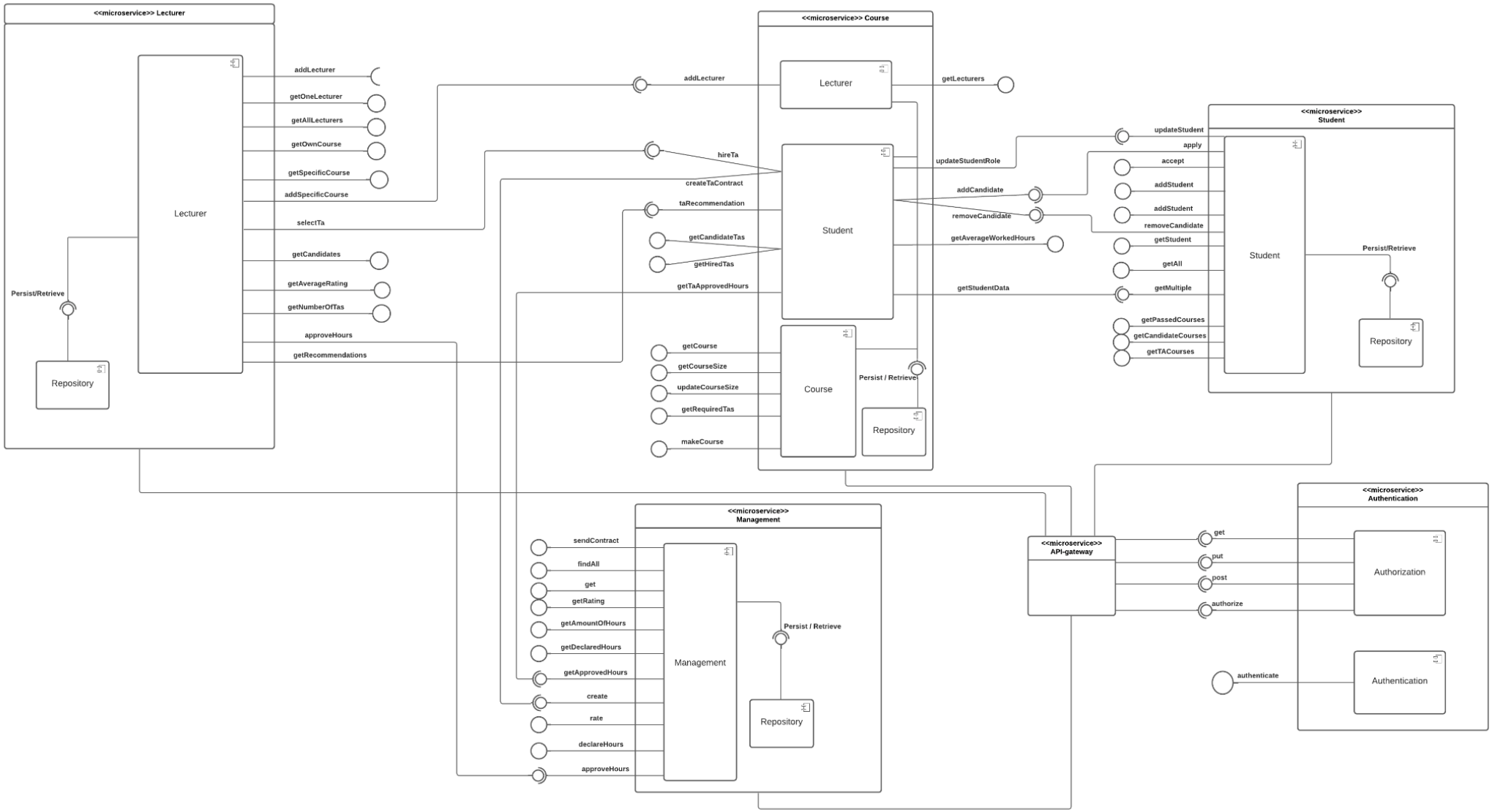
Software Architecture - group 15a

Project architecture



Introduction

Domain driven design (DDD) is a process and domain centered design type, which is compatible with microservices due to the detached architecture of a domain from other domains. Microservices improve scalability, ease testing and help with fault tolerance, which is why this type of an architecture is chosen for this project. The application is a TA Hiring System that has several components that communicate and exchange information, which consists of Authentication, Course, Lecturer, Student and Management components. Each of those 5 bounded contexts is mapped to a separate microservice.

There are several design choices made, while producing these components. For instance, Lecturer and Student have been separated and not used as a single User domain, because the attributes that these domains have and their functionality have significant differences. The Management domain is taking care of hours worked, rating of TAs and sending the contract to the students. The Course domain is responsible for course-related data and also for recommending candidate TAs. Finally, the Authentication domain authenticates credible users and allows them in the system.

Bounded contexts

Users are a core bounded context of the system. It is one of the most important contexts since the system revolves around various types of users interacting with courses through different roles. Users will be subdivided into a Student and Lecturer subtypes where each role interacts with User downstream dependencies Courses and Management differently with different privileges. Although each subrole has similarities such as name and netid, they also have differences as outlined above. As such we decided to map each subrole to a separate microservice in order to have higher cohesion.

Course is a bounded context, and it is one of the core domains. The system depends on a course and has no purpose without courses. Course has dependencies with other domains such as lecturer (who leads the course) and students (enrolled in the course). The size of the course depends on students, while the lecturer set depends on lecturers. Course is a separate domain as it provides the functionality of keeping data of courses.

Management is a separate bounded context that stores the related information for a TA for a course. For example, hours worked/declared/approved and the rating of the TA. It is also responsible for sending the contract to the chosen student to be a TA.

The authentication bounded context is a core domain that also corresponds to an independent microservice. It is unique from the others, in that it is the only one that has no dependencies on other bounded contexts, but is entirely responsible for serving other microservices. For this reason, compared to the other ones in the system, it is the most similar one to emulating a client-server design pattern.

Microservices

Course

As a microservice, the course is independent and has its own methods.

The responsibility of the service is

- To connect (candidate) TAs to one course
- Keep data of all courses
- Keep track of TAs for a course
- Keep capacity of a course
- Keep the number of needed TAs for a course
- To request average worked hours during a course
- Create, edit, remove courses (Admin only)

Course microservice dependencies:

- Endpoint for creating Courses (Admin only)

Each course has a unique id. All communication with other services will be done through the controllers.

Management

Management microservice responsibilities:

- Create a TA contract based on information provided
- Track declared, worked and approved hours of TAs
- Track the rating of a TA for a course
- Allow lecturers to approve/disapprove declared hours by TAs
- Send contracts to candidate TAs
- Send emails to candidate TAs who have been picked for the job

Management microservice dependencies:

- Lecturer creates and approves hours of TAs by interacting with the contract
- TAs declare their working hours

Authentication

The main tasks that the authentication microservice is responsible for are:

- Generate secure JWT tokens to different microservices whose instances wish to access other parts of the system and their databases
- Create a “server” that exists in the background and stores all information related to authentication and security
- Validate all requests to the server by determining the validity of the JWT tokens passed along with the requests
- Maintain different access permissions based on the role that each user has
- Provide endpoint for first time authentication (generate JWT)
- Provide endpoint for validating requests

Student

The responsibilities of the student microservice are to:

- Keep track of passed courses student is eligible to become a TA for and corresponding grade achieved
- Keep track of courses student has candidate themselves as TAs
- Keep track of courses student are/have TAed through Management objects
- Provide endpoint for average rating per TA: lecturer sets a rating, rating is stored in student database

Dependencies of student microservice are:

- Endpoint to declare worked hours
- Candidate student to a course
- Remove student candidate from course
- Get averaged workload for given course

Lecturer

The Lecturer microservice will have the following responsibilities:

- Keep track of the courses per lecturers.
- Get a list of recommended students to become Teaching Assistants based on their grades and rating
- Pick a Teaching Assistant for their course
- Approve working hours for current Teaching Assistants
- Rate the current Teaching Assistants

The Lecturer microservice dependencies:

- Create a Management object for a chosen TA with a fixed working hours for a given course

Design Patterns

Strategy Design Pattern

The project contained design patterns that enriched the end product by employing respected software development techniques into the structure of the program.

```
public interface TaRecommendationStrategy {
    public List<String> getRecommendedTas(Set<Student> candidateTas);
}
```

The first design pattern that was used was the Strategy Design pattern, found in the Course microservice. It was used in the context of allowing the user to specify on what basis they would like to receive a list of TA Recommendations. In particular, the family of algorithms was a sorting method by highest grade, most experience and highest rating.

When retrieving the list by highest rating, the Course microservice must query the corresponding ratings from the Management microservice (where they are stored) for each student. Past this, sorting is a fairly trivial matter.

```
public class RatingStrategy implements TaRecommendationStrategy {
    private transient Course course;
    private transient CommunicationService communicationService;

    public RatingStrategy(Course course, CommunicationService communicationService) {
        this.course = course;
        this.communicationService = communicationService;
    }

    /PMD/
    public List<String> getRecommendedTas(Set<Student> candidateTas) {
        Map<Student, Float> studentRatingMap =
                communicationService.getRatings(candidateTas, course.getCourseId());

        Comparator<Student> comparator = (Student s1, Student s2)
                -> studentRatingMap.get(s2) - studentRatingMap.get(s1) < 0 ? -1 : 1;
        return candidateTas.stream().sorted(comparator).map(s -> s.getNetId()).limit(10)
                .collect(Collectors.toList());
    }
}
```

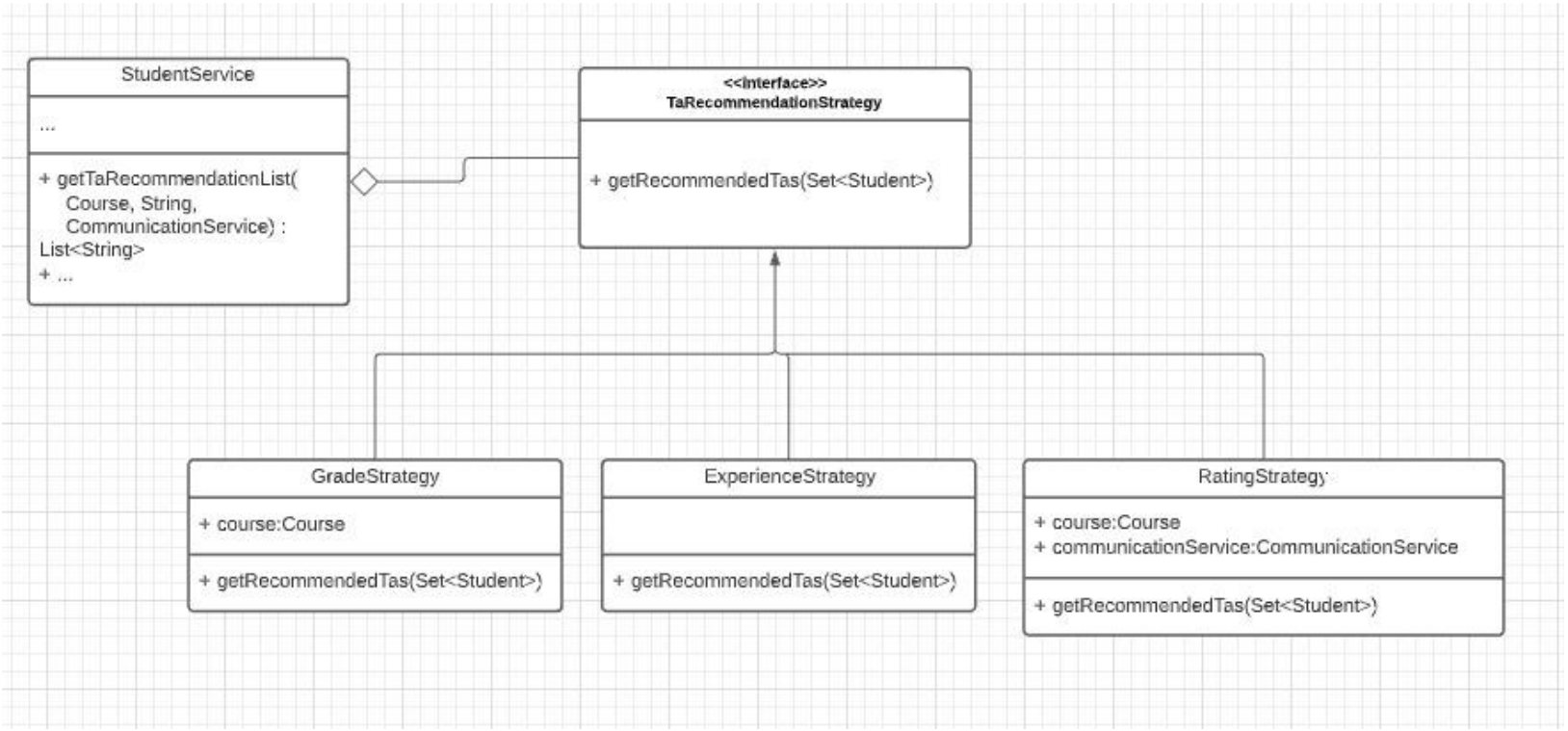
For sorting by grade, sorting is somewhat less simple. To do this, the algorithm must search for all the instances of a grade that a student has received in any edition of the

course and select the highest grade. The only stipulation that has been placed on this is that the grade is only valid if it is less than or equal to 5 years old.

```java
public class GradeStrategy implements TaRecommendationStrategy {
    private transient Course course;

    public GradeStrategy(Course course) { this.course = course; }

    @Override
    /PMD/
    public List<String> getRecommendedTas(Set<Student> candidateTas) {
        Comparator<Student> comparator = new Comparator<Student>() {
            @Override
            public int compare(Student o1, Student o2) {
                Float first = o1.getHighestGradeAchieved(course.getCourseId());
                Float second = o2.getHighestGradeAchieved(course.getCourseId());
                return  first > second ? -1 : 1;
            }
        };

        return candidateTas.stream().sorted(comparator).map(s -> s.getNetId()).limit(10)
                .collect(Collectors.toList());
    }
}
```

Finally, when sorting by experience, we retrieve a list of all the courses a candidate has TA'd for in the past, and simply sort by the size of this list. It is worth noting that in all of these strategies, there is no secondary ordering present. For example, if two potential TA's have the same highest grade in the course, one student will be arbitrarily ranked higher, and one lower. This is to emphasize that the user specifically selected that they wanted a TA Recommendation list ranked based on grades. Thus nothing else is taken into account.

```java
public class ExperienceStrategy implements TaRecommendationStrategy {

    @Override
    public List<String> getRecommendedTas(Set<Student> candidateTas) {
        Comparator<Student> comparator = (Student s1, Student s2)
                -> s2.getTaCourses().size() - s1.getTaCourses().size();
        return candidateTas.stream().sorted(comparator).map(s -> s.getNetId()).limit(10)
                .collect(Collectors.toList());
    }
}
```


StudentService
...
+ getTaRecommendationList(
Course, String,
CommunicationService) :
List<String>
+ ...
<<Interface>>
TaRecommendationStrategy
+ getRecommendedTas(Set<Student>)
GradeStrategy
+ course:Course
+ getRecommendedTas(Set<Student>)
ExperienceStrategy
+ getRecommendedTas(Set<Student>)
RatingStrategy
+ course:Course
+ communicationService:CommunicationService
+ getRecommendedTas(Set<Student>)

Chain of Responsibility pattern

The second design pattern implemented was the Chain of Responsibility design pattern, that exists in the Authentication microservice, within Security Configuration. It is used with the purpose of defining authorization levels for users, such as defining which type of users can access which of the microservices.

To implement this design pattern, we have created a Validator interface which holds the methods that have to be implemented by participating classes, and a BaseValidator class that implements the Validator interface to implement common methods that will be used among extending classes.

```
public interface Validator {

    void setNext(Validator handler);

    void handle(HttpSecurity http) throws Exception;
}
```

There are three classes that extend BaseValidator and therefore participate in the chain of responsibility, LoginValidator, AuthenticationRoleValidator and FilterValidator. In our application, we should first allow any kind of user, whether or not they have a JWT token or whether or not they have been authenticated, to log in, therefore it does not have any restrictions to connect to it. This is what the LoginValidator does, it makes sure that anyone can request to log in to the application.

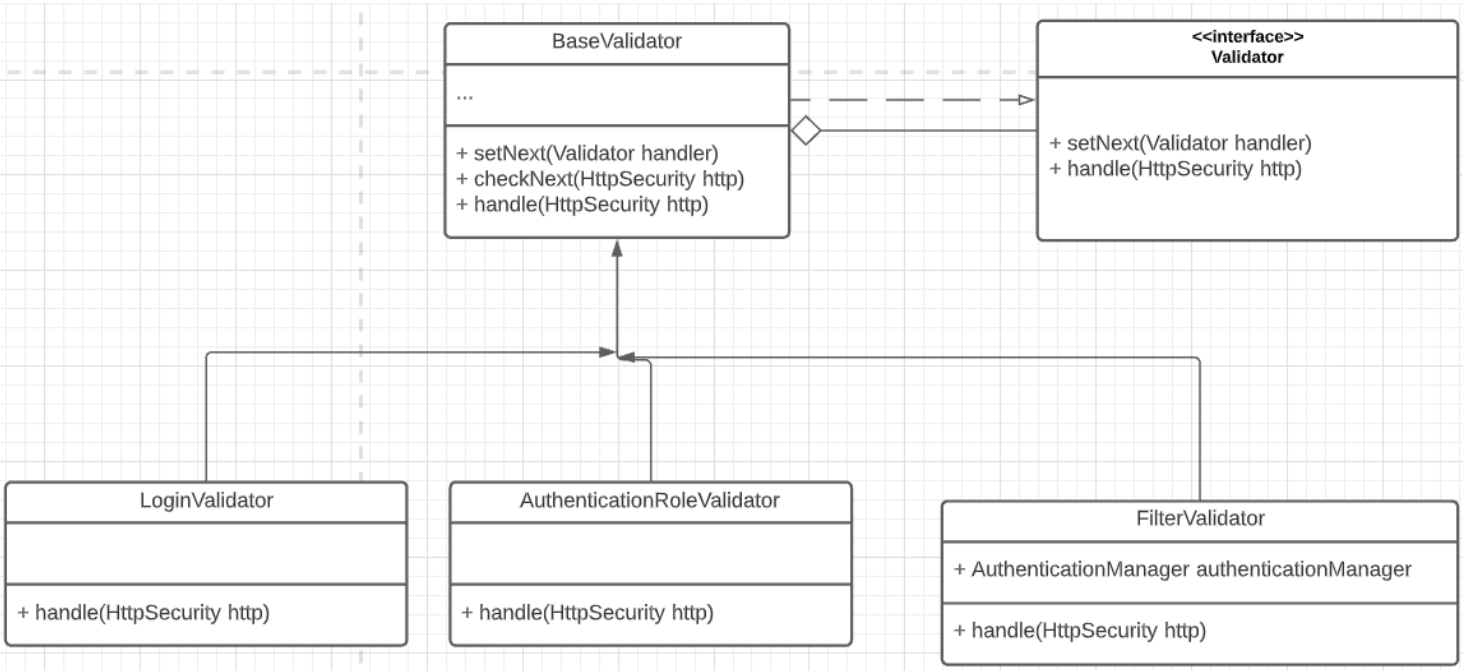
```
public abstract class BaseValidator implements Validator {

    private Validator next;

    public Validator getNext() { return next; }

    /**
     * set the next validator.
     *
     * @param handler the next element in the chain of responsibilities
     */
    @Override
    public void setNext(Validator handler) { this.next = handler; }

    /**
     * checks if there is another responsibility afterwards.
     *
     * @param http security config object
     * @throws Exception if next.handle goes wrong
     */
    protected void checkNext(HttpSecurity http) throws Exception {
        if (next != null) {
            next.handle(http);
        }
    }

    /**
     * will handle the responsibilities.
     *
     * @param http configures security
     * @throws Exception if exception does not work.
     */
    public abstract void handle(HttpSecurity http) throws Exception;
}
```

```
public class LoginValidator extends BaseValidator {

    /**
     * disable crfs and permit login to all users.
     *
     * @param http configures security
     */
    @Override
    public void handle(HttpSecurity http) throws Exception {
        http.csrf().disable();
        http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
        http.authorizeRequests().antMatchers( ...antPatterns: "/login").permitAll();
        super.checkNext(http);
    }
}
```

Furthermore, the second class in the chain of responsibility is AuthenticationRoleValidator. This class helps us define specific roles for users to have to access certain endpoints of our application. For example, only an admin can access “/courses/updateSize” and all students can access endpoints starting with “/student”.

Finally, we have FilterValidator, which defines the authentication and authorization filters for our application. Once we make sure the user has the credentials necessary to access an endpoint of our application, this class helps ensure that the filters for authentication (is the user logged in?) and authorization (does the user have a valid JWT token?) are created and applied to the request.

```java
public class FilterValidator extends BaseValidator {

    private AuthenticationManager authenticationManager;

    public FilterValidator(AuthenticationManager authenticationManager) {
        this.authenticationManager = authenticationManager;
    }

    /**
     * add the filters for authentication and authorization.
     *
     * @param http configures security
     */
    @Override
    public void handle(HttpSecurity http) {
        http.addFilter(new CustomAuthenticationFilter(authenticationManager));
        http.addFilterBefore(new CustomAuthorizationFilter(),
                UsernamePasswordAuthenticationFilter.class);
    }

    /**
     * getter for authentication manager.
     *
     * @return authenticationManager
     */
    public AuthenticationManager getAuthenticationManager() { return authenticationManager; }

    /**
     * setter for authenticationManager.
     *
     * @param authenticationManager new authenticationManager
     */
    public void setAuthenticationManager(AuthenticationManager authenticationManager) {
        this.authenticationManager = authenticationManager;
    }
}
```


BaseValidator
...
+ setNext(Validator handler)
+ checkNext(HttpSecurity http)
+ handle(HttpSecurity http)
<<interface>>
Validator
+ setNext(Validator handler)
+ handle(HttpSecurity http)
LoginValidator
+ handle(HttpSecurity http)
AuthenticationRoleValidator
+ handle(HttpSecurity http)
FilterValidator
+ AuthenticationManager authenticationManager
+ handle(HttpSecurity http)