Software Engineering Methods – Group 34B

Software Architecture First Draft

Group Members:

Maarten van der Weide

Stefan Creasta

Rithik Appachi Senthilkumar

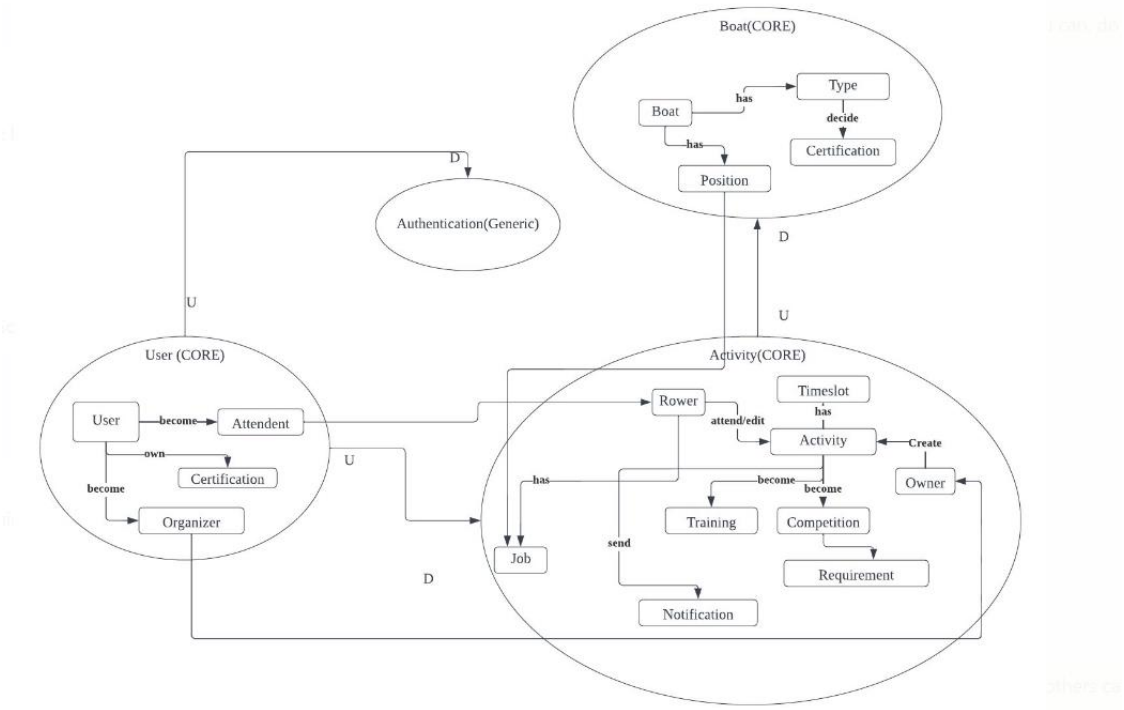
Yongcheng Huang

Viet Luong

Vlad Iftimescu

Contents:

Bounded context-map:



Explanation of the context map:

Bounded contexts: this rowing system's bounded context map contains three core contexts and a generic one. It has a User, an Activity and a Boat context, all of them

being core contexts due to the fact that they are specific and essential to the system. Then it has an Authentication context, which is deemed to be generic, since it is not specific to this system.

1. **User (core):** the context that represents the actual users of the applications. It houses the essential information for identifying individual users of the system, such as their username and certifications.
   Users, in the context of Activities, can then become:
   - Organizers, which then becomes an Owner. Owners receive additional permissions for which the Activities that they own;
   - Attendants, which become Rowers in the same context, and who can attend/edit and receive notification for specific Activities. Each Rower then gets assigned a specific Job in the context of a specific Activity.

2. **Activity (core):** the context that houses all the information necessary for creating, managing and removing Activities of all types. It contains attributes that are used for differentiating different Activities, such as a Timeslot, the Rowers that are taking part and the Owner of the Activity.
   There are two main types of Activities:
   - Training sessions, which are represented by the Training class;
   - Contests and competitions, which are housed in the Competition class. They receive an additional Requirement attribute, which describes the previous steps that a Rower should have achieved before participating in that specific Competition.

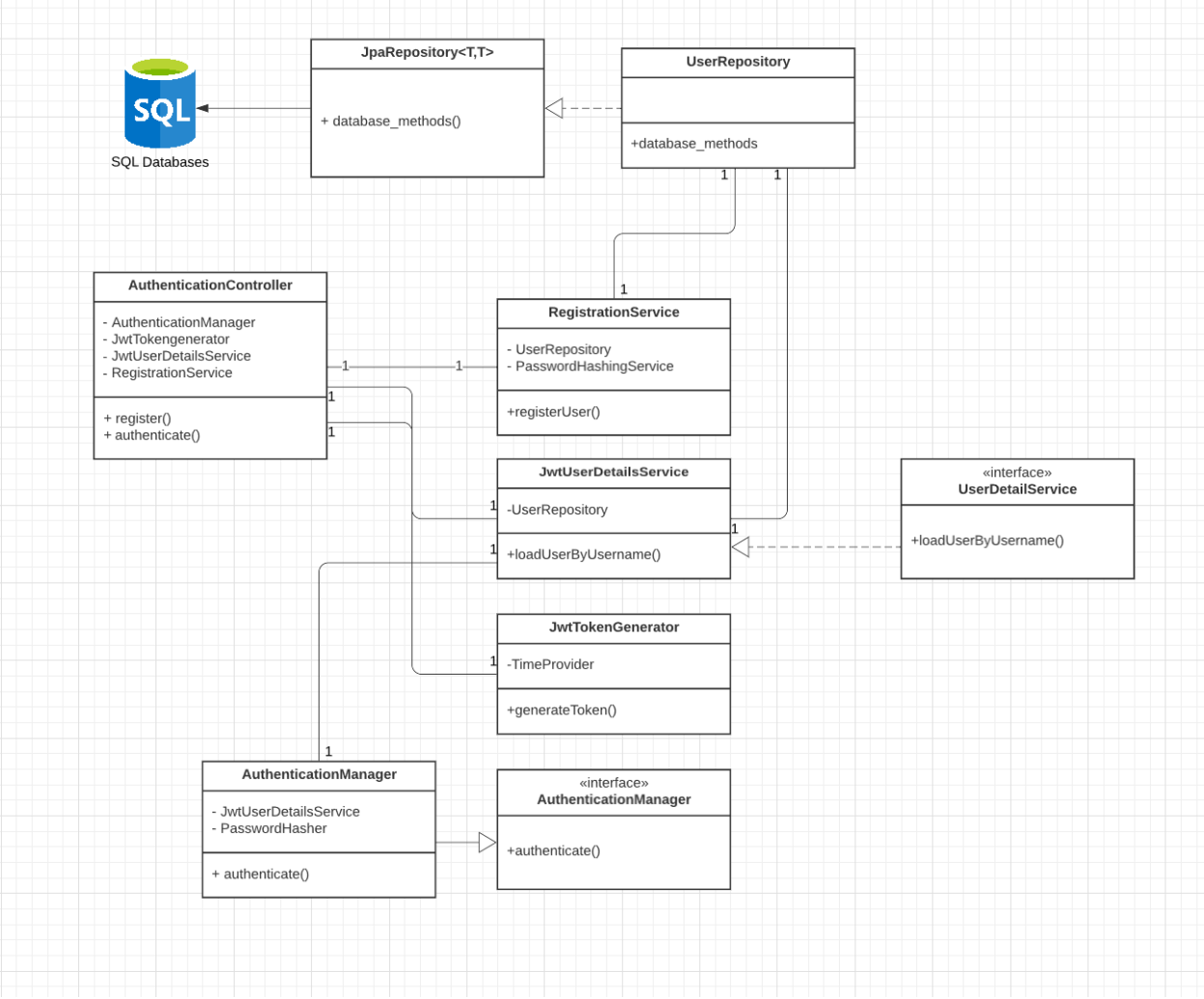
3. **Boat (core):** the context that encapsulates all the important information for the different boats that are part of certain Activities.
   A Boat contains:
   - a certain Type, which is determined by a Certification;
   - a Position, which then becomes a Job in the context of an Activity.

Authentication (generic): which is responsible for all of the security-related aspects of the system. It takes care of functionalities such as user authentication, their permissions and the verification of the requests that are sent to the system. All of the security is handled by the Spring Security service, which means that Users will have to register and log in using an username/email address and a password.

Authentication context microservice:

The authentication microservice in our project is responsible for providing users with a token that enables them to access all other microservices. This token is granted by the authentication microservice in exchange for correct login information consisting of a netID and a password.

The authentication microservice exposes this functionality with 2 API-endpoints, one is called register and one is called authenticate.



The above is a UML diagram depicting the core functionality of the microservice. The exposed endpoints are handled in the authentication

controller, where a pipeline is executed depending on what endpoint was accessed.

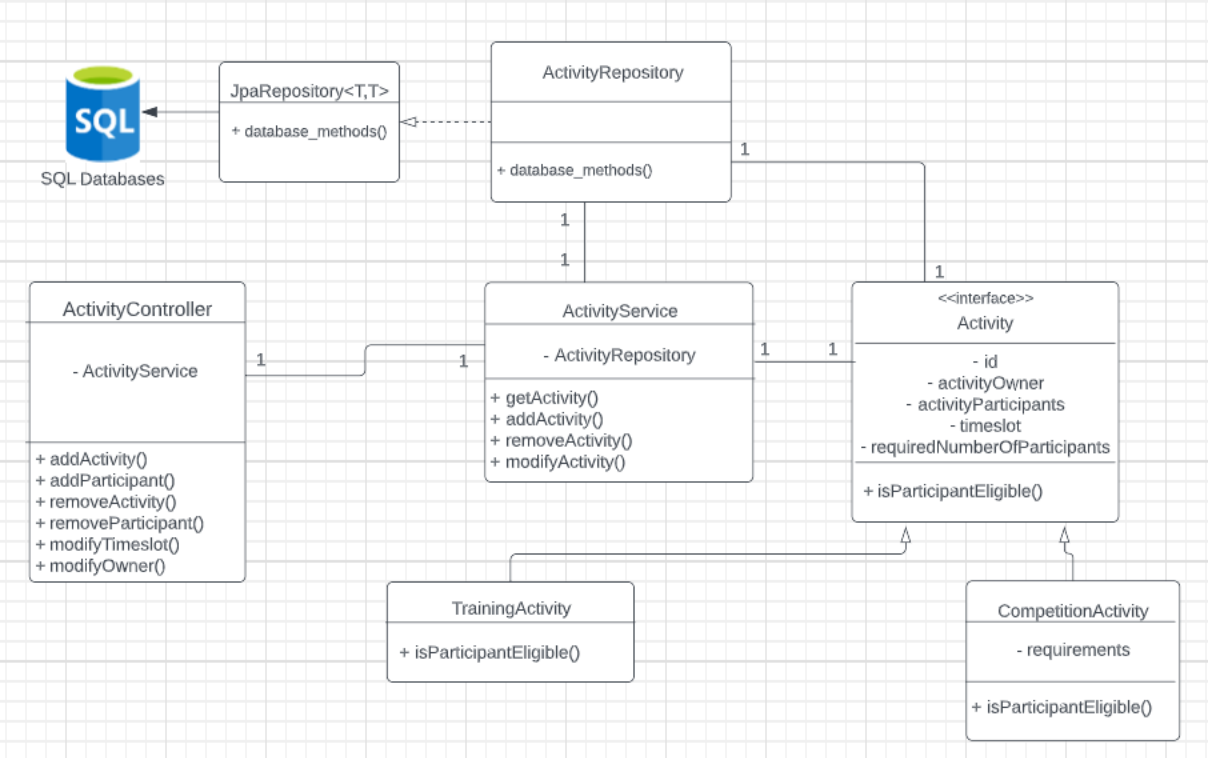
Given that a user tries to authenticate itself with the authenticate API endpoint the following happens: The authentication controller uses the “AuthenticationManager” interface provided by Spring Security to authenticate the details provided by the user. This is achieved by using the “JwtUserDetailsService“ to see if the provided details match a stored record in the database. For further clarity, the concrete implementation of the “AuthenticationManager” is provided by Spring. If this authentication is successful, the controller will use the “JwtTokenGenerator” to create a Jwt token, this is then returned to the requestee.

Now we consider the case that a user tries to register with out endpoint. The controller uses the “RegistrationService” to store the new user record in the database, where of course the “RegistrationService” uses it’s” UserRepository” to interface with the database.

Furthermore, this microservice contains a handful of data models and value objects / entities which I have chosen to emit for simplicities sake.

Activity context microservice:

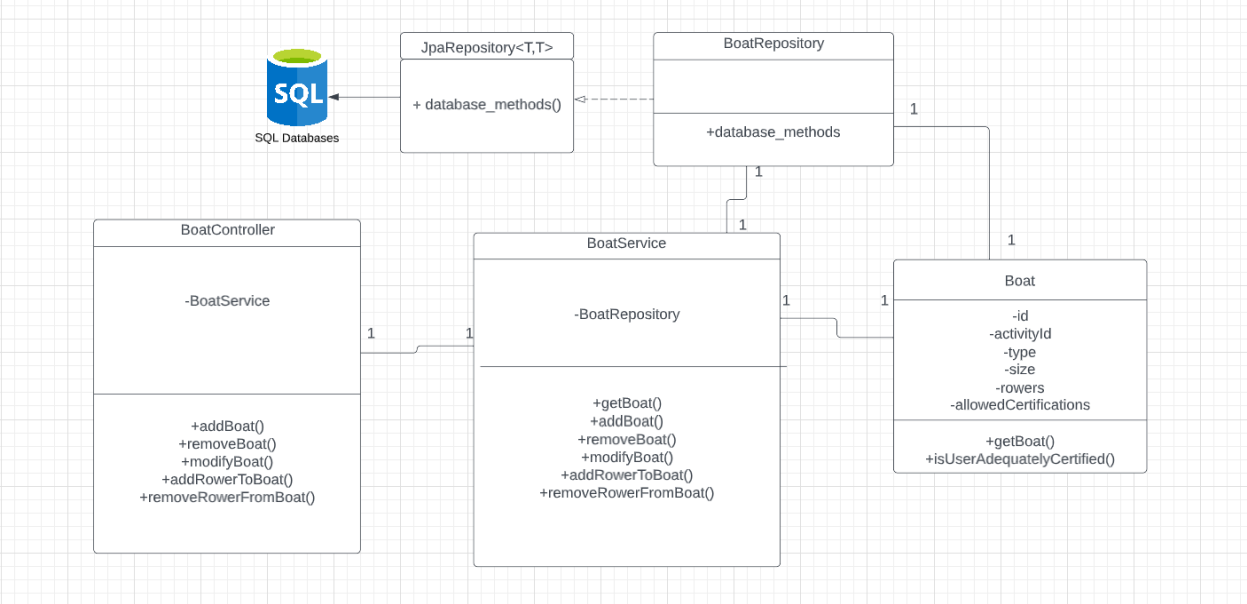
The activity microservice in our project is responsible for adding, removing and modifying activities in the activity database. An activity has an owner, a list of participants, a timeslot and the number of required participants. Because of this, there are 6 API gateways, 2 of which are designated to remove or add activities, and the other 4 which modify the fields of an activity. Note that when an activity is created, the owner should be known (the user who created the activity).



The UML diagram presents how the microservice works. Depending on the type of API accessed in activity controller, different methods will be called inside the activity service. If the fields of an activity must be modified, then the modifyActivity method will be called. Otherwise, addActivity or removeActivity will be called. Whenever we have to modify an activity, we first need to get it. So, the database retrieves it as an object, Activity, and then the service can create a new activity with the new fields, and replace the old Activity object with the new one. At the same time, an activity can either be a training or a competition. The isParticipantEligible method inside the TrainingActivity class checks whether the user sent the request within half an hour before the start of the training (in which case, he shouldn't be able to join it). The isParticipantEligible method inside the CompetitionActivity checks whether the user meets certain requirements (all rowers in the boat need to be part of the same organization for example, or that some competitions should not let amateur rowers be participants) and whether the user sent the request within a day before the start of the contest. At the same time, in order for a user to be accepted as a participant, it must be first approved by the owner.

Boat context microservice:

The boat microservice is responsible for adding, removing and modifying boats in the boat database. A boat has a unique ID, the activityId (for the activity that this boat is being used for), a type (implying the possibility of having more types in the future), the size (total size of the boat), the list of rowers, and a list of allowed certifications (a user must possess at least one certificate in said list in order to be allowed to operate with that boat). When a boat is created, a unique ID for said boat is generated.



The UML diagram details how this microservice works. Different methods in the BoatService will be called depending on the API accessed in the BoatController.

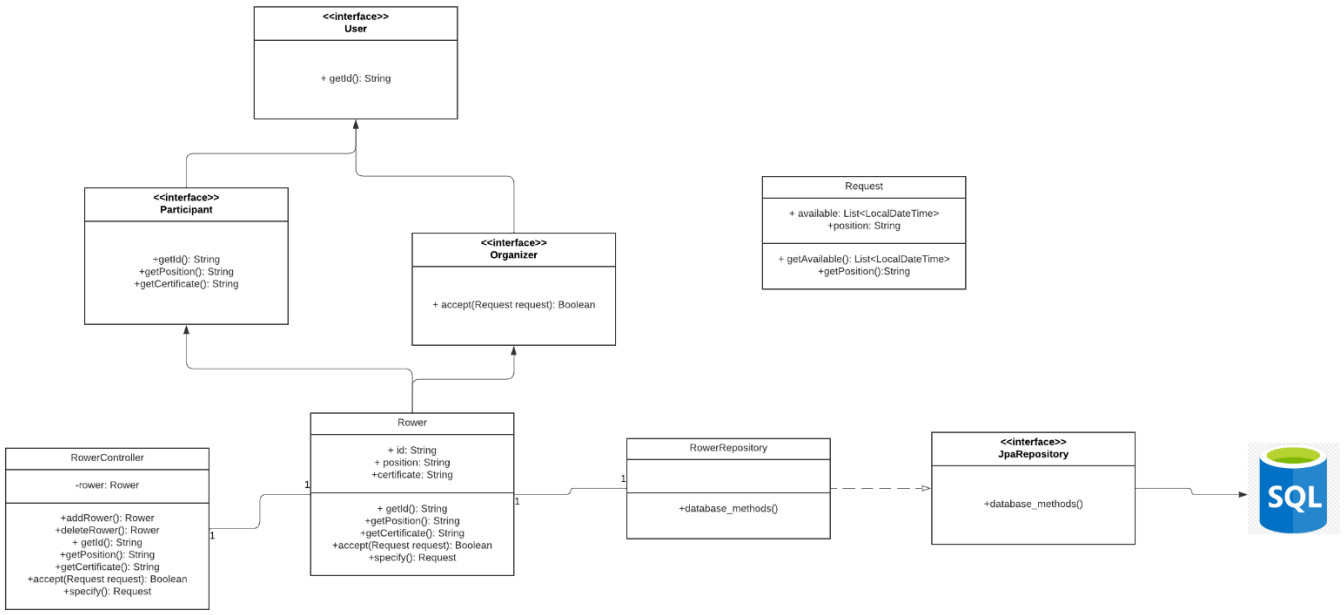
If the fields of a boat (such as its type) needs to be modified, then the modifyBoat method will be called. The database here retrieves the boat as a Boat object, and the new Boat object created by the service replaces the old one. To add/remove a boat, the relevant methods addBoat and removeBoat respectively are called. In order to add/remove a rower from a

boat, the methods addRowerToBoat and removeRowerFromBoat are invoked, with a similar mechanism.

The isUserAdequatelyCertified method inside the Boat class checks whether a user (the cox) is adequately certified to steer the given type of boat (so it checks the user's certifications, and compares it with the boat's allowed certifications – if at least one of the certificates match, it returns true, else false). For example, a cox with 4+ certificate can steer a C4 boat, while the opposite cannot occur.

User context microservice:

The rower micro-service in our project is responsible for allowing user to register as participant, organizer or both. A rower has three attributes: id, position, and certificate. Besides, a rower can accept a request or specify availability of the rower .Because of this, a RowerController class has 5 API gates: getId(), getPosition(), getCertificate(), accept(Request), specify(). The first 3 gates are designated for get the information of the player. The other two gates are designated for accepting the request or specifying the availability.



We have 3 interfaces: User, Participant and Organizer. A rower can be a participant, organizer, or can both. In the Participant interface, we have three methods: getId for returning the id of a participant, getPosition() for returning the position of the participant, and getCertificate for returning the certificate that the participant had. In the Organizer interface, we have one method accept(). This method will return the decision of the organizer whether he or she accepts this rower to join the competition or the training session.

Last, the Rower class implements both interface participant and organizer. This class implements all the methods in both interfaces. The UML diagram presents how the microservice works. Different methods in the Rower class will be called depending on the API accessed in the RowerController. If an organizer wants to accept a request to participate a training session, the accept() method in RowerController class will be called. If a rower want to specify an availability, the specify() method in RowerController class will be called. Two function addRower() and deleteRower() serve for adding a

rower when he or she register an account or request a deletion of an account.