Department of Computer Science
Technical University of Cluj-Napoca

Debate Application

Software Engineering, Laboratory activity

GitHub Repository: DebateApp
Developed By : Beáta Keresztes and Borbála Fazakas
Group: 30432

Teaching Assistant: Eneia Nicolae Todoran

Contents

Chapter 1

Abstract

The purpose of this document is to present the goal and the main aspects of the implementation of our DebateApp. The DebateApp is a web application developed in the context of the Software Engineering course, that aims to provide people a safe online environment for debates: it is an application that encourages an open and respectful discussion between people of different opinions, with the hope of bringing them closer together. To implement this application, we relied mainly on the Spring Web MVC Framework, that defines the applications' structure and the communication between the client and the server, on the Daily API for integrating a video conference and on WebSockets for making real-time communication between the participants of a debate possible.

Chapter 2

Problem statement

The goal of our application is to provide an online environment for coordinated debates.

With the recent evolution of political and social events, it is more and more clear that there is a growing need for well-structured, respectful debates with rational reasoning and arguments supported by evidence. Although philosophers have established several formats and rules for such debates, they have only been followed by debate contest participants in formal environments, but not in everyday life.

There is an ongoing debate, for example, about online and onsite education, triggered by the evolution of the pandemic. In the lack of proper communication between the two parties, the people tend to have extremist opinions. They usually discuss their ideas with others sharing their opinion, and simply ignore the arguments of the other team, thus amplifying their own voice, and making the gap between the two parties impossible to bridge.

We hope that our application will bring people of different opinions closer to each other, provide them a safe space for sharing their perspective on different notions, in a respectful manner, in which evidence has a major weight in the discussion and changing one's opinion after listening to the arguments of the opponent's team is not a shame, but rather a proof of thinking rationally and having an open mind.

Chapter 3

Problem Analysis. The Debate Protocol

In order to achieve the above mentioned goals, we decided to adapt the structure of the policy debates, which is often used at debating contests. You can read more about it on Wikipedia (here or here) for example, but you'll also find a short summary of the rules adopted to our application in the next section.

The main elements of a debate are:

- **The Topic**: an affirmative statement, defined by the judge.
- **The Teams**: participants join the **Affirmative or the Negative team** based on their personal opinion before the beginning of the debate.
- **The Judge**: the coordinator of a debate session.

The judge is the one who defines all the parameters of a debate session, in the format of a **debate template**, including:

- the title and the topic
- the statement
- the length of each debate phase, as described in the following section
- any resources recommended for the team members to analyze in the preparation session

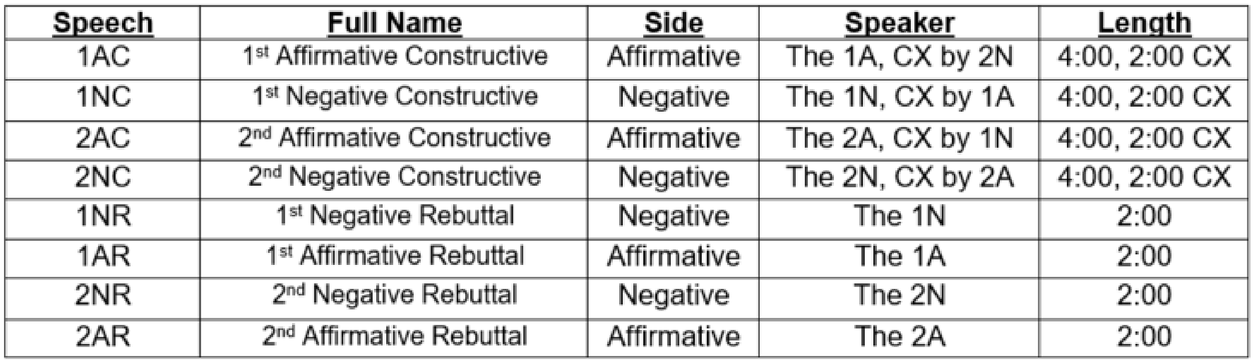
Additionally, the judge can decide when to allow participants to join a debate session and when to activate the debate, meaning that players are no longer allowed to join, and the debate session is kicked off with the preparation phase.

The phases of the debate are the following:

1. **The Preparation Session**: after at least 2-2 users joined each team, the judge can activate the debate, which starts with the preparation session.
    - team members can only communicate between each other

- team members are encouraged to discuss the topic:
  - express arguments, supporting their opinion
  - collect evidence for their arguments
  - anticipate and discuss arguments of the opponent team
- the judge can observe and intervene in the preparation of both teams

2. **The Deputy Selection**: at the end of the preparation session, both teams must elect a **Deputy 1 and a Deputy 2**
   - the deputies of the Affirmative Team are: **Affirmative1 (1A) and Affirmative2 (2A)**
   - the deputies of the Negative Team are: **Negative1 (1N) and Negative2 (2N)**
   - the deputies are elected inside a team through voting, one after the other. The two deputies of the team cannot be the same person.
3. **The Battle**: after the deputies are elected, the actual debate begins, which consists of 8 speeches and 4 discussions delivered by the 4 deputies:
   - **4 constructive speeches (AC, NC)**, which allows deputies to bring in new arguments
   - **4 rebuttal speeches (AR, NR)**, which allow deputies to defend their previous arguments and formulate their final conclusions
   - **4 cross examination (CX) sessions**, 1 after each constructive speech, to refute the opponent's previously presented arguments, ask questions, etc.
   - the order of the speeches is summarized in the following figure:



Speech	Full Name	Side	Speaker	Length
1AC	1st Affirmative Constructive	Affirmative	The 1A, CX by 2N	4:00, 2:00 CX
1NC	1st Negative Constructive	Negative	The 1N, CX by 1A	4:00, 2:00 CX
2AC	2nd Affirmative Constructive	Affirmative	The 2A, CX by 1N	4:00, 2:00 CX
2NC	2nd Negative Constructive	Negative	The 2N, CX by 2A	4:00, 2:00 CX
1NR	1st Negative Rebuttal	Negative	The 1N	2:00
1AR	1st Affirmative Rebuttal	Affirmative	The 1A	2:00
2NR	2nd Negative Rebuttal	Negative	The 2N	2:00
2AR	2nd Affirmative Rebuttal	Affirmative	The 2A	2:00

4. **The Final Voting**: after the battle, the players are given another chance to express their opinion about the debate's statement, after taking into account all the arguments and evidences presented during the battle. It may occur, that someone from the Affirmative or the Negative team now changes their opinion and votes against their initial statement.
5. **The Final Discussion**: finally, when the statistics about the final votes and the changes in the belief of the players is gathered, the players of both teams are given a chance to freely discuss the key learnings and observations related to the battle. Remember, the goal of the debate is not to reach a full agreement, but rather to get to know each other's opinions and have an eye-opening discussion

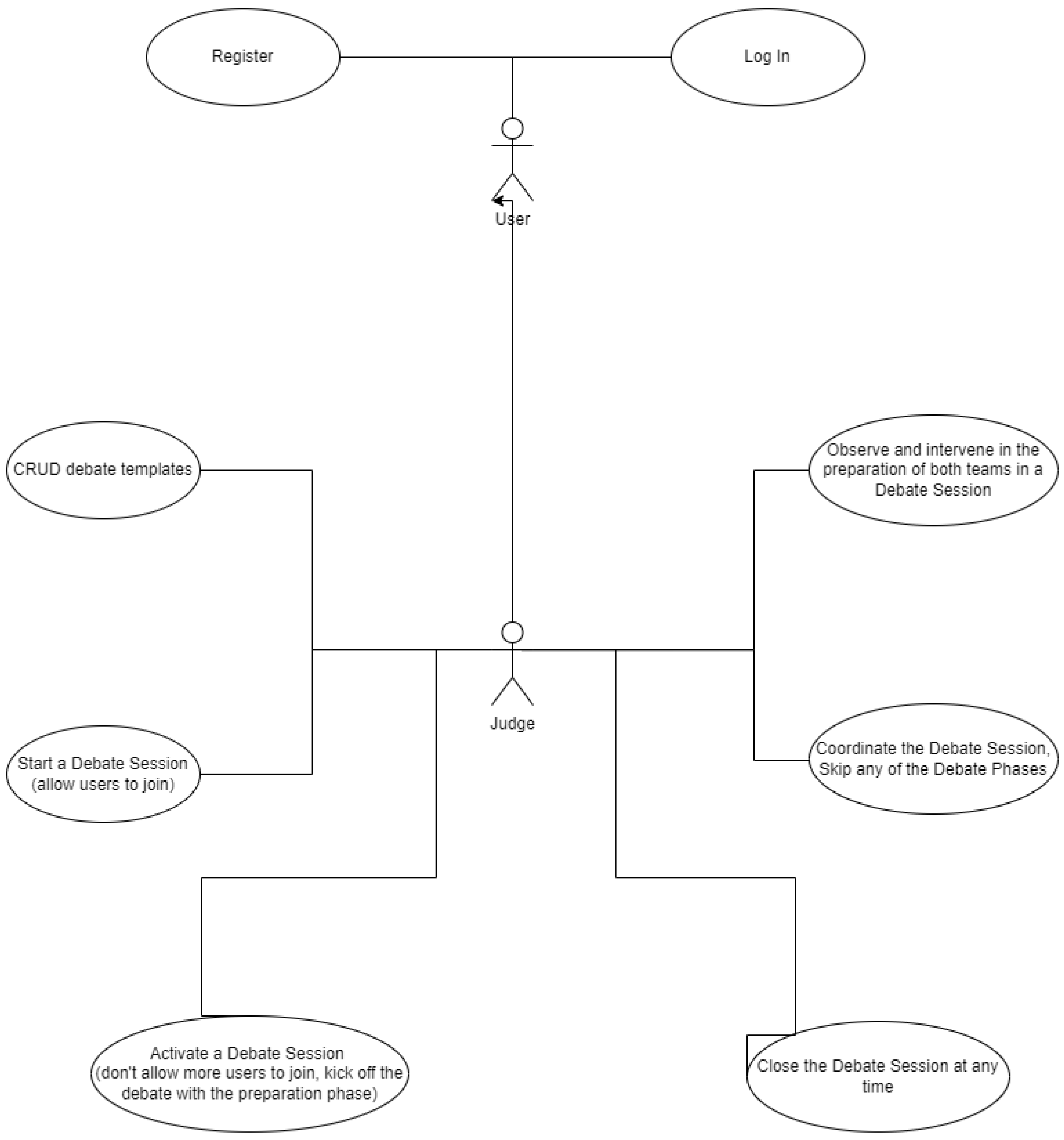
Chapter 4

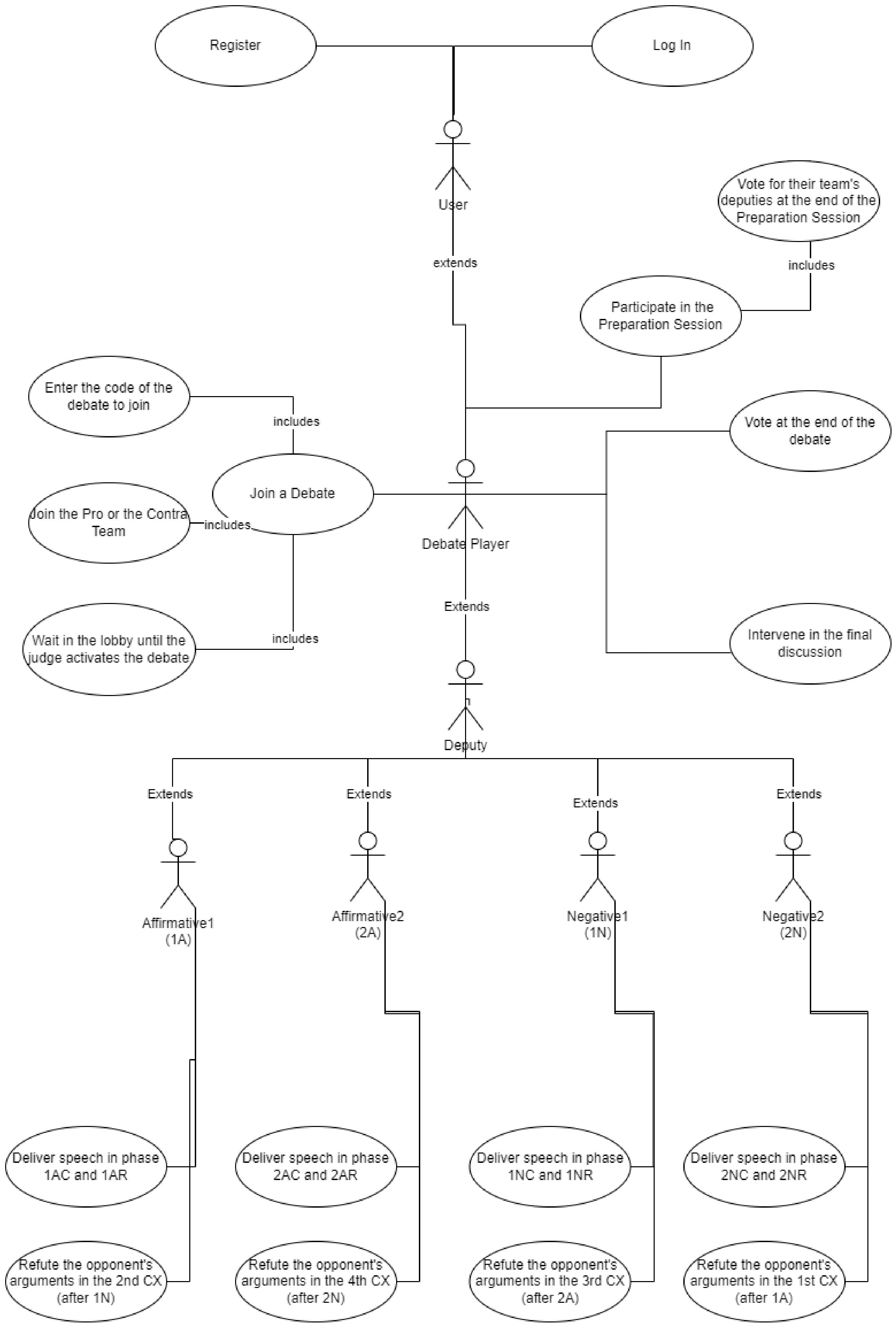
Use cases

The use cases for the judge of a debate and for a player of a debate are presented separately. However, note that the 'judge' and 'player' are only user roles, and **the same user may have different roles in different debates**.

Similarly, a player of a debate may have the role of the Affirmative Deputy 1, Affirmative Deputy 2, Negative Deputy 1, Negative Deputy 2 or may have no role at all. A user who is a player in multiple consecutive debate sessions may have different roles in these debate sessions.

See the Player-Role Pattern section for more information.


Register
Log In
User
CRUD debate templates
Observe and intervene in the preparation of both teams in a Debate Session
Judge
Start a Debate Session (allow users to join)
Coordinate the Debate Session, Skip any of the Debate Phases
Activate a Debate Session (don't allow more users to join, kick off the debate with the preparation phase)
Close the Debate Session at any time


Register
Log In
User
extends
Vote for their team's deputies at the end of the Preparation Session
includes
Participate in the Preparation Session
Enter the code of the debate to join
includes
Join a Debate
Join the Pro or the Contra Team
includes
Vote at the end of the debate
Debate Player
Wait in the lobby until the judge activates the debate
includes
Extends
Intervene in the final discussion
Deputy
Extends
Extends
Extends
Extends
Affirmative1 (1A)
Affirmative2 (2A)
Negative1 (1N)
Negative2 (2N)
Deliver speech in phase 1AC and 1AR
Deliver speech in phase 2AC and 2AR
Deliver speech in phase 1NC and 1NR
Deliver speech in phase 2NC and 2NR
Refute the opponent's arguments in the 2nd CX (after 1N)
Refute the opponent's arguments in the 4th CX (after 2N)
Refute the opponent's arguments in the 3rd CX (after 2A)
Refute the opponent's arguments in the 1st CX (after 1A)

Chapter 5

Technologies used

5.1 Spring Boot and Web MVC

The application is built relying on the Spring Web MVC Framework, a "model view controller" web framework built on top of the Servlet API, which is well suited for developing web applications, guaranteeing the separation of concerns in the model-view-controller design pattern. It contains an embedded Tomcat server for handling HTTP requests.
In Spring MVC, the communication between the server and client sides is realised by the means of controllers, beans annotated with @Controller or @RestController, which handle the incoming HTTP requests. Methods inside the controller are mapped to HTTP by using @RequestMapping, @GetMapping or @PostMapping annotations, depending on the request's nature.

5.1.1 Sping Securiy

Spring Security provides an architecture for realizing authentication and access control, and it is used to handle the user's registration or login into the application.
The authentication is implemented using an AuthenticationProvider instance, through which it can be specified how to handle the user's credentials, for example setting a password encoder. Spring Security in the web tier architecture, having a ui view and a back-end server, is based on Servlet Filters, which realize the connection between the server and client side and they are responsible for propagating the requests to the corresponding servlet to handle it, based on the given URI path.

5.2 Spring WebSockets

The WebSocket protocol realizes a full-duplex, two-way communication between client and server and it makes the application more interactive. It is mainly used to handle certain server side events of which multiple clients should be notified in real-time.

To make using the WebSockets easier, we used the STOMP protocol built on top of WebSockets, that defines how the client and the server interract with each other (how a client connects or subscribes to the WebSocket, ...). On the server side, the Spring WebSockets were used which gracefully handle messaging based on the STOMP protocol. Additionally, on the client side, we relied on SockJS to support also the browsers that do not natively support WebSockets.

For example, sockets are used to implement the timing functionality of the application's states. The application consists of some timed states/phases, which after the end of the pre-defined period would trigger the transition to the next state.
Upon such a state-transitioning event it is desired that all the clients subscribed to the specific websocket will be notified of this event.

Other types of events which are handled using sockets are when a user joins the debate and waits until it receives a notification that the judge has activated the debate, or when the judge has closed the debate and all the participants are kicked out of the current meeting.

5.3 Data Persistence with MySql

The MsSql server is used to store the application's data. The back-end server, which controls the application's flow, communicates with the database server to retrieve the required information during a debate session.
With Spring managing the persistence and retrieval of the data becomes simpler. Every class defined in the application, which is annotated with @Entity, has an associated entity/table in the database.

5.4 Daily Video-conference API

The main functionality of the application revolves around providing a video-call in which the users can join in and conduct a debate about a given statement or topic.
In order to create such a video-conference, we used the API provided by Daily, built on WebRTC, providing the functionalities of creating a new meeting, and joining in with different privileges such as a simple participant or the owner of the meeting, who can control the status of all the participants in the meeting.

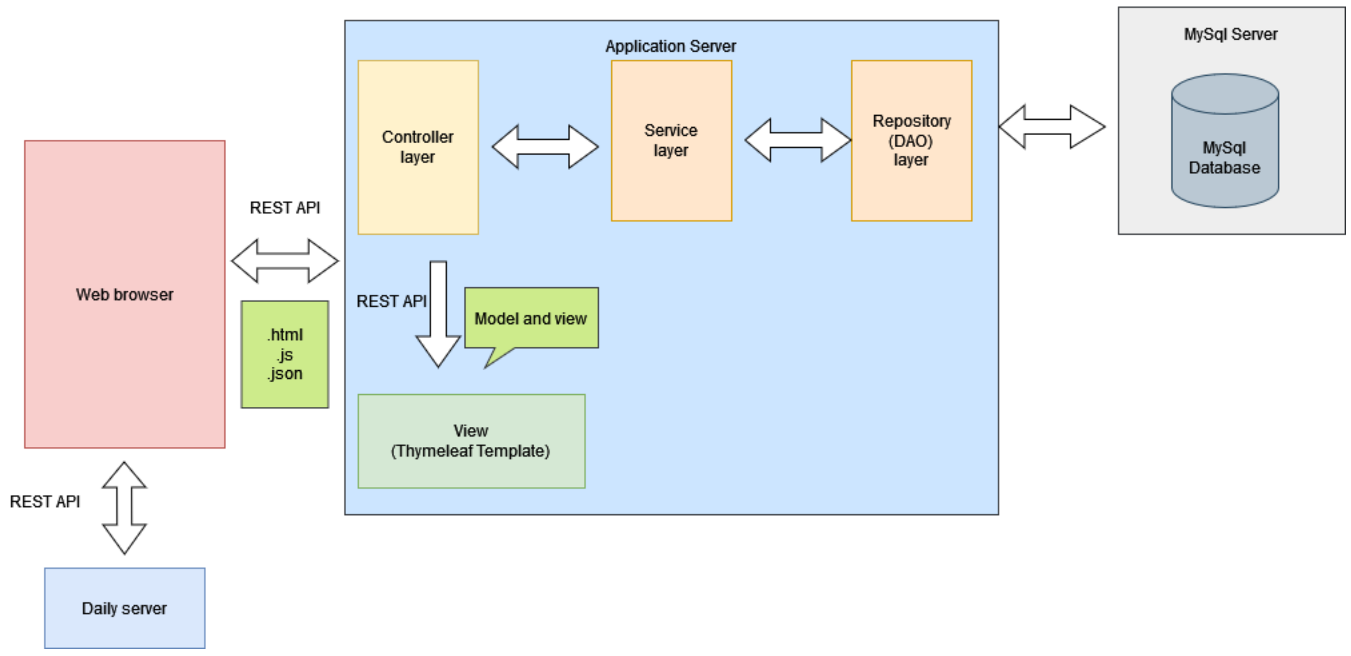
5.5 Bootstrap

For creating the user interface, we used the pre-defined tools and components provided by Bootstrap. This way, the different pages of the UI have a consistent style and look, and it provided a considerable speed-up in the development of the view layer.

Chapter 6

Architecture, Deployment Diagrams

The web-application's structure is built on top of the MVC architectural model, containing a view, controller and model (service and dao) layer.



- **Browser:** The user sends HTTP requests to the back-end server from its own browser. The browser also communicates with the Daily server which provides the video calls and allows them to join the conference.

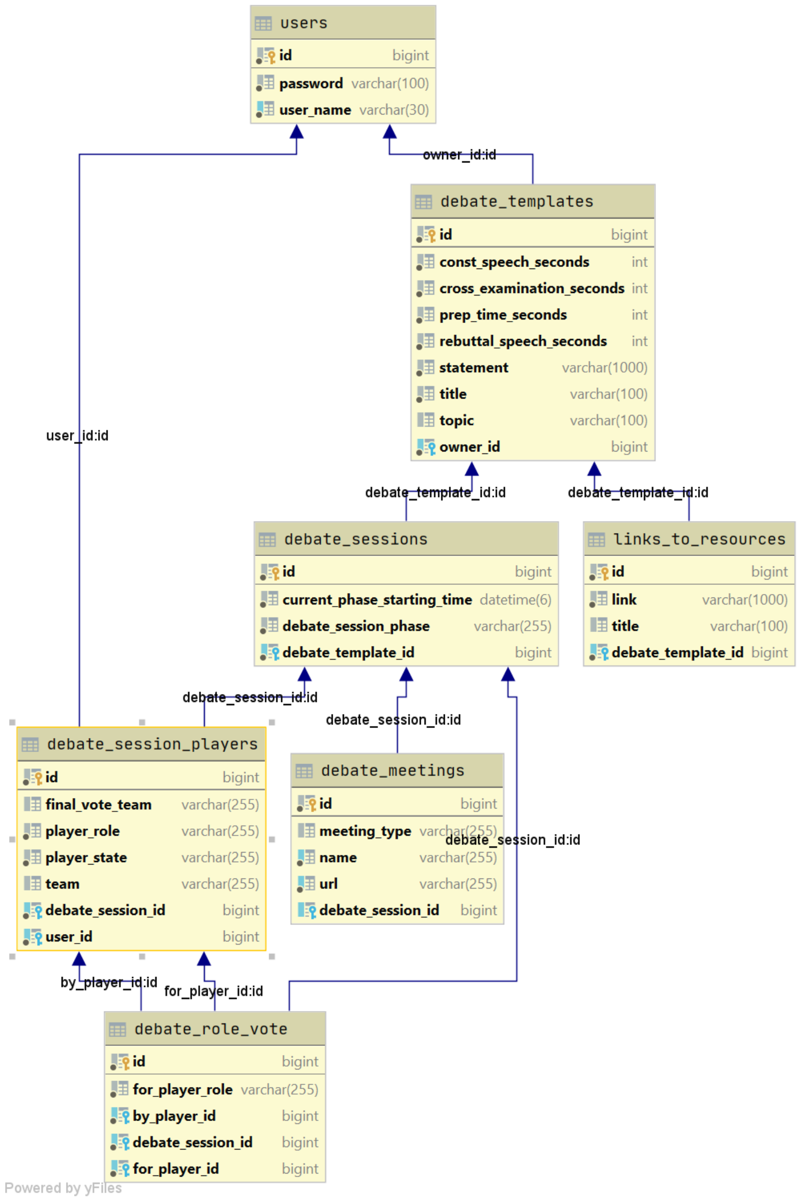
- **Controller layer:** The controller realizes the communication between the view and the model layers. It relies on the REST API to send requests to the view layer to update the UI, and it also interacts with the service layer to handle each request from the user side.

- **View layer:** For creating the view layer, the Thymeleaf template engine was used, which processes the html and javascript files of the user interface. It can also retrieve attributes added to the model and accessed in the corresponding view (html page).

- **Service layer:** The service layer contains the business logic of the application and it represents an additional layer in order to provide a simpler interface to the access the information in the data layer. It only contains repository interfaces that are implemented in the DAL.

- **Repository layer (DAL):** The DAO layer contains the entities, mapped to existing tables in the database, and the Repositories, providing an abstraction of the data storage mechanism, in order to easily process predefined CRUD operations on the data and some more complex queries, defines by us, the developers.

- **MySql Database:** The data used by the application is persisted in the database, so that it will not be lost from one execution to the other. The details of each registered user and the created debate templates/sessions is stored in the database for being able to access it later. The DAO layer communicates with the MySql database using SQL queries and update requests.

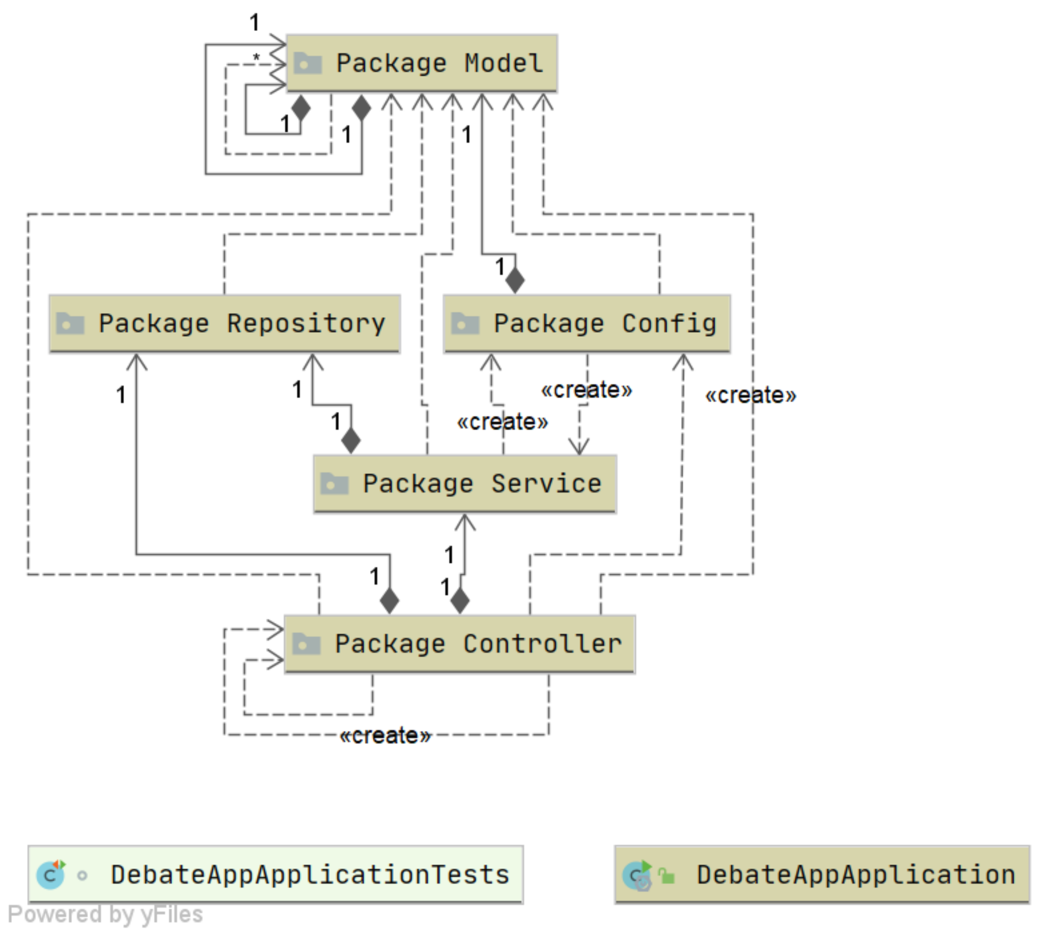


Chapter 7

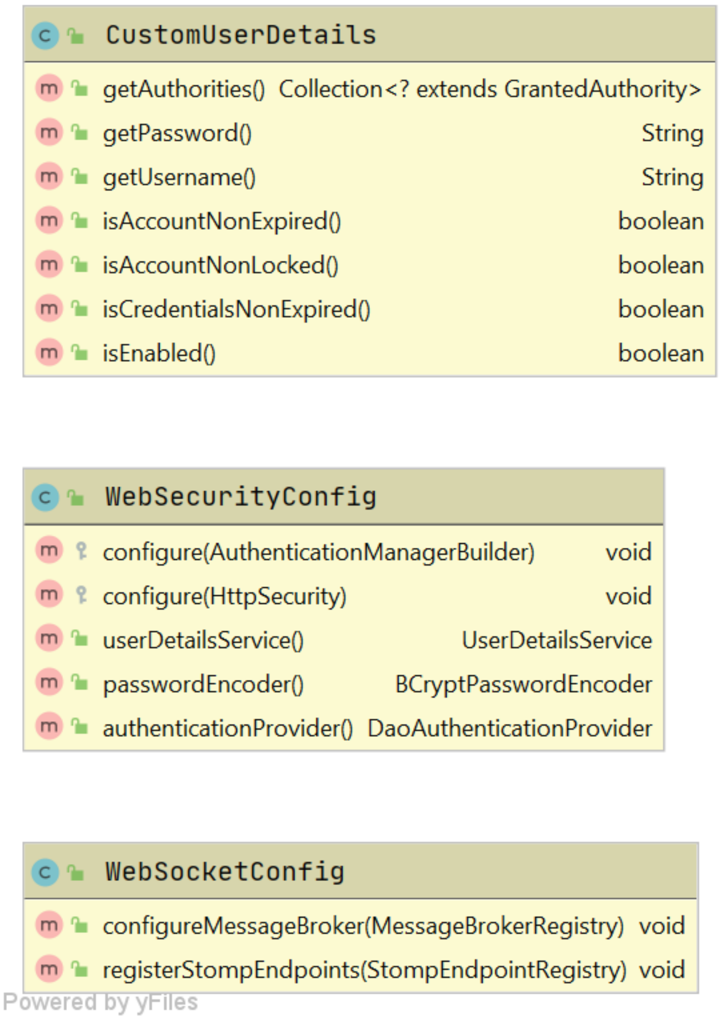
Implementation, UML Diagrams, Design Patterns

7.1 Layered Architecture

Following the above presented layered architecture, the application's components are grouped together in specific packages, each corresponding to a specific layer:



1. **Config package:** contains the Spring Security configuration classes, for managing the authentication of the users to the application as well as for controlling the access to the application's functionalities, validating the source of some incoming requests and accepting or rejecting them.


CustomUserDetails
getAuthorities() Collection<? extends GrantedAuthority>
getPassword() String
getUsername() String
isAccountNonExpired() boolean
isAccountNonLocked() boolean
isCredentialsNonExpired() boolean
isEnabled() boolean
WebSecurityConfig
configure(AuthenticationManagerBuilder) void
configure(HttpSecurity) void
userDetailsService() UserDetailsService
passwordEncoder() BCryptPasswordEncoder
authenticationProvider() DaoAuthenticationProvider
WebSocketConfig
configureMessageBroker(MessageBrokerRegistry) void
registerStompEndpoints(StompEndpointRegistry) void
Powered by yFiles

2. **Controller package:** represents the controller architectural layer and it contains the components which handle the interaction with the client by exchanging data using http requests.
It communicates with the view through the REST API, accepting get and post requests and processing the effect of the user interactions or sending an appropriate response back to the client side.

3. **Model package:** contains the classes defined to model the application's state, such as the DebateSession.java class and the required data, such as the DebateMeeting.java or DebateTemplate.java classes containing information about each debate session. The User.java and the DebateSessionPlayer.java contain information about the users, also participants in the debates.

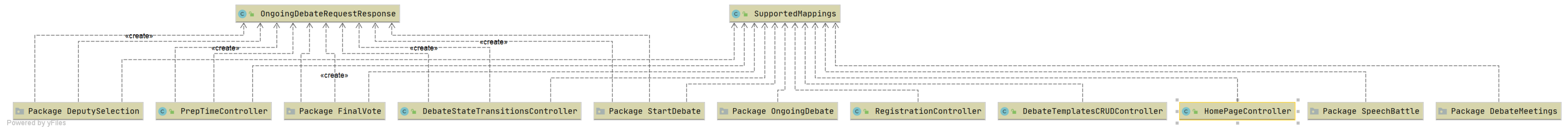
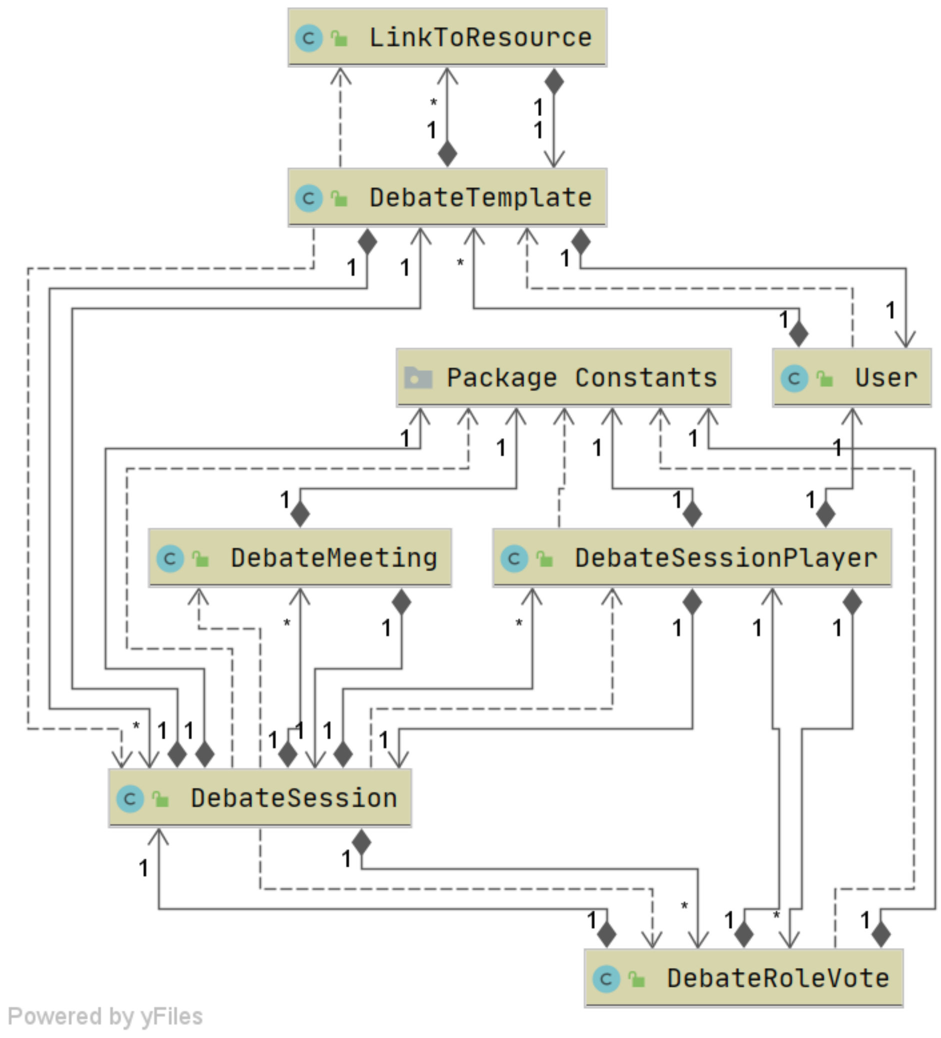
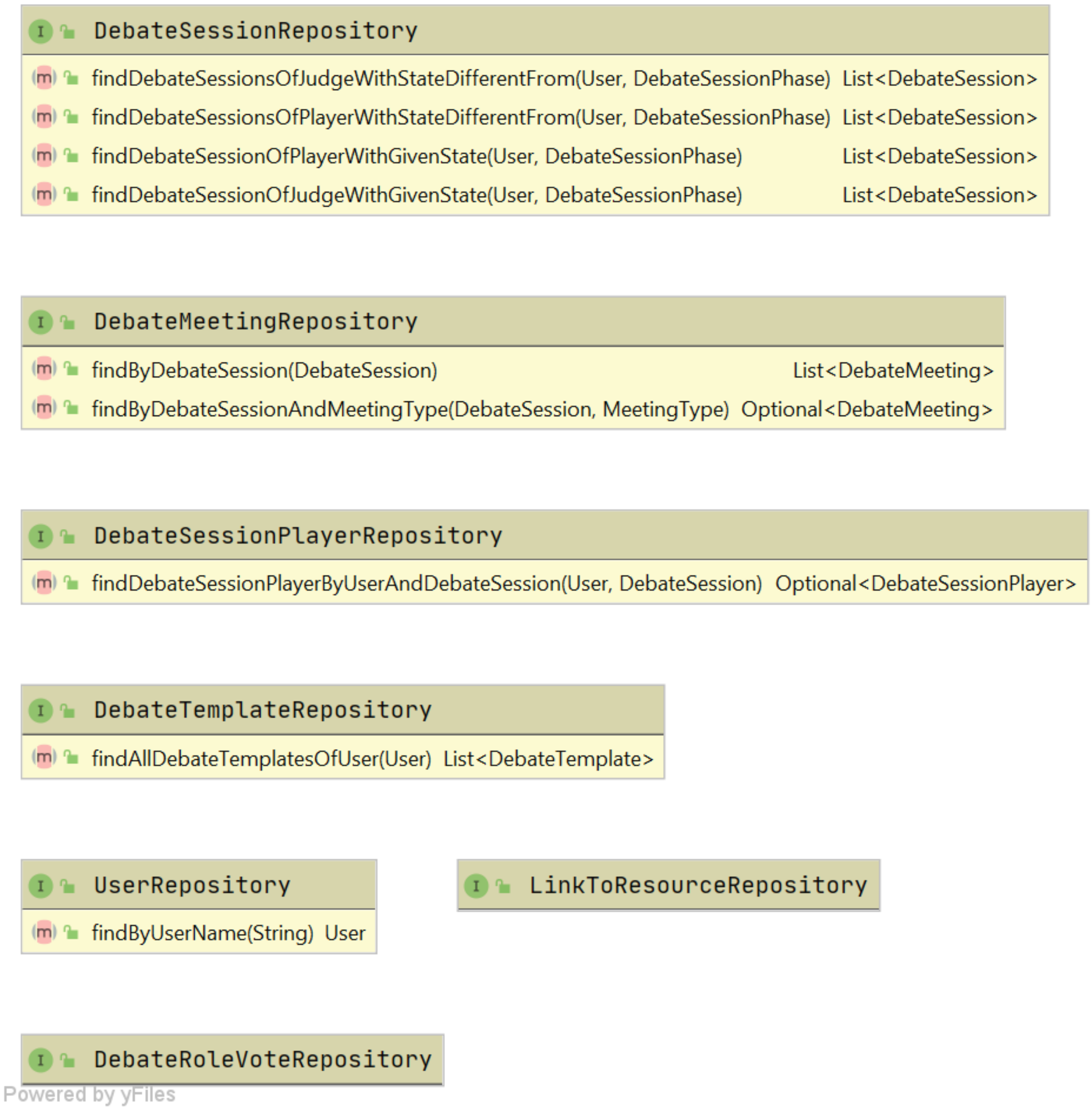


Figure 7.1: Controller layer



4. **Repository package:** represents the DAO layer of the application and it contains the Repository interfaces for managing each entity mapped to a class in the Model layer.
   It abstracts away the way the data is persisted in the application, providing default methods for storing or retrieving the information related to certain entities without having to write explicit Sql queries, except in case of more complex queries, for example which join multiple tables in order to retrieve the required information.


DebateSessionRepository
findDebateSessionsOfJudgeWithStateDifferentFrom(User, DebateSessionPhase) List<DebateSession>
findDebateSessionsOfPlayerWithStateDifferentFrom(User, DebateSessionPhase) List<DebateSession>
findDebateSessionOfPlayerWithGivenState(User, DebateSessionPhase) List<DebateSession>
findDebateSessionOfJudgeWithGivenState(User, DebateSessionPhase) List<DebateSession>
DebateMeetingRepository
findByDebateSession(DebateSession) List<DebateMeeting>
findByDebateSessionAndMeetingType(DebateSession, MeetingType) Optional<DebateMeeting>
DebateSessionPlayerRepository
findDebateSessionPlayerByUserAndDebateSession(User, DebateSession) Optional<DebateSessionPlayer>
DebateTemplateRepository
findAllDebateTemplatesOfUser(User) List<DebateTemplate>
UserRepository
findByUserName(String) User
LinkToResourceRepository
DebateRoleVoteRepository
Powered by yFiles

5. **Service package:** represents the business logic of the application. It controls the debate's flow, and the state transitions, from one phase of the debate to the other, implemented using the State pattern.

7.2 The Model

The application's internal state and the state of each user, respectively the related information about them is represented in the Model layer, through the following classes:

1. **User**
   Represents information about the user which has registered to the application, such as the username and the password. It is mapped to a corresponding entity ("users") in the

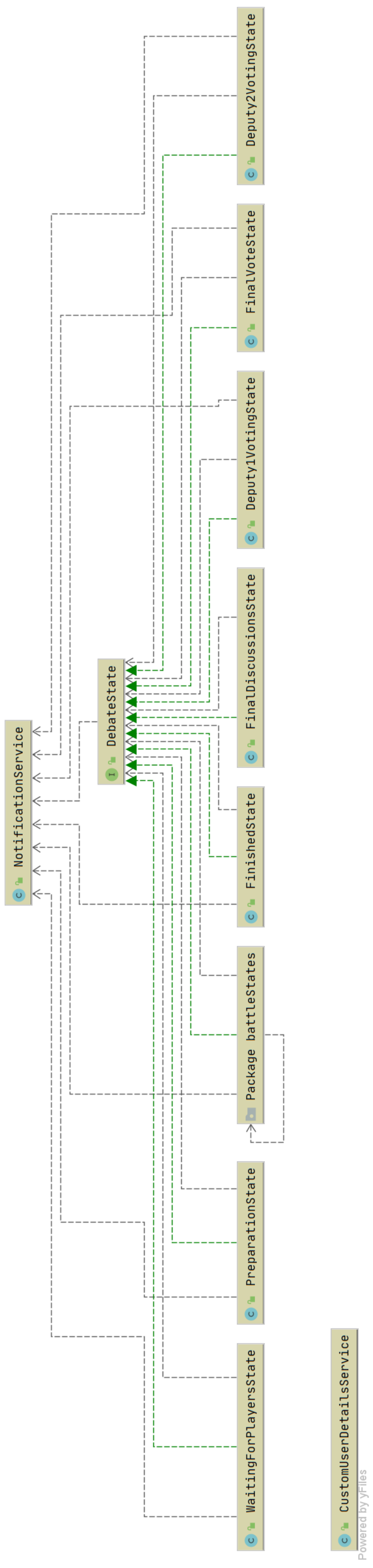
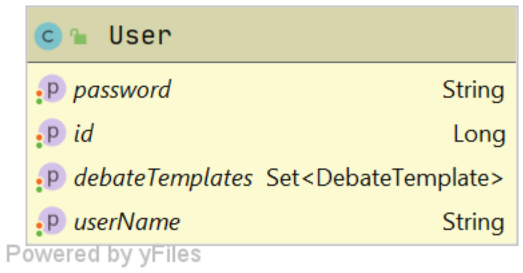


Figure 7.2: Service layer

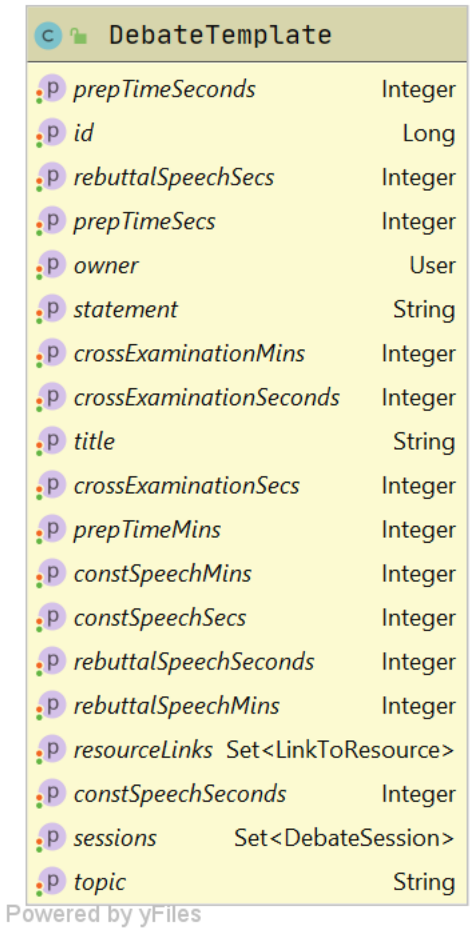
database.
Each user can be the owner of a set of debate templates that he/she has created, therefore there is a many-to-many relationship between the User and the DebateTemplate entities.



User	
password	String
id	Long
debateTemplates	Set<DebateTemplate>
userName	String

Powered by yFiles

2. **DebateTemplate**
Represents information about the debate template used to configure a debate session, including the topic or statement of a debate, the timing constraints of each phase of the debate, and the owner of the debate template which is at the same time the judge during an ongoing debate session.
Multiple debate sessions can be started from the same debate template, but a user can be part of only one ongoing debate session at a time. Therefore, there is a one-to-many relationship between the DebateTemplate and DebateSession entities.
It is mapped to a corresponding entity ("debate_templates") in the database.

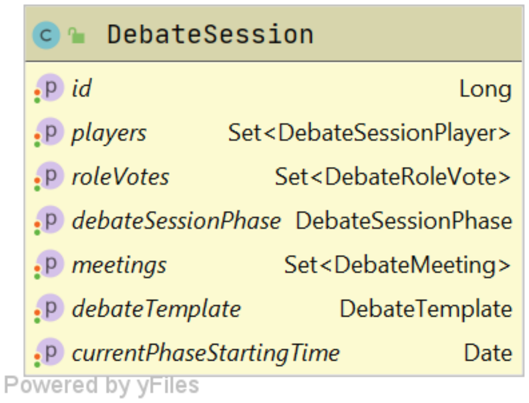


DebateTemplate	
prepTimeSeconds	Integer
id	Long
rebuttalSpeechSecs	Integer
prepTimeSecs	Integer
owner	User
statement	String
crossExaminationMins	Integer
crossExaminationSeconds	Integer
title	String
crossExaminationSecs	Integer
prepTimeMins	Integer
constSpeechMins	Integer
constSpeechSecs	Integer
rebuttalSpeechSeconds	Integer
rebuttalSpeechMins	Integer
resourceLinks	Set<LinkToResource>
constSpeechSeconds	Integer
sessions	Set<DebateSession>
topic	String

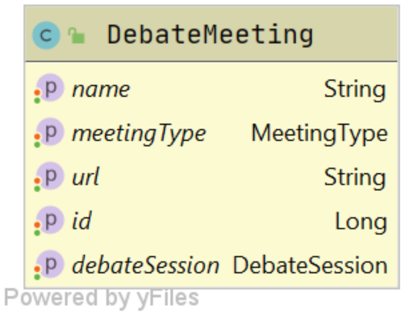
Powered by yFiles

3. **DebateSession**
Represents information about the debate session, such as the template after which it is configured, the current phase of the debate, as well as when did the current phase start,

in order to monitor the timed phases. It also contains the players who have successfully joined the debate, and finally the votes cast by each player to select the deputies who would hold the speeches.
Aside from this it contains a list of meetings which hold information about the video call's started during certain phases of the debate.
It is mapped to a corresponding entity ("debate_sesions") in the database.

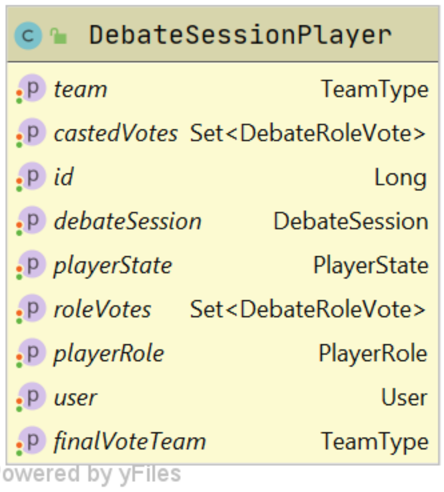

DebateSession
id Long
players Set<DebateSessionPlayer>
roleVotes Set<DebateRoleVote>
debateSessionPhase DebateSessionPhase
meetings Set<DebateMeeting>
debateTemplate DebateTemplate
currentPhaseStartingTime Date
Powered by yFiles

4. **DebateMeeting**
Represents information about a give meeting/video call, such as the name of the meeting and the url. These are necessary to be saved so that later they can be used to generate tokens with different access privileges for the players and the judge who would like to join the debate meetings. It is assumed that a player can leave/ rejoin a meeting while the debate session is still active.
For each debate session 3 meetings are created, using the Daily API-s: 2 for the preparation phase of both teams, Pro and Contra, so that they would discuss separately, and 1 for the battle phases in which all debate players should be present. The meeting type is used to identify each of these these 3 types of debate meetings.
It is mapped to a corresponding entity ("debate_meetings") in the database.

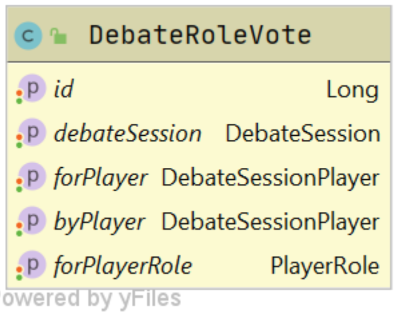


5. **DebateSessionPlayer**
Represents information about the players of a debate session such as the team to which he/she belongs, the state of the player, such as Waiting to join or Joined, the role of each player, such as Deputy1 or Deputy2 or no just a aismple participant (no role), and the debate session in which it participated.
Each user can be a debate session player in multiple debate sessions, therefore there is a one-to-many relationship between the User and the DebateSessionPlayer entities.
Each debate session player can cast their vote for selecting the deputies (2 votes) and during the final voting phase, so there is a one-to-many relationship between the DebateSessionPlayer and the DebateRoleVote entities. It is mapped to a corresponding entity

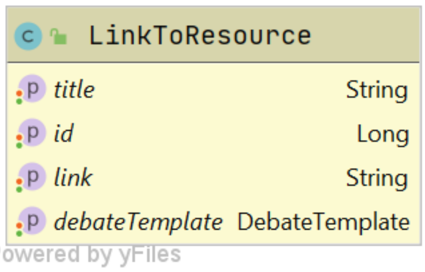
("debate_session_players") in the database.



6. **DebateRoleVote**
   Represents information about vote cast for selecting the deputies for each team during the deputy selection phases, such as the debateSessionPlayer who cast the vote (byPlayer), the candidate selected (forPlayer) and the role for which they voted (forPlayerRole), which can be either Deputy1 or Deputy2.
   It is mapped to a corresponding entity ("debate_role_votes") in the database.


DebateRoleVote
id Long
debateSession DebateSession
forPlayer DebateSessionPlayer
byPlayer DebateSessionPlayer
forPlayerRole PlayerRole
Powered by yFiles

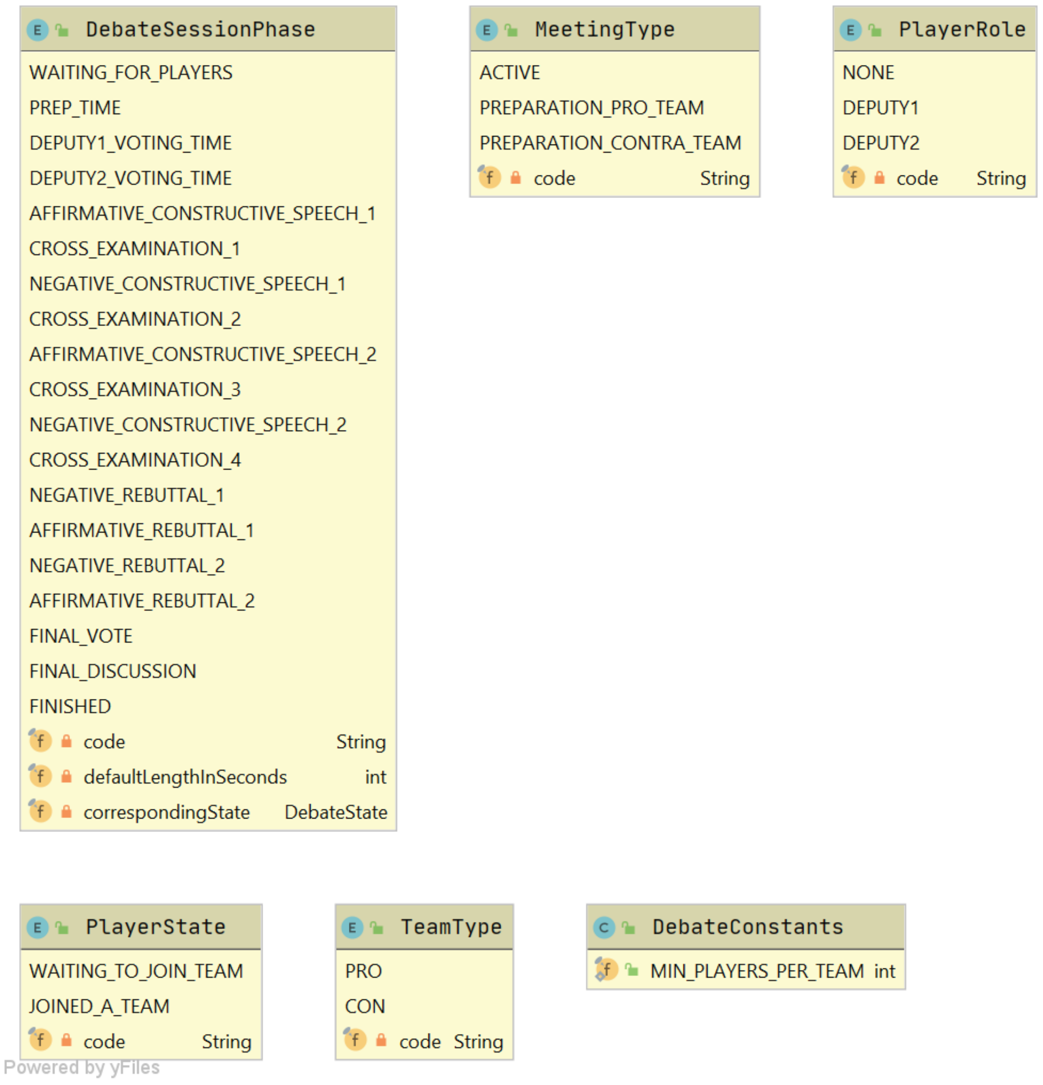
7. **LinkToResource**
   Represents information about a resource link provided by the owner of the debate for the players during the preparation time. A debate template can have many associated resource links.
   It is mapped to a corresponding entity ("links_to_resource") in the database.



8. **Constants package**
   Contains the constants defined for representing the properties of the models, such as:

   - **DebateSessionPhase:** the current phase of the debate session.

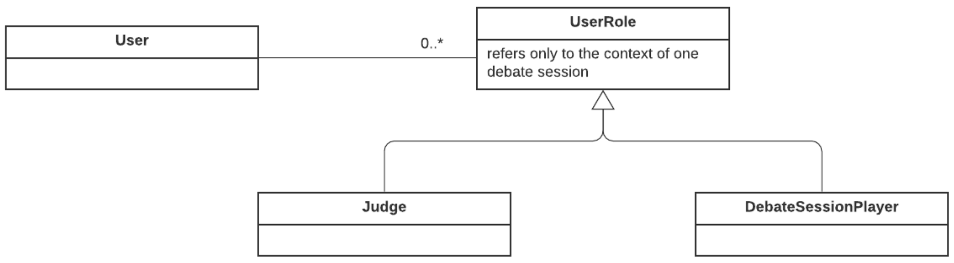
- **MeetingType:** the type of the debate meeting, can be preparation meeting or an active meeting.
- **PlayerRole:** the role assigned to a debate session player, which can be either a deputy or a simple participant.
- **PlayerState:** the current state of the players, can be waiting to join a team or already joined a team.
- **TeamType:** distinguishes 2 teams: Pro and contra.
- **DebateConstants:** specifies the min number of players needed to start a debate.



7.3 Player-Role pattern

7.3.1 A modified application of the pattern, for the Judge/Player roles

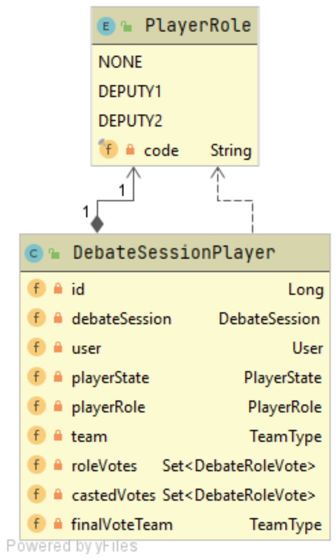
As previously stated, the users of our application may have different roles in different debate sessions: a user may be a judge in one debate session and a player in the other one. To express this in code, we applied the Player-Role pattern, in which the players are the users, and the roles are the "judge" and the "debate session player".



However, given the structure of our relational database, the UserRole and the Judge classes do not appear explicitly: the judge of a debate session is defined by the owner of a debate template based on which the session was created, instead.

7.3.2 A direct application of the pattern, for the roles of a player inside a Debate Session

A more explicit version of the Player-Role pattern was applied to specify the role of a player inside a debate session: initially, all players have the role "None", but later some of them may take up the roles "Deputy1" or "Deputy2" of their team.



In this context, the PlayerRole class was implemented as an enum with the enum constants corresponding to the different roles that a debate session player can take up.

7.4 State Pattern

It can me immediately observed, that a Debate Session consists of multiple consecutive states, and the judge's/players' rights and the possible actions depend on current state of the debate session that they are the participants of.

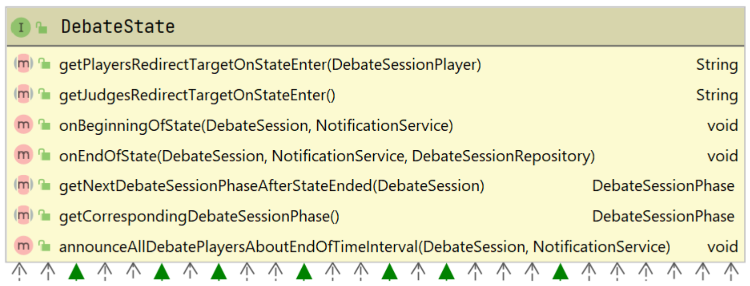
The order of the states is generally the following:

1. Waiting for Participants: after the judge starts the debate, the debate cannot be activated until at least 2-2 players join each team. When there are enough players, it's up to the judge to decide when to kick off the debate with the Preparation State.
2. Preparation State
3. Deputy 1 Voting State
4. Deputy 2 Voting State
5. 1st Affirmative Constructive State: 1AC
6. Cross Examination 1
7. 1st Negative Constructive State: 1NC
8. Cross Examination 2
9. 2nd Affirmative Constructive State: 2AC
10. Cross Examination 3
11. 2nd Negative Constructive State: 1NC
12. Cross Examination 4
13. 1st Negative Rebuttal State: 1NR
14. 1st Affirmative Rebuttal State: 1AR
15. 2nd Negative Rebuttal State: 2NR
16. 2nd Affirmative Rebuttal State: 2AR
17. Final Voting State
18. Final Discussion State
19. Finished State

Moreover, based on the template of the debate defined by the judge, prior to the debate, some phases may have the length 0, meaning that they will be skipped. In this sense, **each state is optional**.

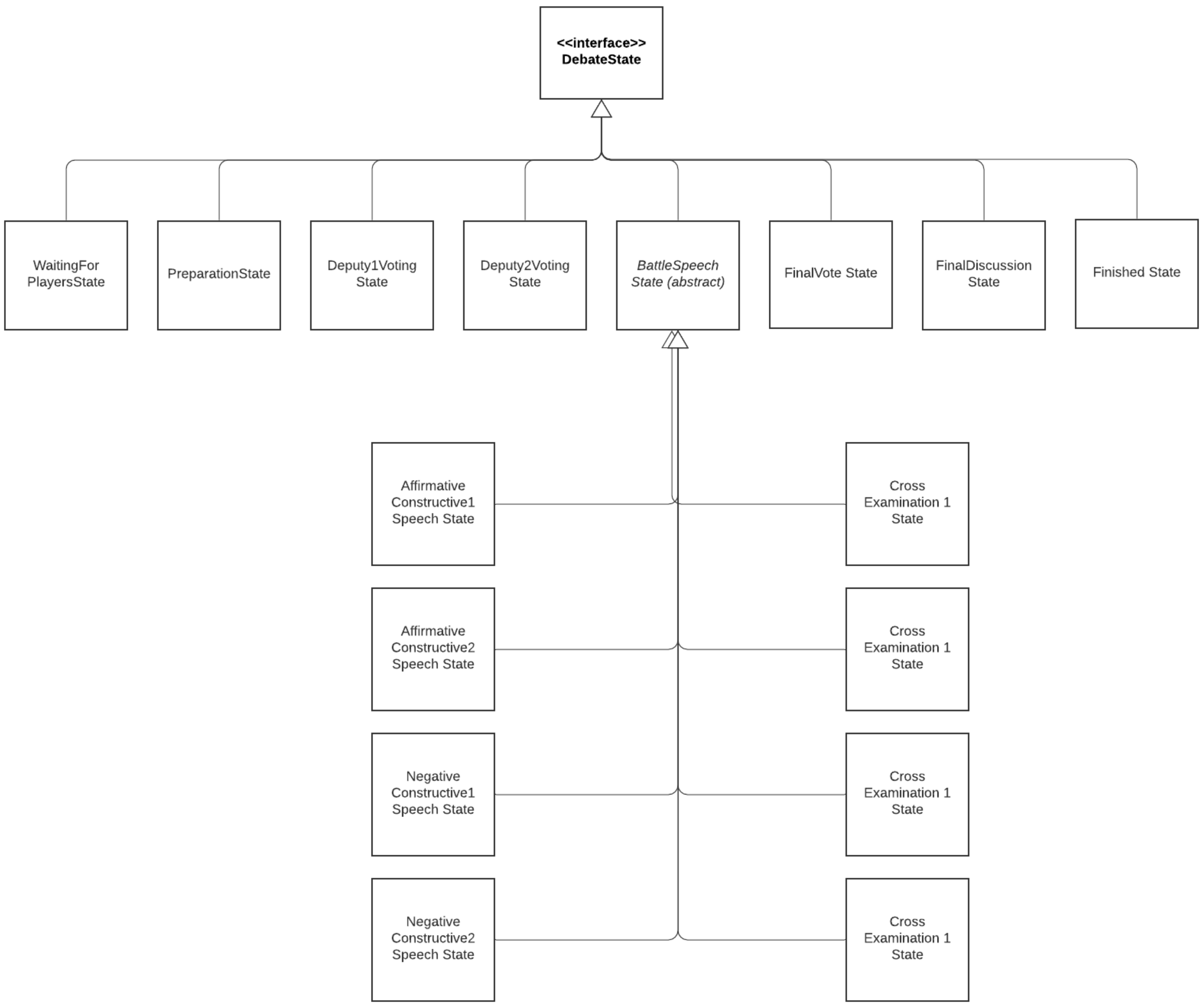
Furthermore, the judge may decide at any time during the debate session, to **skip the current phase** (for example, if the speakers expected to deliver their speech in that state are not present), or to **close the debate session entirely**. In the first case, the debate session proceeds to the next state, whereas in the letter case, the debate session is set to the "Finished" state.

Handling the behavior of so many states, knowing when and where each user needs to be redirected and when notifications need to be sent to them would have been nearly impossible without applying the state pattern. Thus, we defined the DebateState interface and implemented it with 1 class separately for each of the above debate states.



DebateState

getPlayersRedirectTargetOnStateEnter(DebateSessionPlayer)	String
getJudgesRedirectTargetOnStateEnter()	String
onBeginningOfState(DebateSession, NotificationService)	void
onEndOfState(DebateSession, NotificationService, DebateSessionRepository)	void
getNextDebateSessionPhaseAfterStateEnded(DebateSession)	DebateSessionPhase
getCorrespondingDebateSessionPhase()	DebateSessionPhase
announceAllDebatePlayersAboutEndOfTimeInterval(DebateSession, NotificationService)	void


<<interface>>
DebateState
WaitingFor PlayersState
PreparationState
Deputy1Voting State
Deputy2Voting State
BattleSpeech State (abstract)
FinalVote State
FinalDiscussion State
Finished State
Affirmative Constructive1 Speech State
Cross Examination 1 State
Affirmative Constructive2 Speech State
Cross Examination 1 State
Negative Constructive1 Speech State
Cross Examination 1 State
Negative Constructive2 Speech State
Cross Examination 1 State

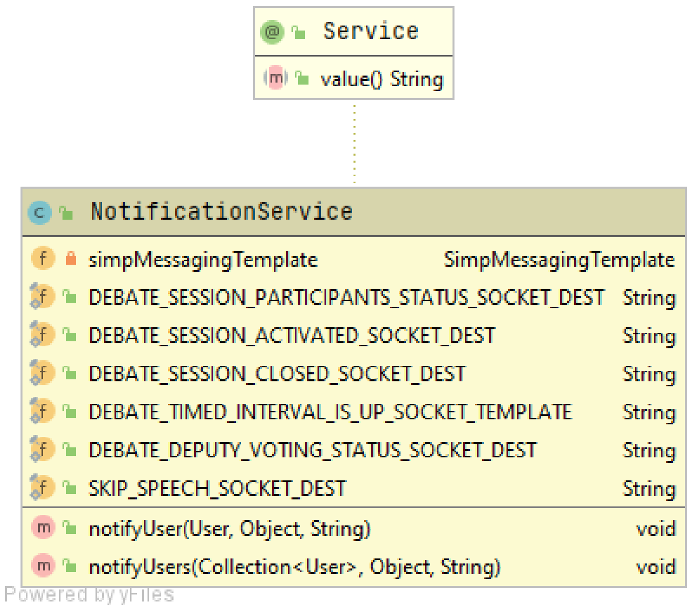
7.5 Observer Pattern

Another critical part of the application, given that it is in fact a multiplayer game in which what each player of the same debate session sees must be synchronized, was the Observer Pattern: we had to notify specific users at specific events. For example, we had to announce

- all players of the debate session, whenever a state transition in that debate session occurred
- the the judge, whenever a new player joined a debate session/a team of the debate session
- the judge, whenever a new vote got cast for the deputies
- all players of the debate session, if the judge closed the debate
- ...

Unfortunately, we couldn't apply the Observer Pattern directly, since the Observable objects were on the server side and the Observers were the clients. To overcome this issue, we implemented a modified version of the Observer pattern through WebSockets. For this, the clients had to subscribe to the events = WebSockets that they were interested in, and whenever an event occurred, the server simply sent a message to all the clients that the event had an effect on and who subscribed to that socket. Then, it was the client who knew how to handle the message, notification received from the server: the server's only duty was to notify the clients, but didn't need to handle any of the front-end events.

To make the usage of the WebSockets simpler, we defined the NotificationService Service, as it can be seen in the following figure.



Service
value() String
NotificationService
simpMessagingTemplate SimpMessagingTemplate
DEBATE_SESSION_PARTICIPANTS_STATUS_SOCKET_DEST String
DEBATE_SESSION_ACTIVATED_SOCKET_DEST String
DEBATE_SESSION_CLOSED_SOCKET_DEST String
DEBATE_TIMED_INTERVAL_IS_UP_SOCKET_TEMPLATE String
DEBATE_DEPUTY_VOTING_STATUS_SOCKET_DEST String
SKIP_SPEECH_SOCKET_DEST String
notifyUser(User, Object, String) void
notifyUsers(Collection<User>, Object, String) void
Powered by yFiles

Appendix A

Final Project

Since the amount of the code written for the final project is too large, it was impossible to include it fully in this pdf document. Instead, we published the entire code on GitHub. Please follow this link to access it.

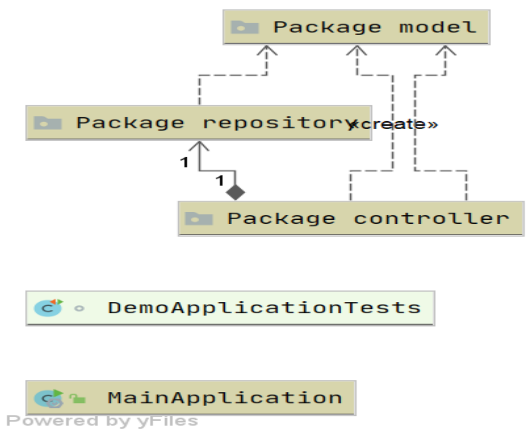
Appendix B

Mini Project

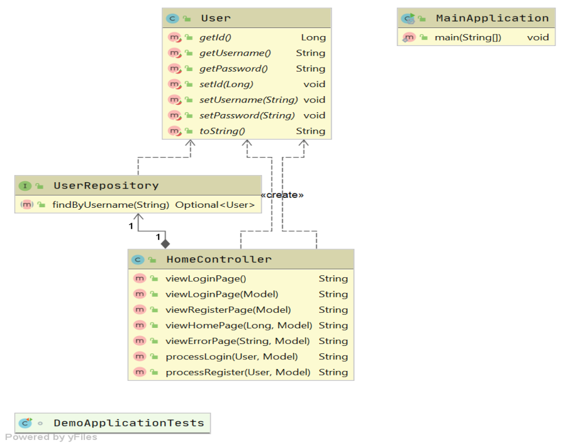
The mini project consist of a simple web application built on top of the MVC pattern, using the Spring Web MVC Framework.
The user can register or log in to the application with a username and a password. Upon successful authentication or registration he/she will be redirected to the home page of the application, while upon failure he/she will be redirected to an error page.

On the client side the user can view the registration, login and home pages and navigate between them. Their inputs will be sent to the back-end server in the form of http requests.
On the server side, the incoming http requests from the clients are handled using controllers, such as HomeController.java class.
The http server, representing an embedded Tomcat server, communicates in turn with a MySql database to store or retrieve the required information for the application.
The model is represented by the User.java class, which corresponds to an existing table in the database, and it is managed by the UserRepository.class, fetching or saving the user related data to the database.

The following UML diagrams illustrates the relationship between the classes in the model-view-controller layer:



(a) Package level



(b) Class level

Figure B.1: The UML diagram of the mini project's components

```java
package com.example.demo;
```

```java

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MainApplication {

  public static void main(String[] args) {
    SpringApplication.run(MainApplication.class, args);
  }

}
```

Listing B.1: Main application

```java
package com.example.demo.model;

import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import lombok.ToString;

import javax.persistence.*;

@Entity
@Table(name = "user")
@NoArgsConstructor
@Getter
@Setter
@ToString
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(nullable = false, unique = true, length = 100)
    private String username;
    @Column(nullable = false, length = 100)
    private String password;
}
```

Listing B.2: Model

```java
package com.example.demo.repository;

import com.example.demo.model.User;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

import java.util.Optional;

public interface UserRepository extends JpaRepository<User, Long> {

    @Query("Select u FROM User u WHERE u.username = ?1")
    Optional<User> findByUsername(String username);
}
```

Listing B.3: Repository

```java
package com.example.demo.controller;

import com.example.demo.model.User;
import com.example.demo.repository.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestParam;

import java.util.Optional;

@Controller
public class HomeController {

    private static final String INDEX_PAGE_URL = "/index";
    private static final String HOME_PAGE_URL = "/home";
    private static final String LOGIN_PAGE_URL = "/login";
    private static final String REGISTER_PAGE_URL = "/register";
    private static final String ERROR_PAGE_URL = "/error";

    private static final String INEXISTENT_USER_ERROR = "User with given id
   doesn't exist";
    private static final String FAILED_TO_LOG_IN_ERROR = "Failed to log in
   with given username and password!";
    private static final String DUPLICATE_USERNAME_ERROR = "There already
   exists another user with the given username!";

    @Autowired
    UserRepository userRepository;

    @GetMapping("/index")
    public String viewLoginPage() {
        return INDEX_PAGE_URL;
    }

    @GetMapping("/login")
    public String viewLoginPage(Model model) {
        model.addAttribute("user", new User());
        return LOGIN_PAGE_URL;
    }

    @GetMapping("/register")
    public String viewRegisterPage(Model model) {
        model.addAttribute("user", new User());
        return REGISTER_PAGE_URL;
    }

    @GetMapping("/home")
    public String viewHomePage(@RequestParam Long id, Model model) {
        Optional<User> user = userRepository.findById(id);
        if (user.isPresent()) {
            model.addAttribute("user", user.get());
            return HOME_PAGE_URL;
        } else {
            return viewErrorPage(INEXISTENT_USER_ERROR, model);
        }

    }

    @PostMapping("/error")
    public String viewErrorPage(@RequestBody String errorMessage, Model model) {
        model.addAttribute("errorMessage", errorMessage);
        return ERROR_PAGE_URL;
    }

    @PostMapping("process_login")
    public String processLogin(User user, Model model) {
        Optional<User> savedUser = userRepository.findByUsername(user.getUsername());
        if (savedUser.isPresent() && savedUser.get().getPassword().equals(user.getPassword())) {
            return viewHomePage(savedUser.get().getId(), model);
        }
        return viewErrorPage(FAILED_TO_LOG_IN_ERROR, model);
    }

    @PostMapping("process_register")
    public String processRegister(User user, Model model) {
        Optional<User> existingUser = userRepository.findByUsername(user.getUsername());
        if (existingUser.isEmpty()) {
            // the username is unique
            User savedUser = userRepository.save(user);
            return viewHomePage(savedUser.getId(), model);
        }
        return viewErrorPage(DUPLICATE_USERNAME_ERROR, model);
    }
}
```

Listing B.4: Controller

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Welcome</title>
    <link rel="stylesheet" th:href="@{/styles.css}" />
</head>
<body>
<div class="container text-center">
    <p>Welcome!</p>
    <a
        th:href="@{/login}"
        class="button"
        >
        Log in
    </a>
    <a
            th:href="@{/register}"
            class="button"
    >
        Register
    </a>
</div>
</body>
</html>
```

Listing B.5: Index view

```
<!DOCTYPE html>
```

```
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <title>Login</title>
  <link rel="stylesheet" th:href="@{/styles.css}" />
</head>
<body>
<div class="container text-center">
  <form
          th:object="${user}"
          th:action="@{/process_login}"
          method="post"
    >
    <h3>Login</h3>
    <label th:for="username-input">Username</label>
    <input id="username-input" th:field="*{username}" type="text">
    <label th:for="password-input">Password</label>
    <input id="password-input" th:field="*{password}" type="password">
    <button
            id="login-button"
            type="submit"
            class="button"
    >
      Login
    </button>
    <a th:href="@{/register}" class="button">
      Don't have an account yet? Click here to register.
    </a>
  </form>
</div>
</body>
</html>
```

Listing B.6: Login view

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Register</title>
    <link rel="stylesheet" th:href="@{/styles.css}" />
</head>
<body>
<div class="container text-center">
    <form
            th:object="${user}"
            th:action="@{/process_register}"
            method="post"
    >
        <h3>Register</h3>
        <label th:for="username-input">Username</label>
        <input id="username-input" th:field="*{username}" type="text">
        <label th:for="password-input">Password</label>
        <input id="password-input" th:field="*{password}" type="password">
        <button
                id="login-button"
                class="button"
                type="submit">
            Register
        </button>
    </form>
```

```html
</div>
</body>
</html>
```

Listing B.7: Register view

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Home</title>
    <link rel="stylesheet" th:href="@{/styles.css}" />
</head>
<body>
<div class="container text-center" th:object="${user}">
    <p th:text="'Welcome ' + *{username} + ' !'"></p>
    <a
            th:href="@{/index}"
            class="button"
    >
        Log out
    </a>
</div>
</body>
</html>
```

Listing B.8: Home view

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Error</title>
    <link rel="stylesheet" th:href="@{/styles.css}" />
</head>
<body>
<div class="container text-center">
    <h1>An error occurred!</h1>
    <p th:text="${errorMessage}"></p>
    <a
            th:href="@{/index}"
            class="button"
    >
        Go back to start page
    </a>
</div>
</body>
</html>
```

Listing B.9: Error view

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.
   org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache
   .org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.5.6</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.example</groupId>
  <artifactId>demo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>demo</name>
  <description>Demo project for Spring Boot</description>
  <properties>
    <java.version>11</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-devtools</artifactId>
      <scope>runtime</scope>
      <optional>true</optional>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <version>1.18.22</version>
      <scope>provided</scope>
    </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <scope>runtime</scope>
    </dependency>

    </dependencies>

  <build>
    <plugins>
```

```
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>14</source>
          <target>14</target>
        </configuration>
      </plugin>
    </plugins>
  </build>

</project>
```

Listing B.10: Pom.xml file