed to do.

**Class Modeling**

At this stage our team designed the model for the main classes from our application, using support materials from the seminars and the Use Case diagrams previously mentioned.

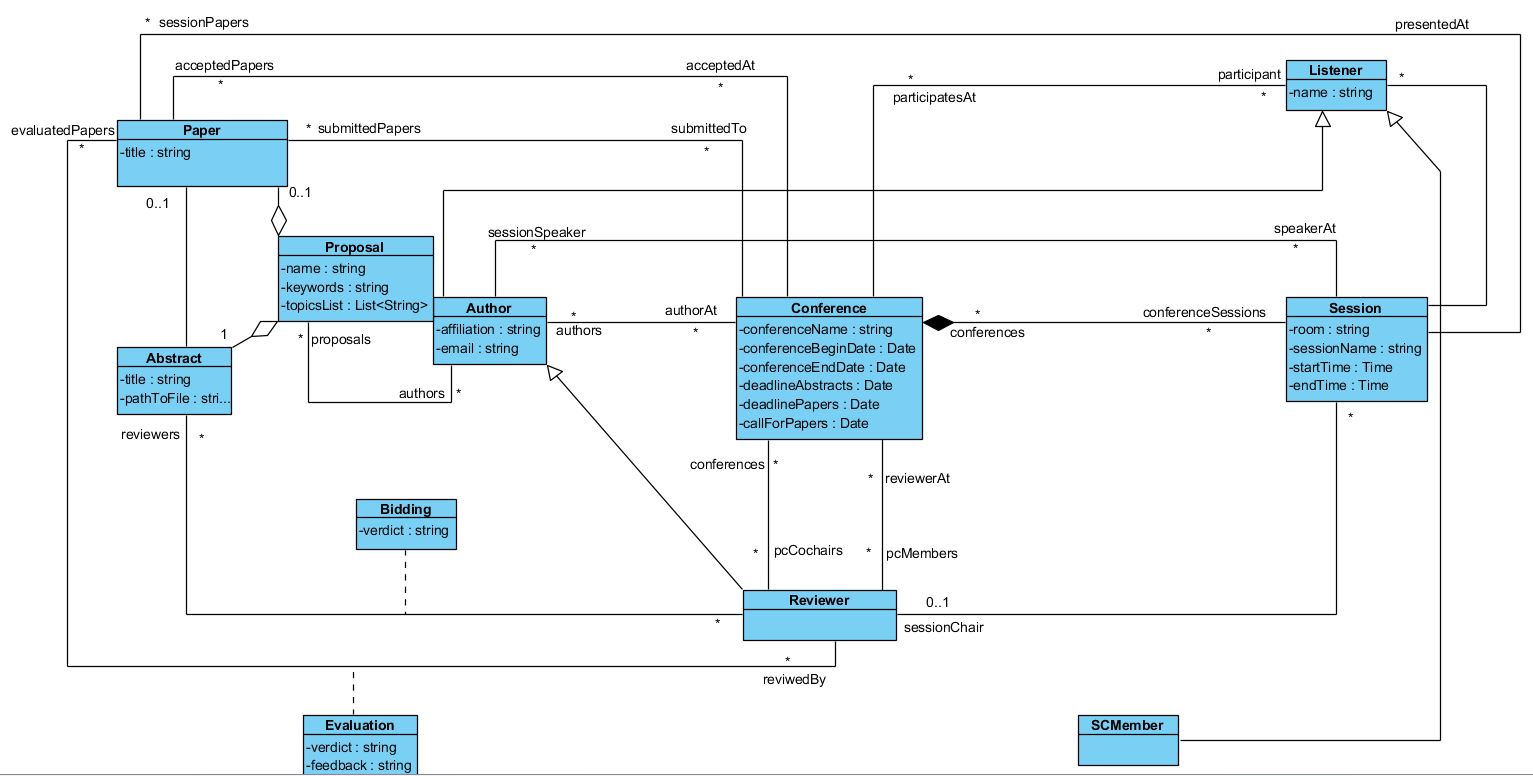


Figure 9: Class Diagram, version 1

However, when we started implementing the application we quickly realized that, due to the constraints imposed by the framework we used for managing the association between our entities and the database, namely Hibernate, we had to follow another approach to design the diagram.

These are the reasons why we chose to re-design the classes:

1. We encountered a problem implementing the many-to-many relations, as Hibernate does not support many-to-many relations with additional fields using annotations only, namely the @ManyToMany annotation. Thus, we had to model these relations by adding an associated class.
2. Working with the @ManyToMany annotation for relations which did not have any additional fields gives less control over managing the entities and by modeling every many-to-many relation to have an associated class we ended up with more classes than we expected, which was hard to maintain as we developed the application.

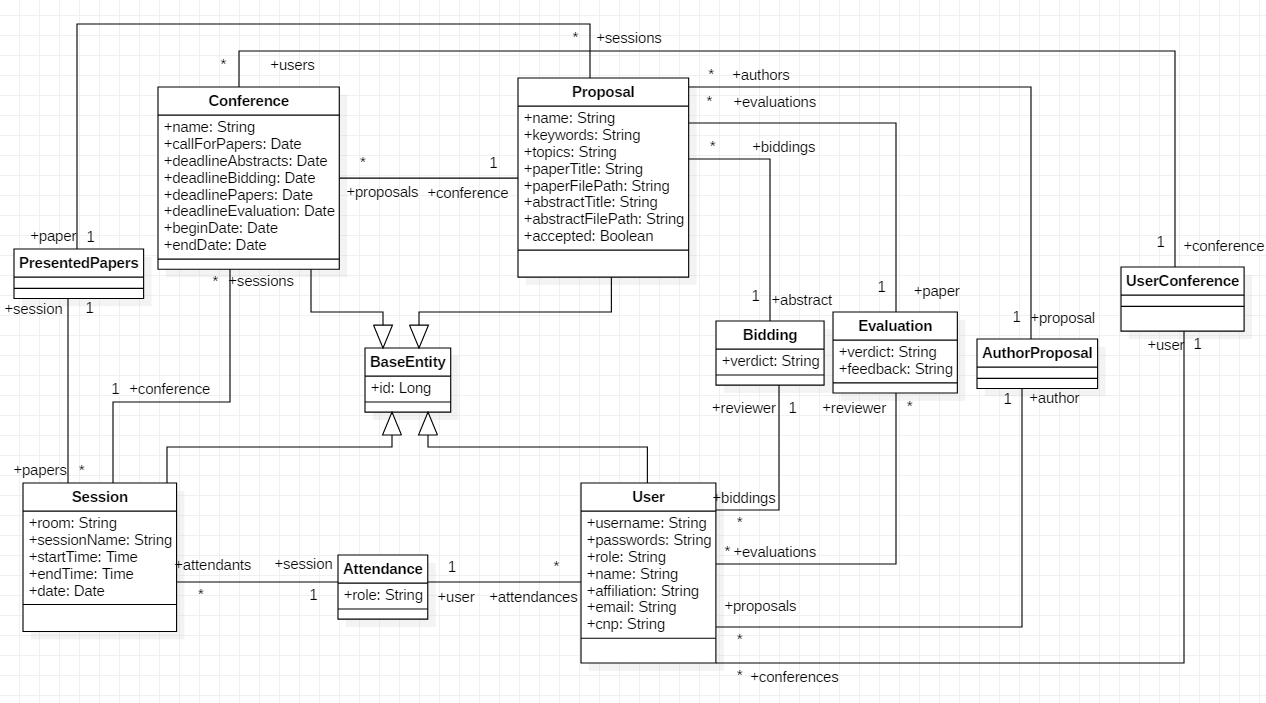


Figure 10: Class Diagram, version 2

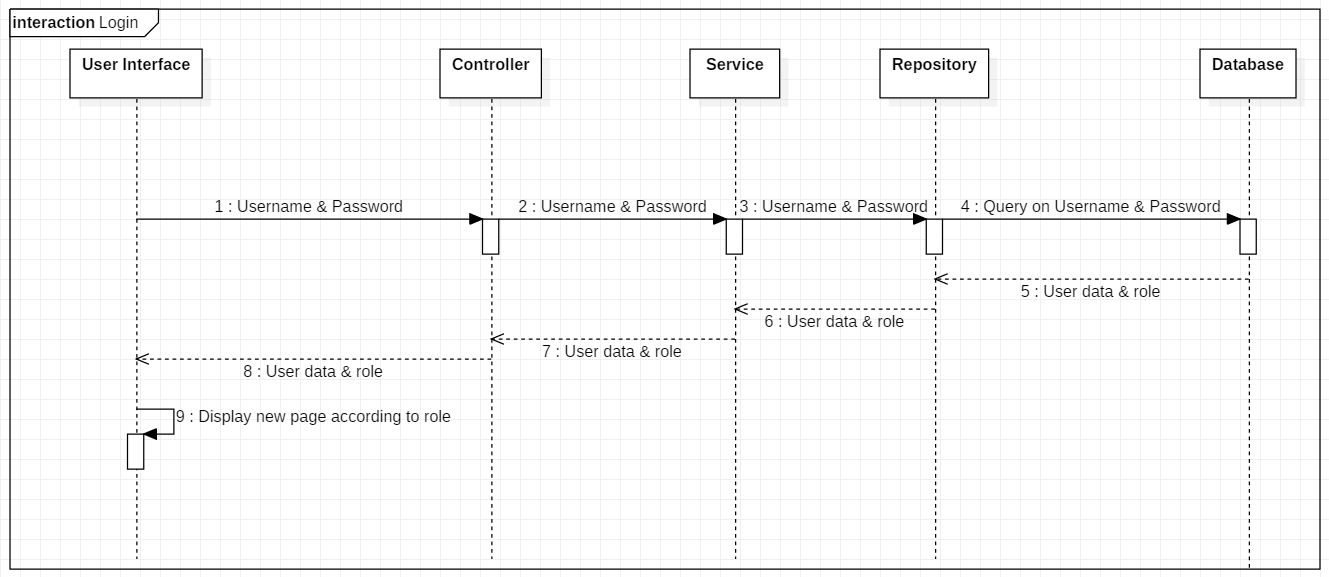
Hence, we chose to reduce the number of classes by adding “roles” to the users. More precisely, each and every Actor from the Use Case diagram now represented a so-called “role”, which dictated what a User could or could not do. By adding this, we minimized the numbers of classes needed, as every role will be shows a specific user interface when logged, which will allow the user to perform the associated actions.

Furthermore, we chose to combine the classes “Paper” and “Abstract” into a single class “Proposal”, considering that the reviewers will only have access through the user interface to the abstract when bidding and to the papers when evaluating. Lastly, we now associate the role a user has during a session, namely session chair, listener or speaker, in the associated class between User and Session.

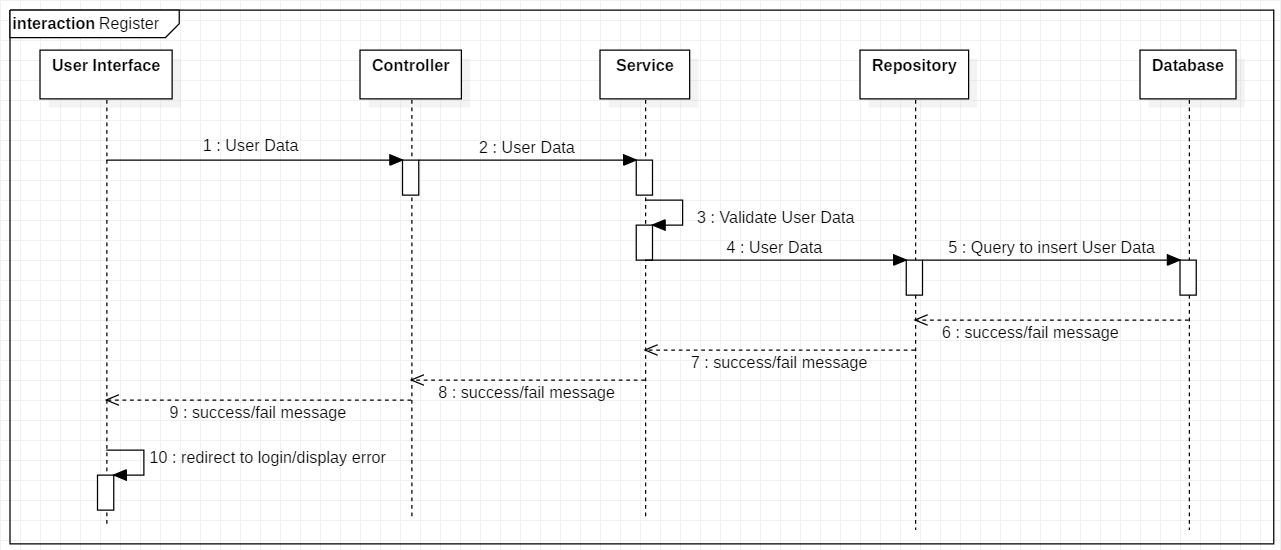
**Sequence Diagrams**

In order to model the sequence diagrams we used StarUML. These are the use cases we illustrated:

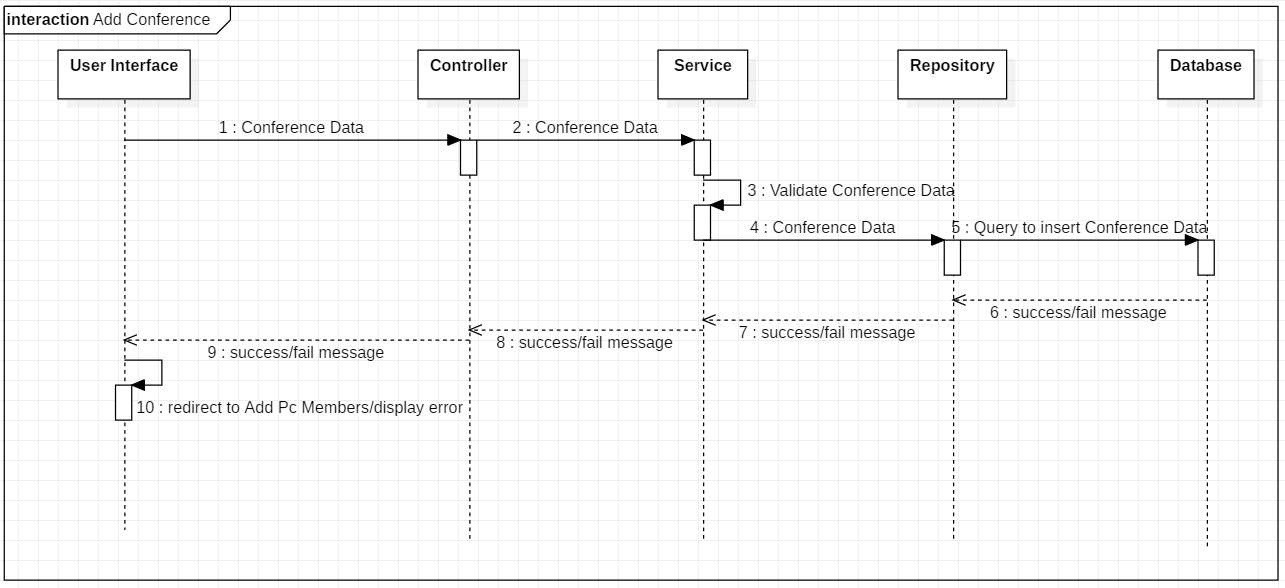
1. Login



1. Register



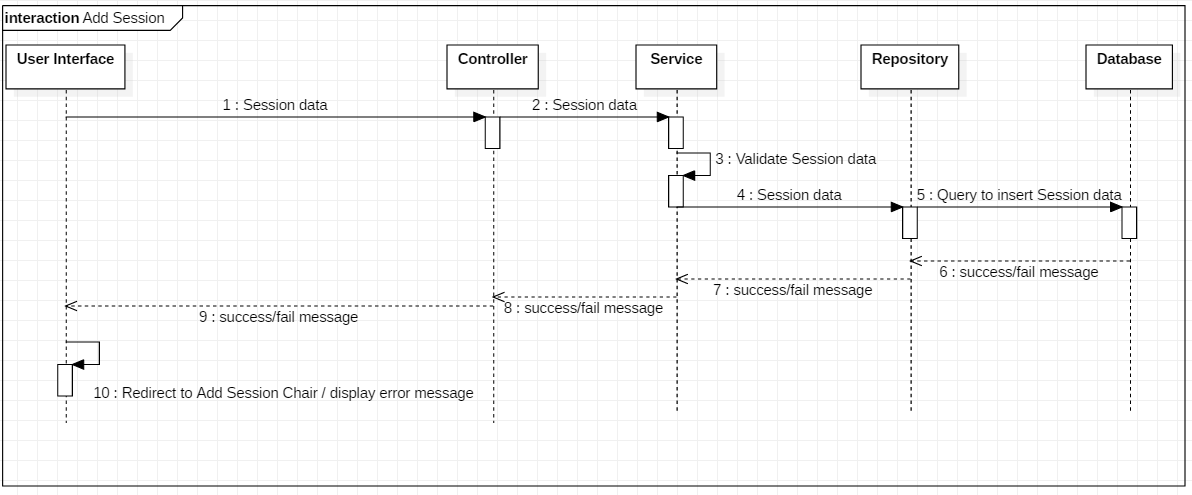
1. Add conference



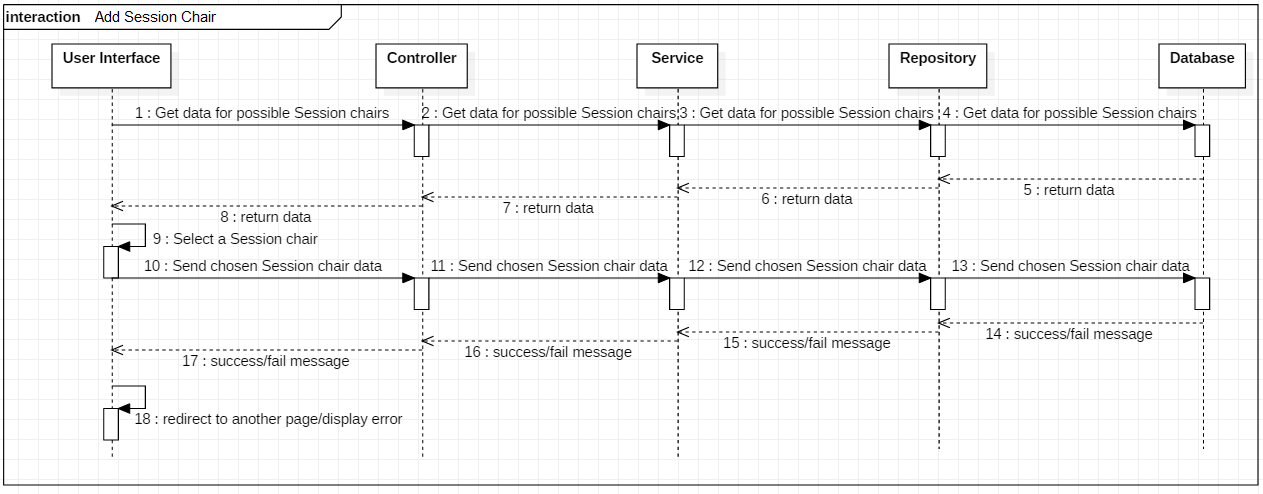
1. Add PC Member



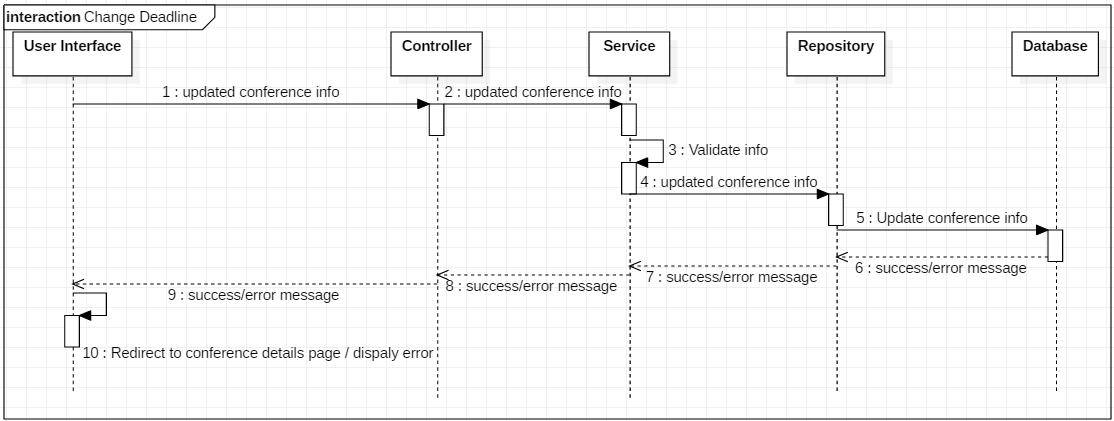
1. Add session



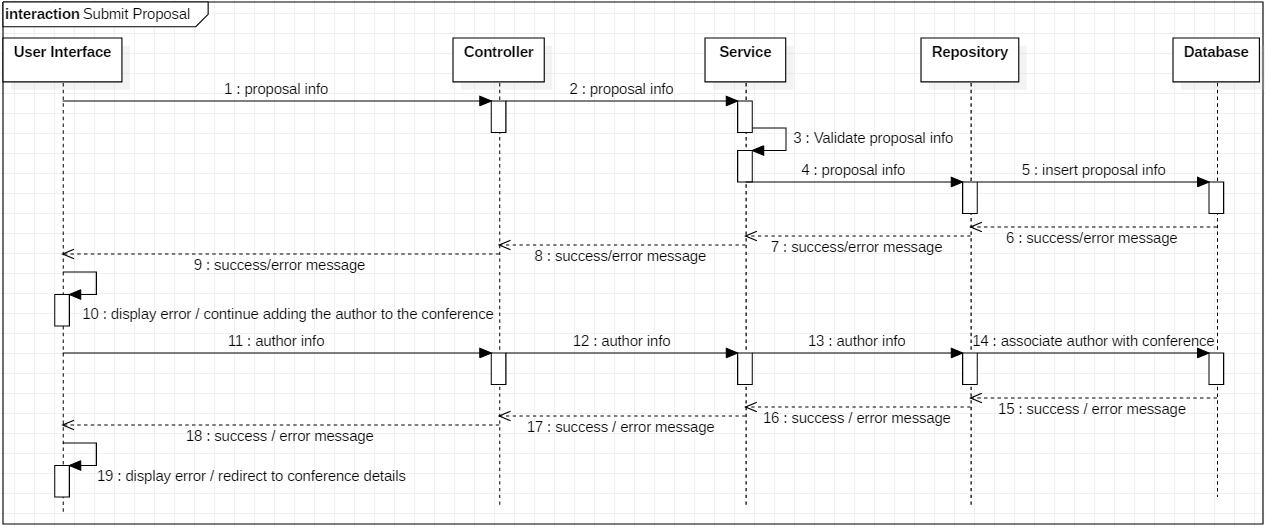
1. Add Session Chair

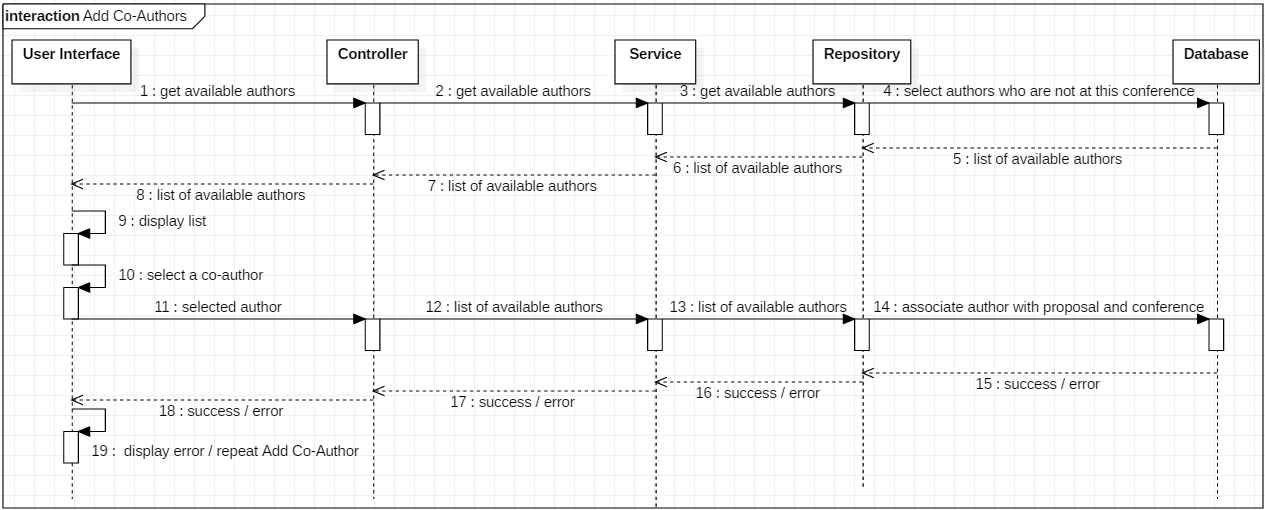


1. Change deadline

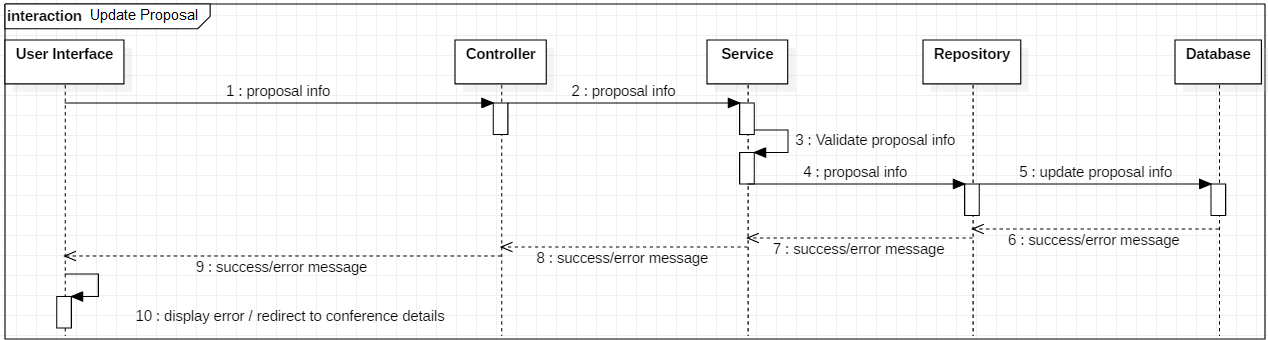


1. Submit proposal & Add Co-Authors

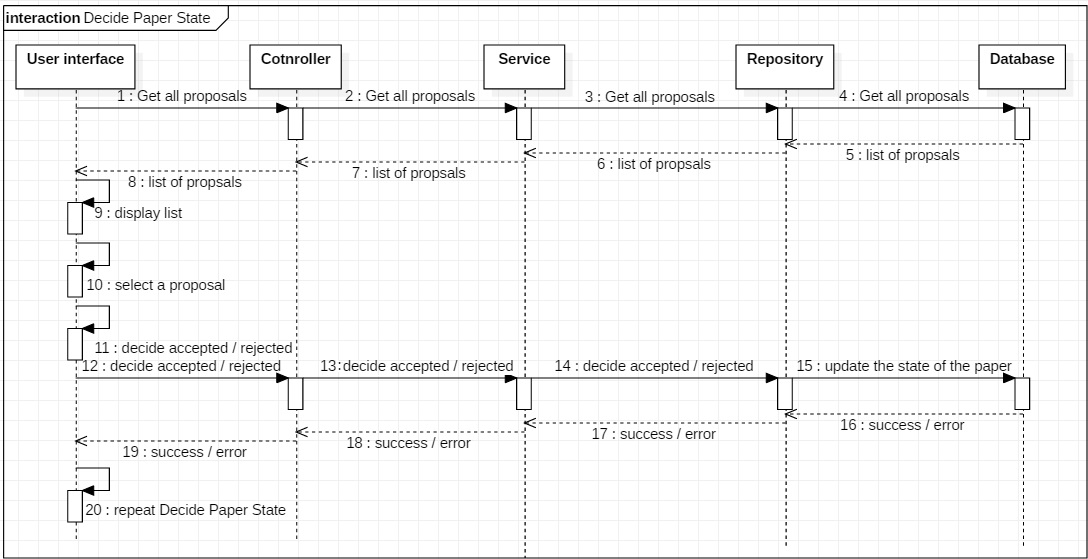




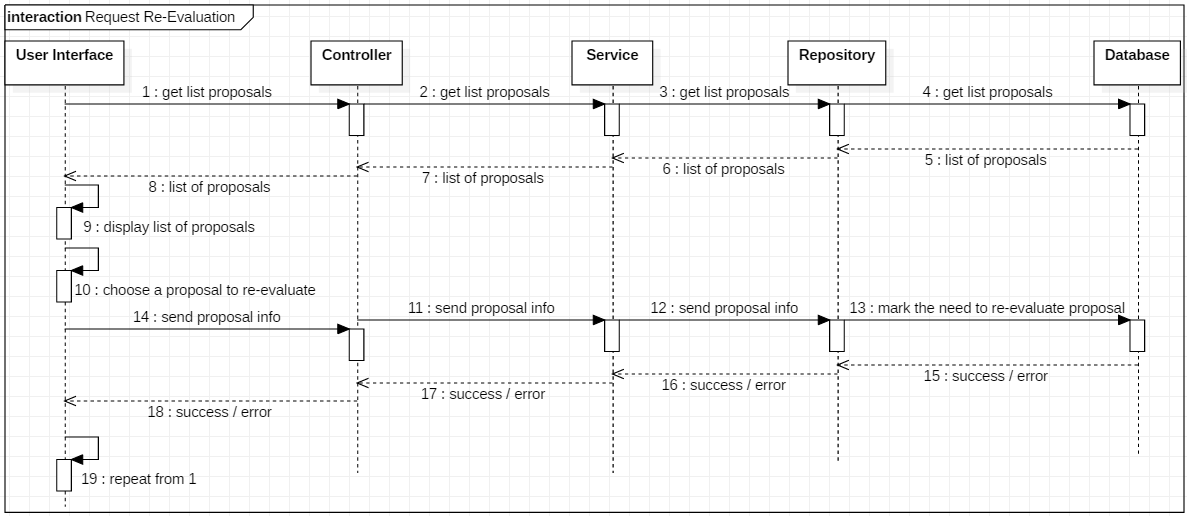
1. Update Proposal



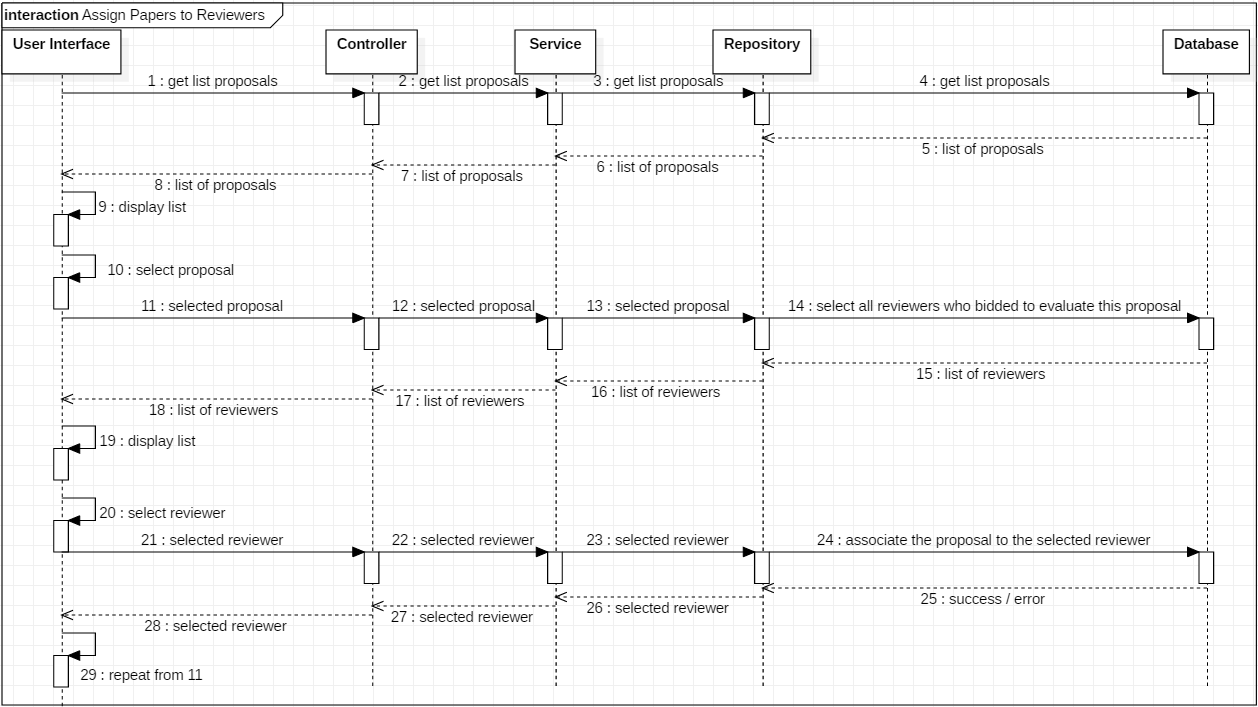
1. Decide Paper state



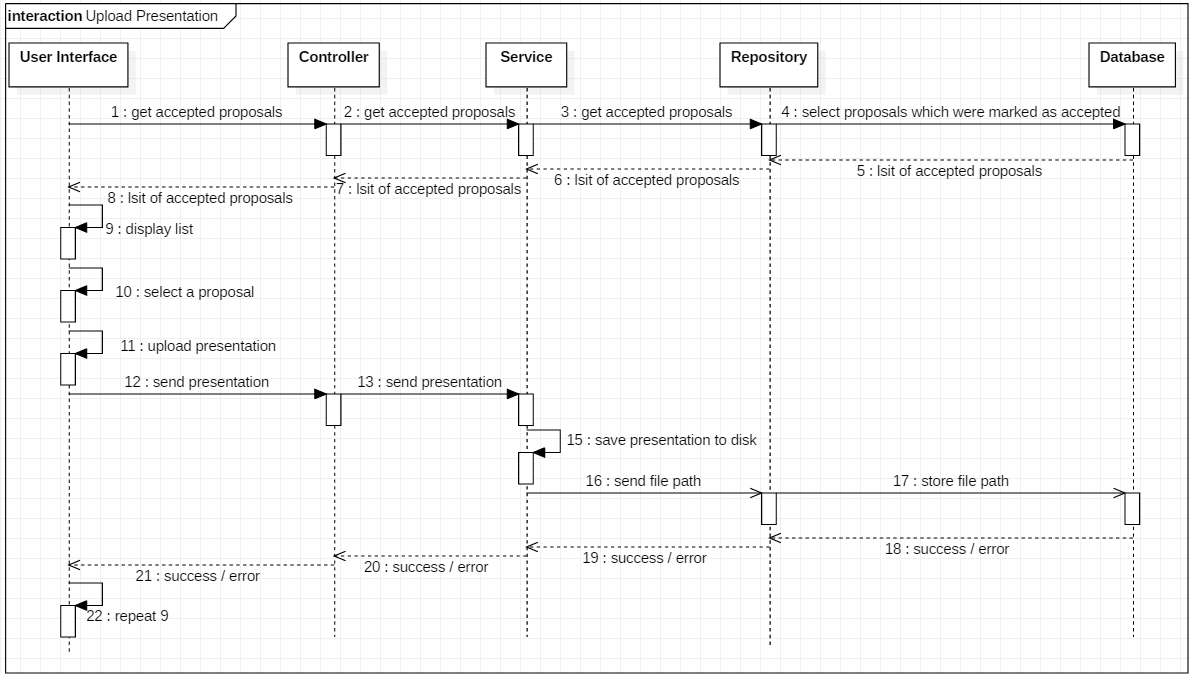
1. Ask for re-evaluation



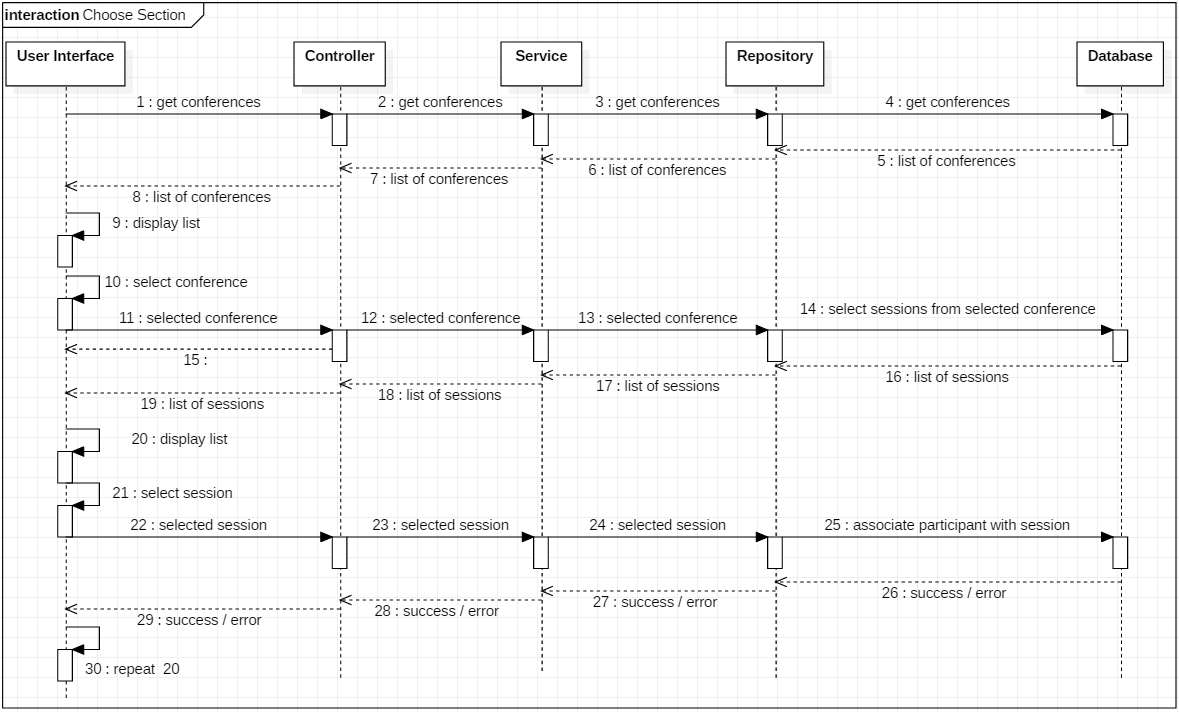
1. Assign Papers to Reviewer



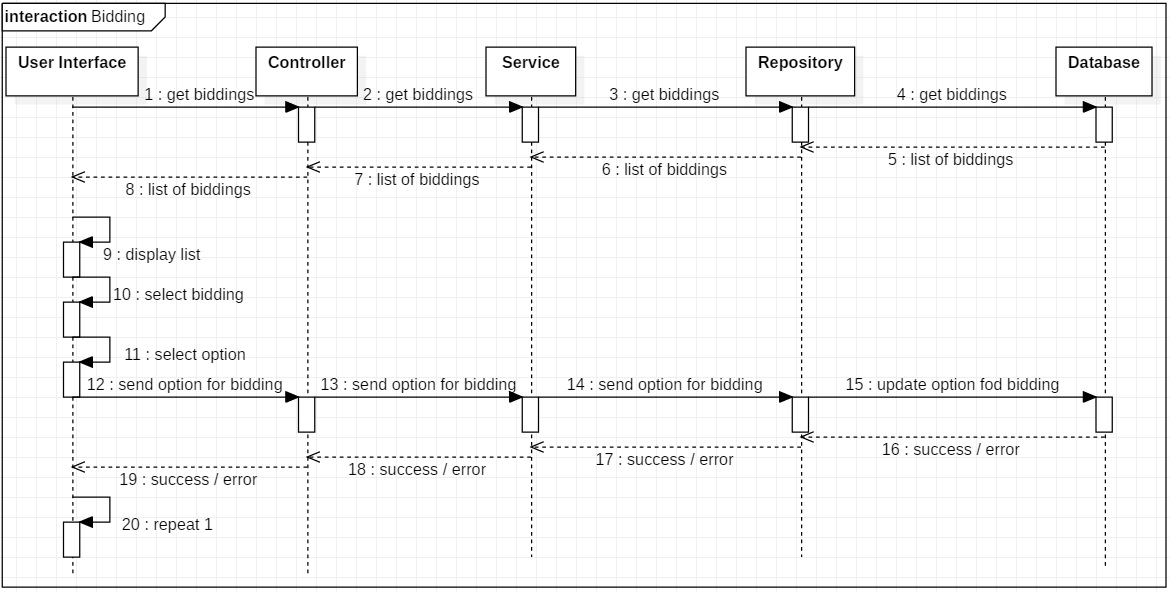
1. Upload presentation



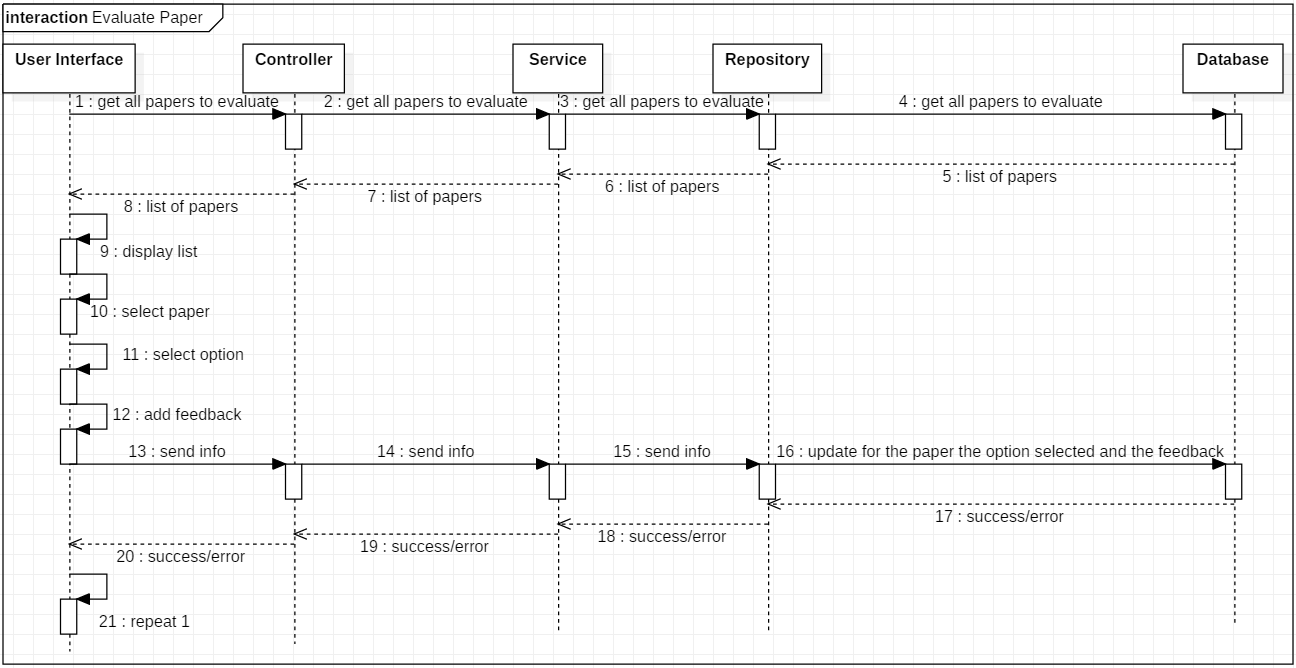
1. Choose section



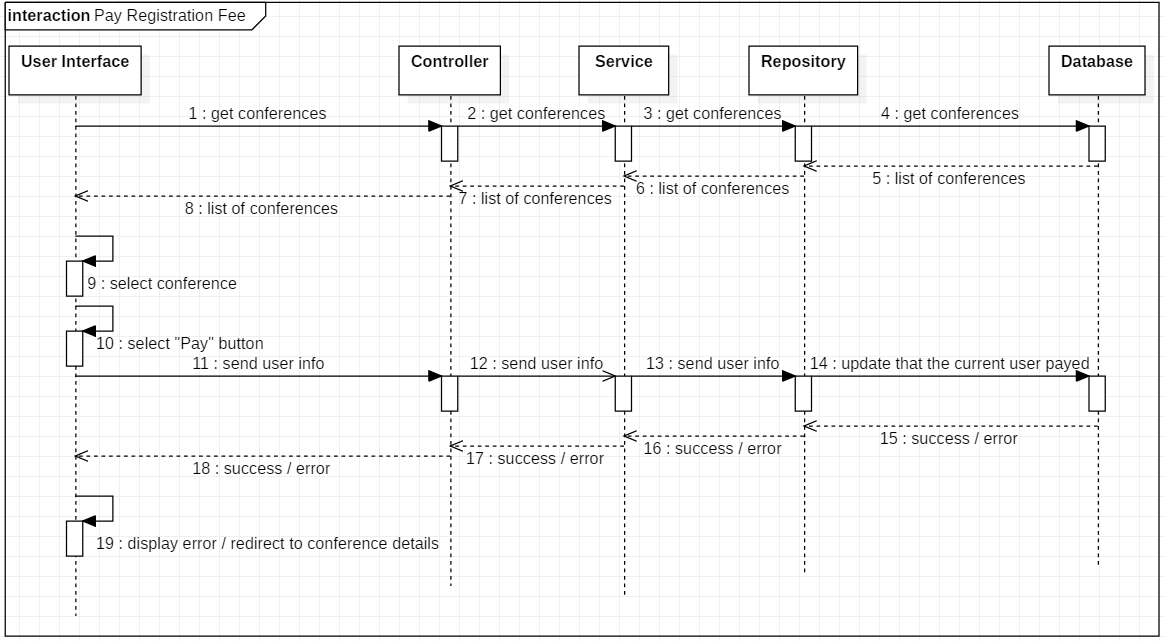
1. Bidding



1. Evaluate Paper



1. Pay Registration Fee



Scheduling

To schedule the development of our application, we split the requirements into tasks, which in turn had subtasks. To keep track of our progress, we used **Trello,** which is a web-based list-making application. Moreover, we were able to use this platform to upload the materials we worked on, such as the diagrams. To communicate and fix problems, we established a **Whatsapp** group in which we helped each other fix bugs, discuss about implementation options etc.

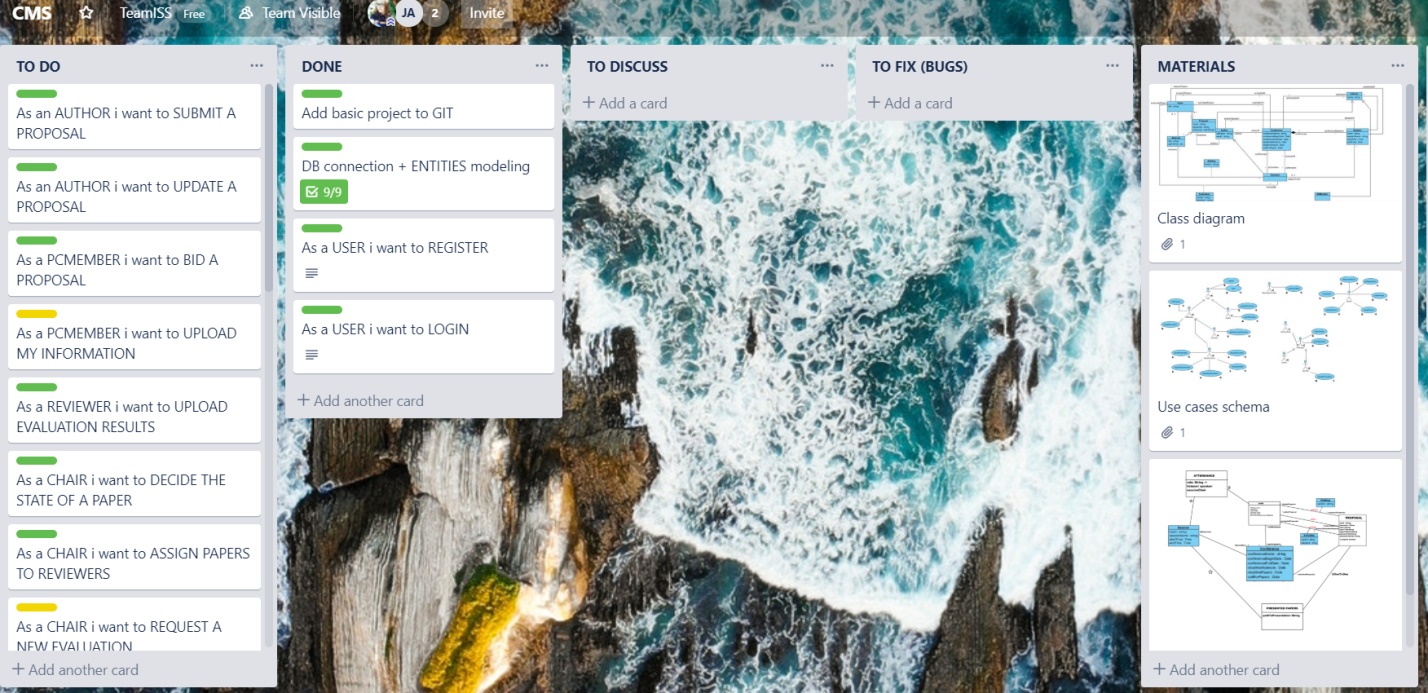


Figure 11: Screenshot of our task division in the early stage of the development

Implementation

**Main layout and Project Diagrams**

The project is structured in multiple packages, each one having a suggestive name.

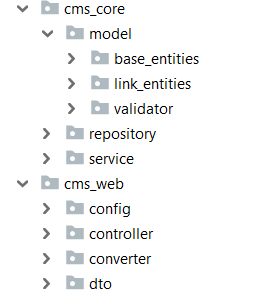


Figure 12: Main structure of the project

Each package contains a set of classes which implement a concept and a set of functionalities.

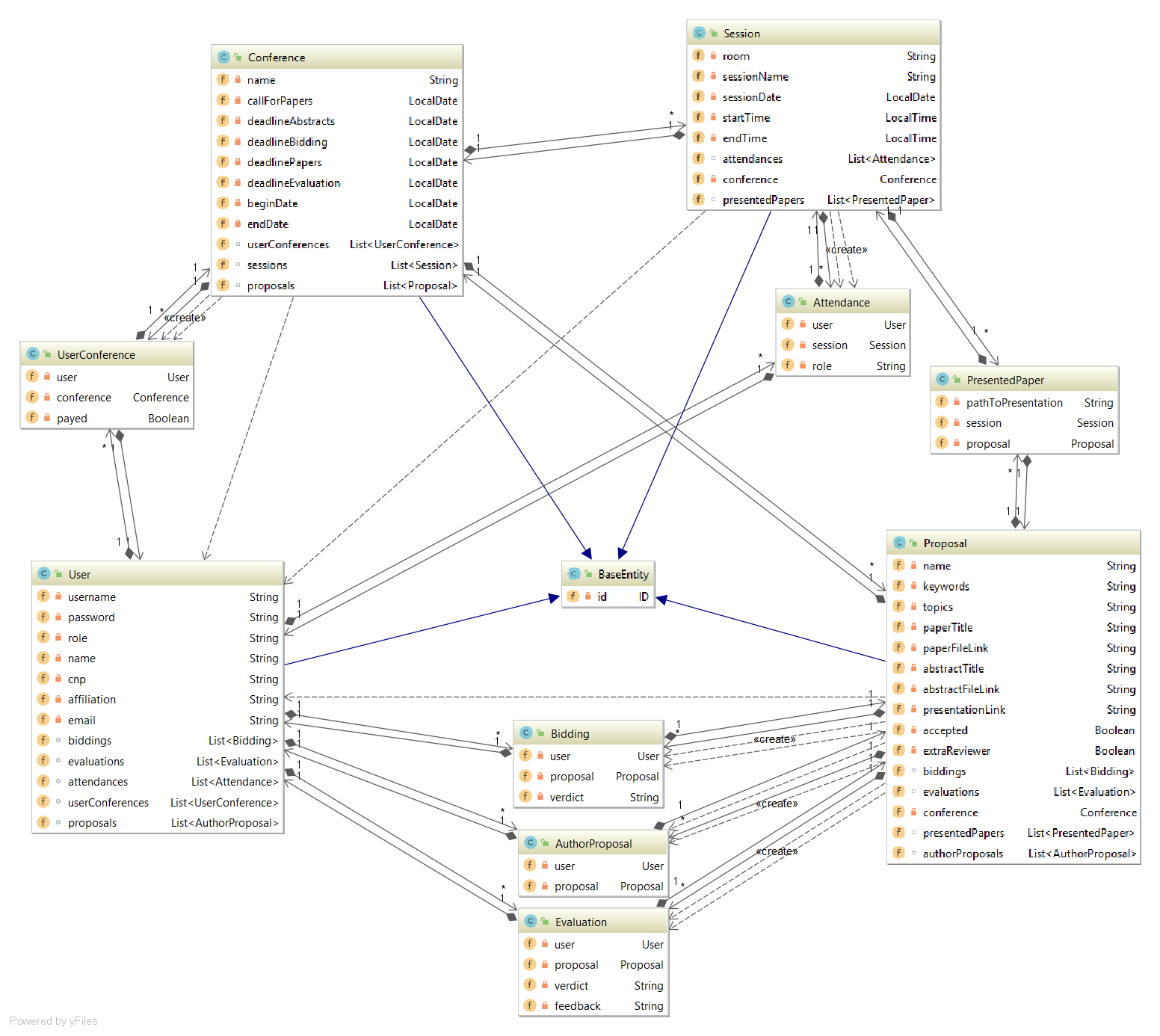


Figure 13: Class Diagram of the main entities

The classes illustrated in the diagram above are a direct implementation of the class model designed during the modeling phase. Their sole purpose is to illustrate the relations between our main entities and to store their data.



Figure 14: Database Schema

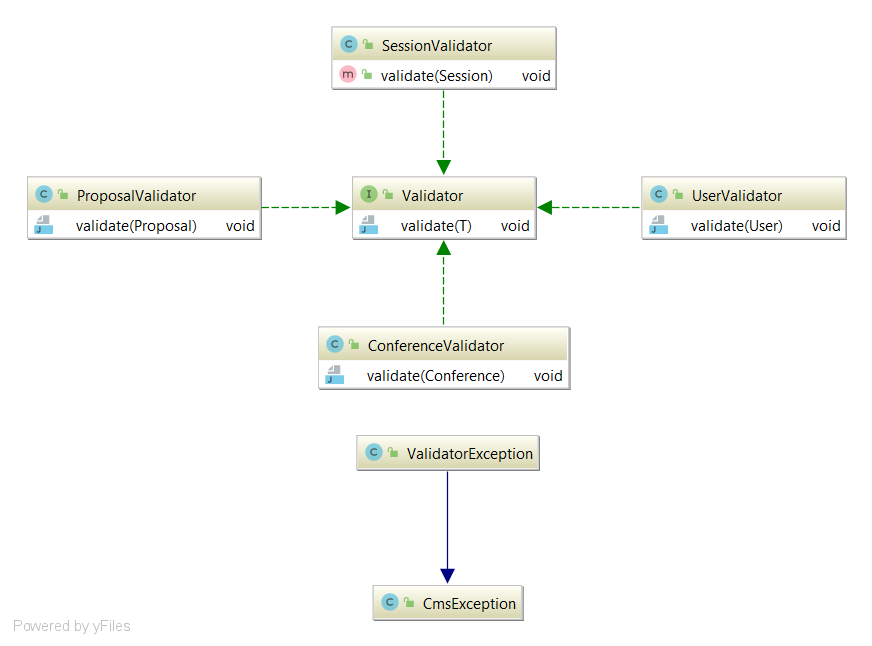


Figure 15: Class Diagram for Validators

The Validator classes serve as a safety net for faulty user input. As suggested by their names, each one of them checks if the data for a specific entity is valid. When the user input is not valid, an exception is thrown.

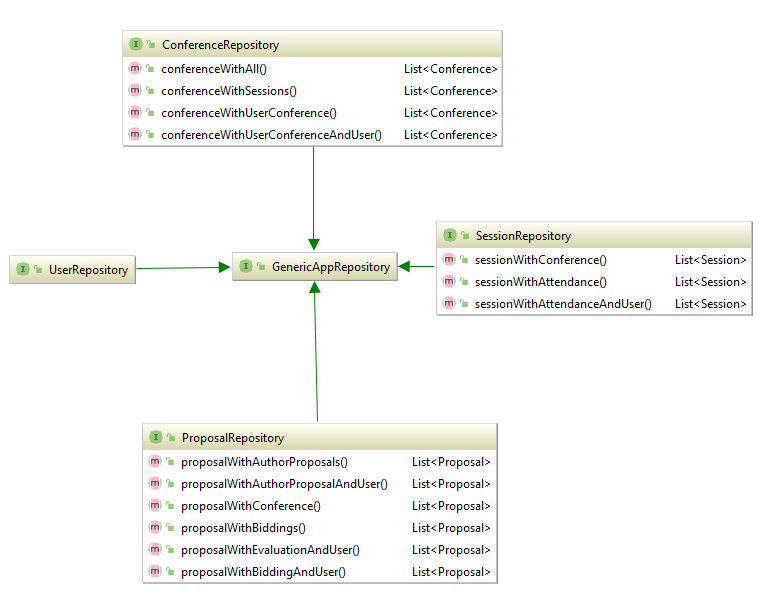


Figure 16: Class diagram for Repositories

The Repository classes serve as the bridge between the database and the rest of the application. Their purpose is to bring data directly from the database and return their corresponding object(s).



Figure 17: Class Diagram for Service

The Service class is where the main business logic lays. Its role is to get data from the Controller and perform operations on/with it. This is where the Validator classes are called in order to check the data for mistakes, and then the objects get passed down to the Repository classes to be persisted in the database.

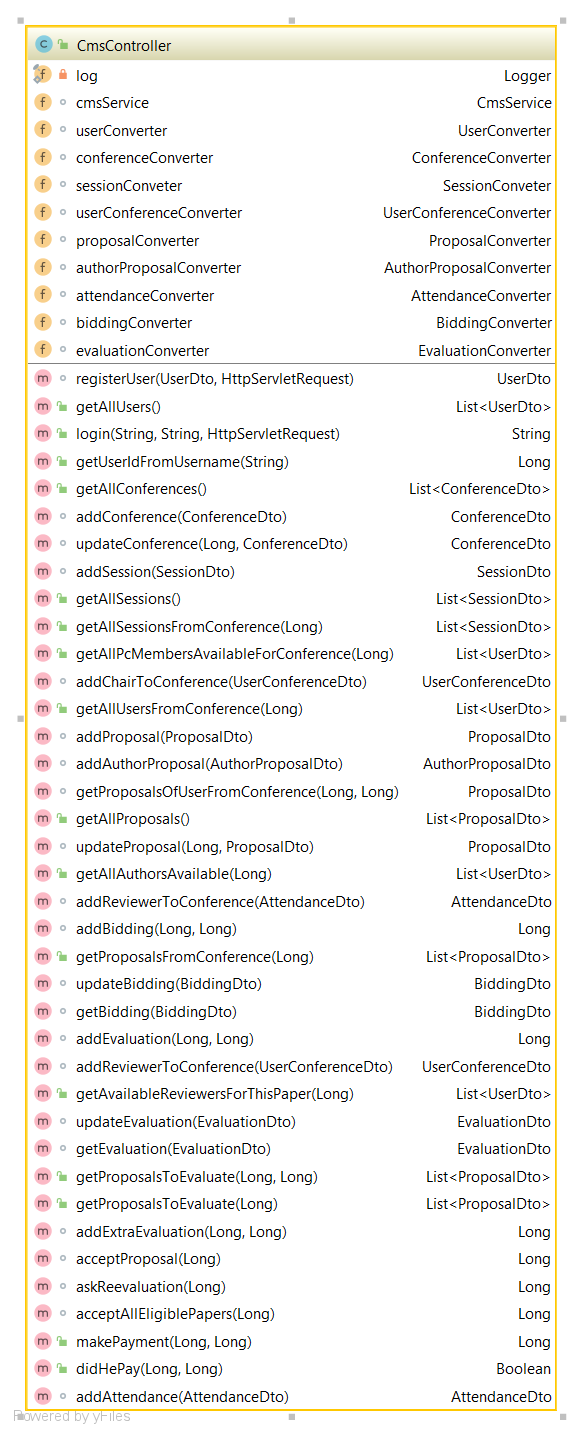
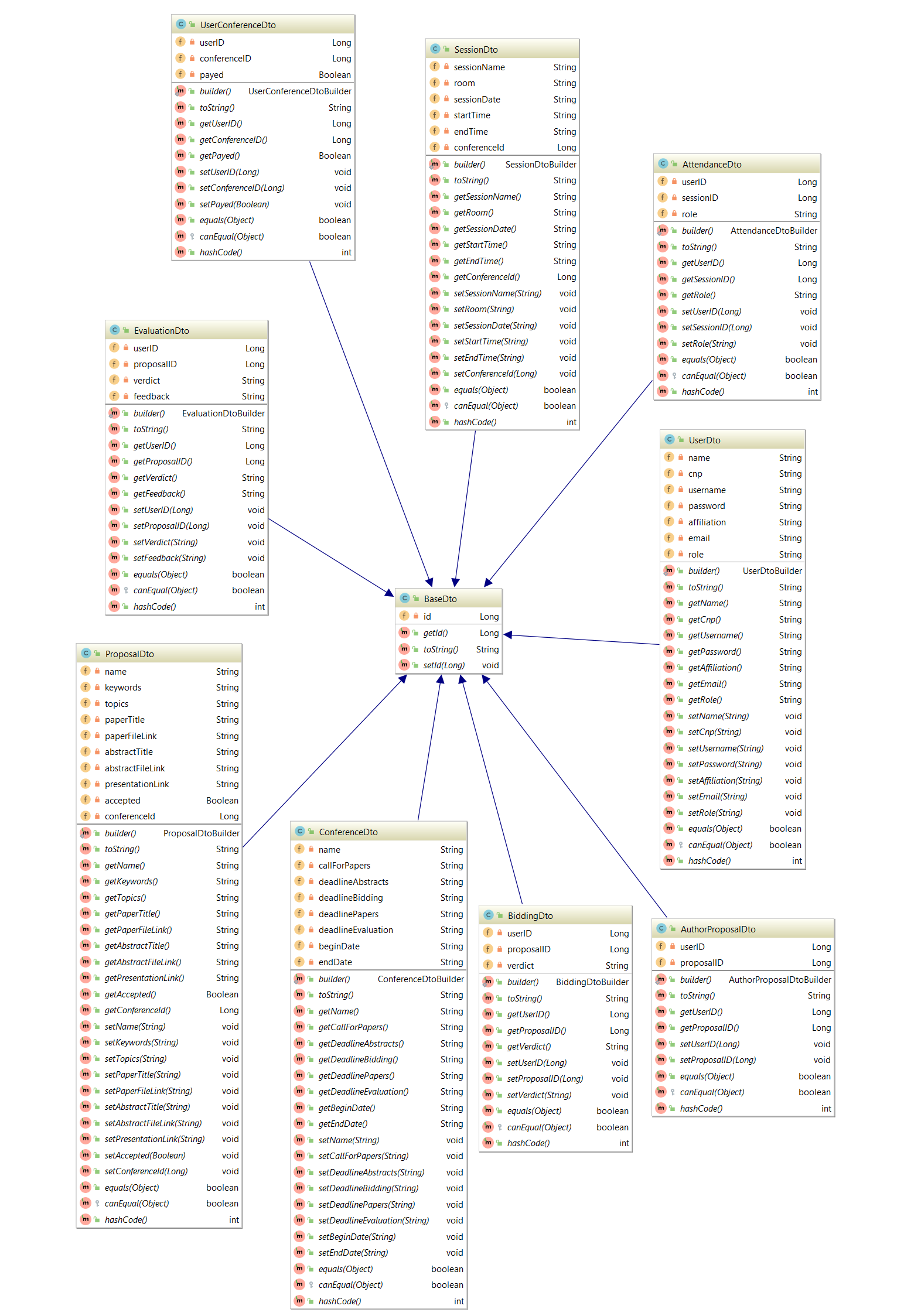


Figure 18: Class Diagram for Controller

The Controller class is the bridge between the user interface (web page) and the Service. It has access to both the Service and the Dtos.



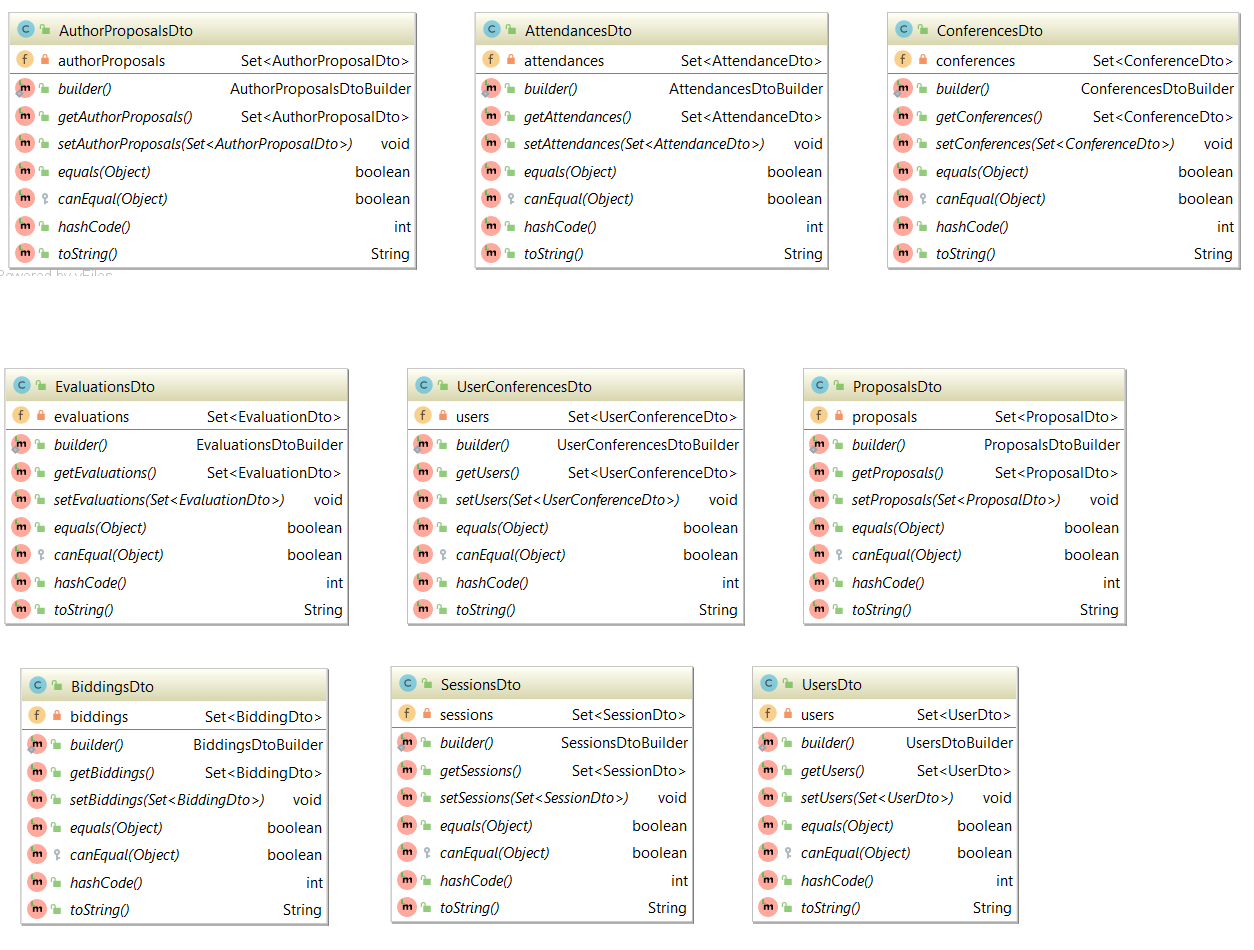


Figure 19: Class Diagram for dtos

The Dto classes are the objects which get sent to the Frontend and back, and they only contain the minimum amount of information that the used is allowed to see.

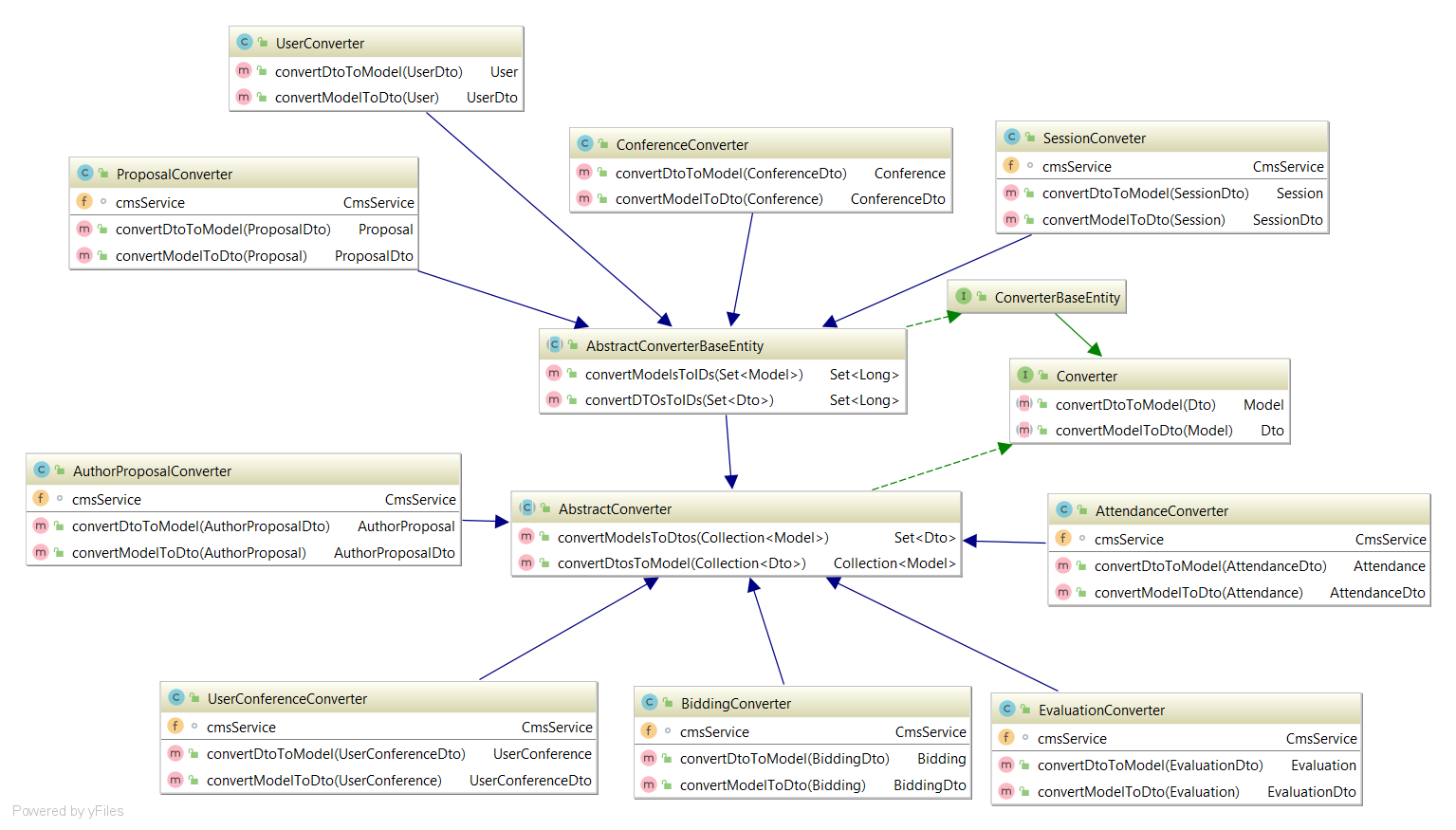


Figure 20: Class Diagram for Converters

The Converter classes, as the name suggests, convert Dtos to their corresponding entities and the other way around.

**Testing**

To test out application we used the **Waterfall Model**. The Waterfall Model is a relatively linear sequential design approach for certain areas of engineering design. As a result of our previous modeling of the application, we had enough materials to support the decision to use this model, as it allowed us to start writing quality code as soon as possible.

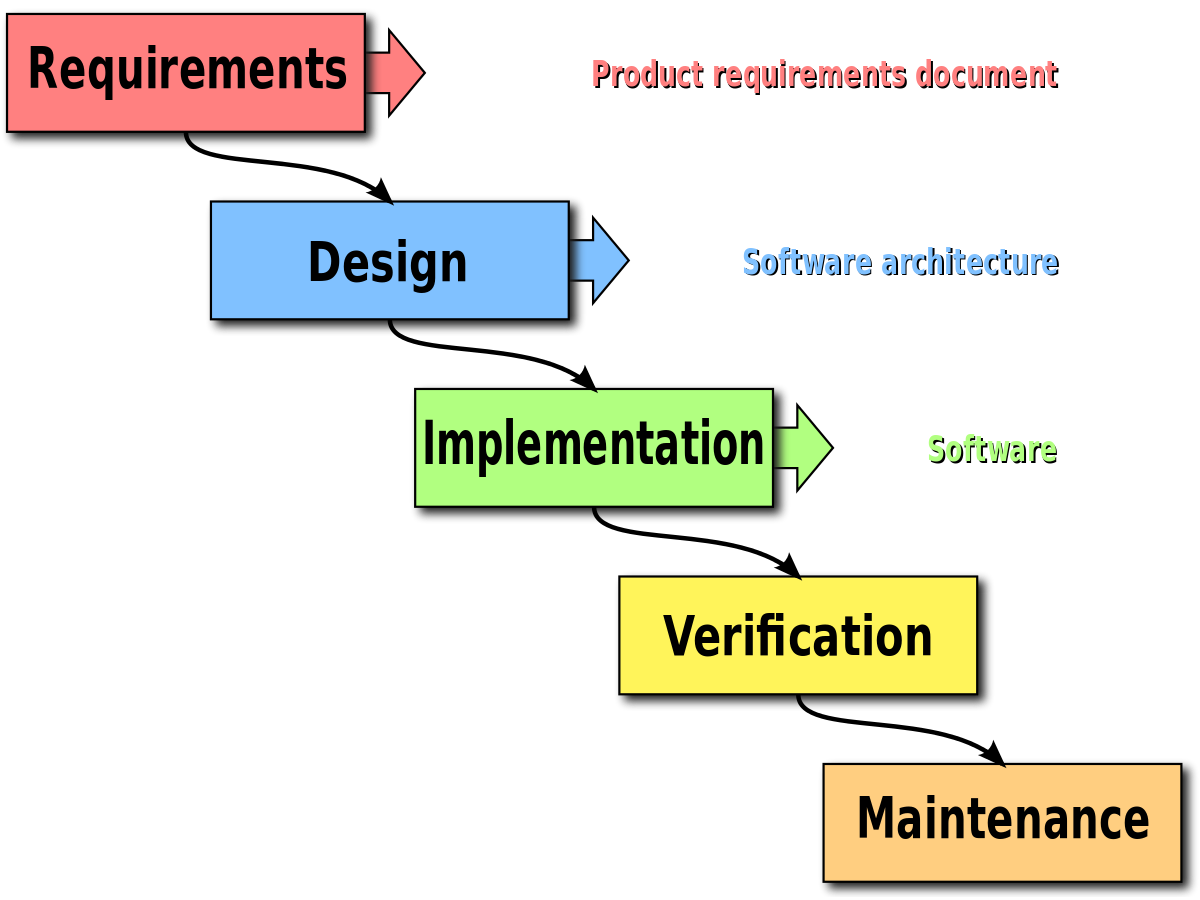


Figure21: Visual representation of the testing model

For our application we decided to have two types of testing: **unit** and **integration**.

**Unit testing** is a level of software testing where individual components of software are being tested. The goal is to insure that each unit of software, which is the smallest testable part of software, performs as desired.

**Integration testing** is a level of software testing where individual units are combined and tested as a group. The goal is to expose faults in the interaction between integrated components.

Even if we did not cover **System testing**, we tested the functionalities from the user’s perspective, namely through the pages shown in the Web Browser and their respective calls to the Backend.

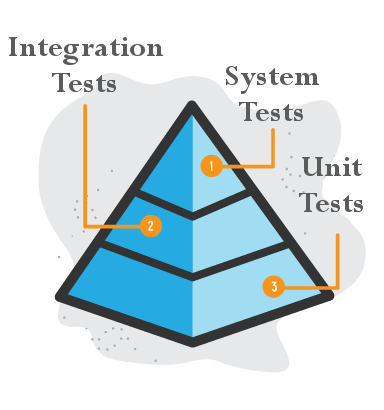


Figure 22: Representation of different types of tests

To achieve this, we used Junit tests in combination with Spring Boot. In order to maintain the data in the database unchanged, we used mock objects, which are configured to perform a certain behavior during a test.

Final thoughts

During the development of this project we believe we have acquired new skills and we have further improved existing ones.

One of the goals of this project was to understand and utilize the knowledge that was passed on to us from lectures and seminars. By working in a team and going through all the stages of software development, from modeling the requirements to implementing the application, and finally testing the quality of our project, we had an opportunity to understand the value of well-thought-out software.

We believe that this project helped us grow as developers, for which we are deeply grateful.