Design Patterns

Chain of Responsibility - Matching microservice

Task 2.1

The main functionality of the ***"Matching" microservice*** is to match a user's availability with an activity based on some constraints. Thus, a **filtering** of the possible activities is needed and we decided that the optimal way to implement that is with the help of **the Chain of Responsibility pattern**.

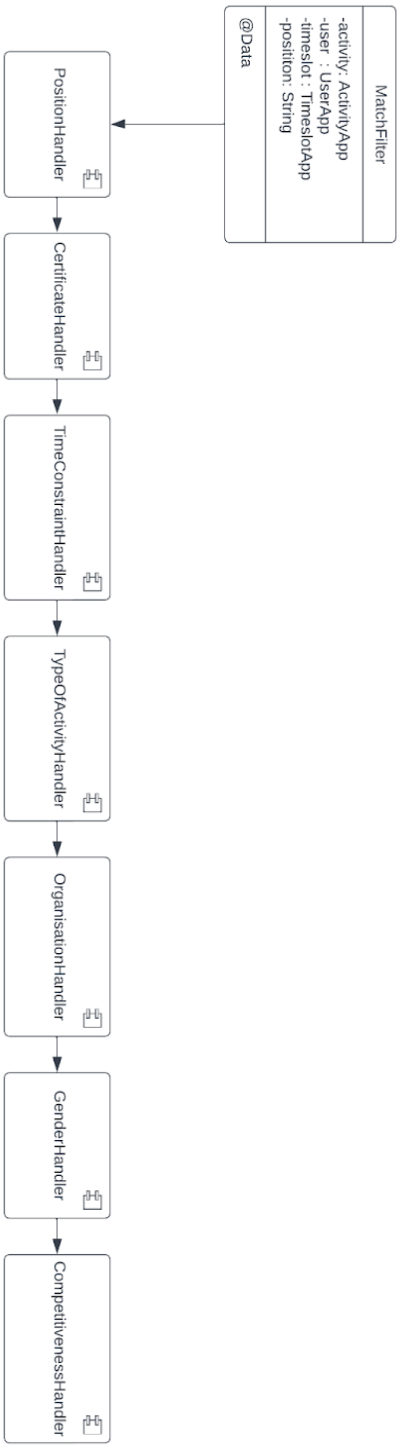
The **filtering process** is composed of multiple levels of validation for an Activity entity and this structure complies with the specification of this pattern. **The Chain of Responsibility** is a behavioral pattern that allows for the loose coupling between the filtering of activities and the actual constraints implemented that can change dynamically over time due to business logic rules that might be added. One might argue that the filtering of activities can be directly done through a series of conditional statements, however this approach is not adhering to the open close principles as having several if statements increases the complexity of the code and the difficulty of testing. That is why the use of handlers and a validation chain provided by this pattern increases maintainability and extendability whilst also taking into account the existence of two different types of activities (competition and training). From the multitude of other design patterns, this one fits the best since the filtering functionality works just like a chain with different constraints on each level.

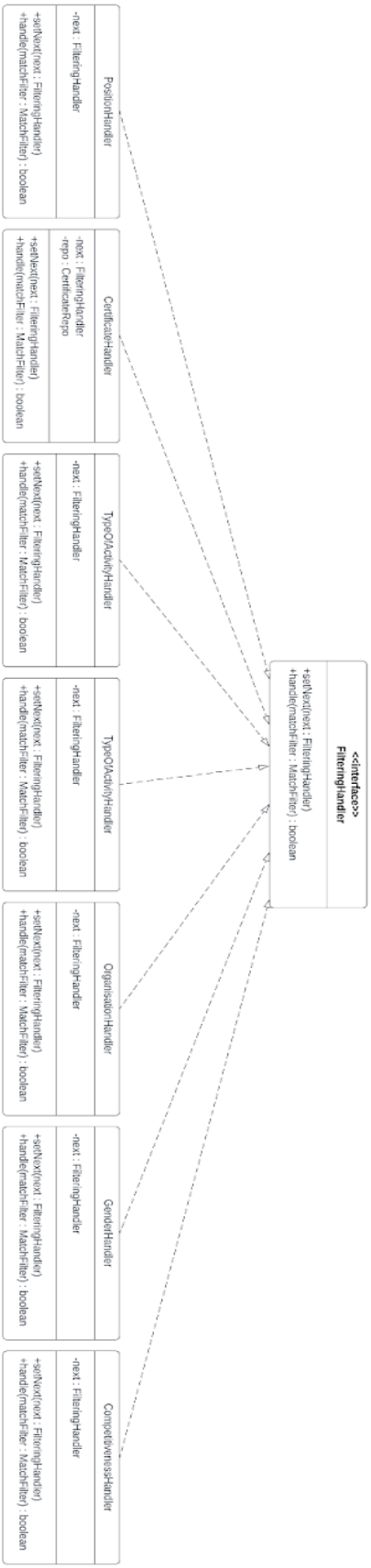
Starting from a list of activities (already in the specified availability) requested to the Activity micro service, we take each activity and the user's preferences and we put them through the filtering sequence. Concretely, we implemented a FilteringHandler interface that is meant to abstract the actual filters used in the pattern structure. This is used to construct the Chain of Responsibility in the following manner: **PositionHandler -> CertificateHandler -> TimeConstraintHandler -> TypeOfActivityHandler -> OrganisationHandler -> GenderHandler -> CompetitivenessHandler,** with all this handlers implementing the aforementioned one.

1. The **PositionHandler** - verify if the position provided by the user is present along the ones that the activity has available for filling.
2. The **CertificateHandler** - if the position in question is the "cox" position then the certificate of the user is compared with the required expertise one for the boat specific for the activity, for any other positions the filtering is passed along the next step in the chain.
3. The **TimeConstraintHandler** - handles the Could haves regarding time constraints for different type of activities, hence for Trainings we ensure that the match is not done in less than 30 minutes before the start of the activity and for Competition 1 day beforehand, respectively.
4. The **TypeOfActivityHandler** - this step takes into consideration the existence of two types of activities and in the case of a Training the chain is stopped and it returns true, whilst for the Competition we continue on filtering on various constraints enforced by the scenario.
5. The **OrganisationHandler -** verifies if the organisation that the user is part of matches the one that the competition requires.
6. The **GenderHandler** - for competitions it further checks if the participants have the same gender, thus not allowing a user to take part in an activity that is meant for a different gender.
7. The **CompetitivenessHandler** - not allowing amateur users to take part in expert competitions.

Note that all these handlers make use of a handle function that returns a boolean representing whether the activity is a match. Thus, if one of the constraints cannot be satisfied the chain is stopped, the handle function returns false and the activity is discarded from the list of possible matches.

Task 2.2
Class Diagrams




<<interface>>
FilteringHandler
+setNext(next : FilteringHandler)
+handle(matchFilter : MatchFilter) : boolean
PositionHandler
-next : FilteringHandler
+setNext(next : FilteringHandler)
+handle(matchFilter : MatchFilter) : boolean
CertificateHandler
-next : FilteringHandler
-repo : CertificateRepo
+setNext(next : FilteringHandler)
+handle(matchFilter : MatchFilter) : boolean
TypeOfActivityHandler
-next : FilteringHandler
+setNext(next : FilteringHandler)
+handle(matchFilter : MatchFilter) : boolean
TypeOfActivityHandler
-next : FilteringHandler
+setNext(next : FilteringHandler)
+handle(matchFilter : MatchFilter) : boolean
OrganisationHandler
-next : FilteringHandler
+setNext(next : FilteringHandler)
+handle(matchFilter : MatchFilter) : boolean
GenderHandler
-next : FilteringHandler
+setNext(next : FilteringHandler)
+handle(matchFilter : MatchFilter) : boolean
CompetitivenessHandler
-next : FilteringHandler
+setNext(next : FilteringHandler)
+handle(matchFilter : MatchFilter) : boolean

Task 2.3

The **Chain of Responsibility pattern** is constructed using the **filteringHandlerSetUp()** function that is called inside the **MatchingService** **@Autowired** constructor.

```java
/**
 * Function for setting up the Chain of Responsibility pattern implemented for filtering.
 */
1 usage  Micloiu Diana +1
public final void filteringHandlerSetUp() {
    this.filteringHandler = new PositionHandler();
    FilteringHandler certificateHandler = new CertificateHandler(certificateRepo);
    this.filteringHandler.setNext(certificateHandler);
    FilteringHandler timeConstraintHandler = new TimeConstraintHandler();
    certificateHandler.setNext(timeConstraintHandler);
    FilteringHandler typeOfActivityHandler = new TypeOfActivityHandler();
    timeConstraintHandler.setNext(typeOfActivityHandler);
    FilteringHandler organisationHandler = new OrganisationHandler();
    typeOfActivityHandler.setNext(organisationHandler);
    FilteringHandler genderHandler = new GenderHandler();
    organisationHandler.setNext(genderHandler);
    FilteringHandler competitivenessHandler = new CompetitivenessHandler();
    genderHandler.setNext(competitivenessHandler);

}
```

This is the FilteringHandler interface used for constructing the pattern

```java
public interface FilteringHandler {

    public void setNext(FilteringHandler handler);

    public boolean handle(MatchFilter matchFilter);
}
```

This is where the **Chain of Responsibility** starts for each activity and user's preferences encapsulated in the **MatchFilter** value object.

```java
Method for filtering the activities based on different constraints in order to match a user.
Params: activities – the activities given by the Activity microservice
        user – the user requesting activities
        position – teh position they can fill in
Returns: the positions the user is matched with
public List<ActivityReponse> filterActivities(List<ActivityApp> activities,
                                              TimeslotApp timeslot,
                                              UserApp user,
                                              String position) {
    return activities.stream().filter(a -> this.filteringHandler.handle(new MatchFilter(a, user, position, timeslot))) Stream<ActivityApp>
            .map(a -> matchUserToActivity(user, position, a)) Stream<ActivityReponse>
            .collect(Collectors.toList());
}
```

This is the **MatchFilter** value object that is passed along the chain in order to filter the activities by constraints.

```
@Data
@AllArgsConstructor
@EqualsAndHashCode
public class MatchFilter {
    private ActivityApp activityApp;
    private UserApp user;
    private String position;
    private TimeslotApp timeslot;
}
```

PositionHandler

```
public class PositionHandler implements FilteringHandler {

    private transient FilteringHandler next;

    Micloiu Diana
    @Override
    public void setNext(FilteringHandler handler) { this.next = handler; }

    Micloiu Diana
    @Override
    public boolean handle(MatchFilter matchFilter) {
        if (matchFilter.getActivityApp().getPositions().containsKey(matchFilter.getPosition())) {
            if (next != null) {
                return next.handle(matchFilter);
            } else {
                return true;
            }
        }
        return false;
    }
}
```

TimeConstraintHandler

```java
public class TimeConstraintHandler implements FilteringHandler {

    private transient FilteringHandler next;

    Lucian Tosa
    @Override
    public void setNext(FilteringHandler handler) { this.next = next; }

    Micloiu Diana +1
    @Override
    public boolean handle(MatchFilter matchFilter) {
        switch (matchFilter.getActivityApp().getType()) {
            case TRAINING: {
                if (LocalDateTime.now().plusMinutes(30)
                        .isBefore(matchFilter.getActivityApp().getTimeslot().getStartTime())) {
                    if (next != null) {
                        return next.handle(matchFilter);
                    } else {
                        return true;
                    }
                } else {
                    return false;
                }
            }
            case COMPETITION: {
                if (LocalDateTime.now().plusDays(1)
                        .isBefore(matchFilter.getActivityApp().getTimeslot().getStartTime())) {
                    if (next != null) {
                        return next.handle(matchFilter);
                    } else {
                        return true;
                    }
                } else {
                    return false;
                }
            }
            default:
                return false;
        }
    }
```

TypeOfActivityHandler

```java
public class TypeOfActivityHandler implements FilteringHandler {

    private transient FilteringHandler next;

    Micloiu Diana
    @Override
    public void setNext(FilteringHandler handler) { this.next = handler; }

    Micloiu Diana
    @Override
    public boolean handle(MatchFilter matchFilter) {
        if (matchFilter.getActivityApp().getType() == TypeOfActivity.TRAINING) {
            return true;
        }

        if (next != null) {
            return next.handle(matchFilter);
        }

        return false;
    }
}
```

OrganisationHandler

```java
public class OrganisationHandler implements FilteringHandler {

    private transient FilteringHandler next;

    Micloiu Diana
    @Override
    public void setNext(FilteringHandler handler) { this.next = handler; }

    Micloiu Diana
    @Override
    public boolean handle(MatchFilter matchFilter) {
        if (matchFilter.getActivityApp().getOrganisation().equals(matchFilter.getUser().getOrganisation())) {
            if (next != null) {
                return next.handle(matchFilter);
            } else {
                return true;
            }

        }
        return false;
    }
}
```

GenderHandler

```java
public class GenderHandler implements FilteringHandler {

    private transient FilteringHandler next;

    Micloiu Diana
    @Override
    public void setNext(FilteringHandler handler) { this.next = handler; }

    Micloiu Diana
    @Override
    public boolean handle(MatchFilter matchFilter) {
        if (matchFilter.getActivityApp().getGender().equals(matchFilter.getUser().getGender())) {
            if (next != null) {
                return next.handle(matchFilter);
            } else {
                return true;
            }

        }
        return false;
    }
}
```

CompetitivenessHandler

```java
public class CompetitivenessHandler implements FilteringHandler {

    private transient FilteringHandler next;

    Micloiu Diana
    @Override
    public void setNext(FilteringHandler handler) { this.next = handler; }

    Micloiu Diana +1
    @Override
    public boolean handle(MatchFilter matchFilter) {
        if (!matchFilter.getActivityApp().isCompetition() || matchFilter.getUser().isCompetitive()) {
            if (next != null) {
                return next.handle(matchFilter);
            } else {
                return true;
            }

        }
        return false;
    }
}
```

Builder - Notification microservice

Task 2.1

The ***"Notification" microservice*** has an implementation which respects the Builder design pattern, as this service mainly focuses on the **construction of a complex object** with many possible representations, the notification, and on its propagation through email towards users.

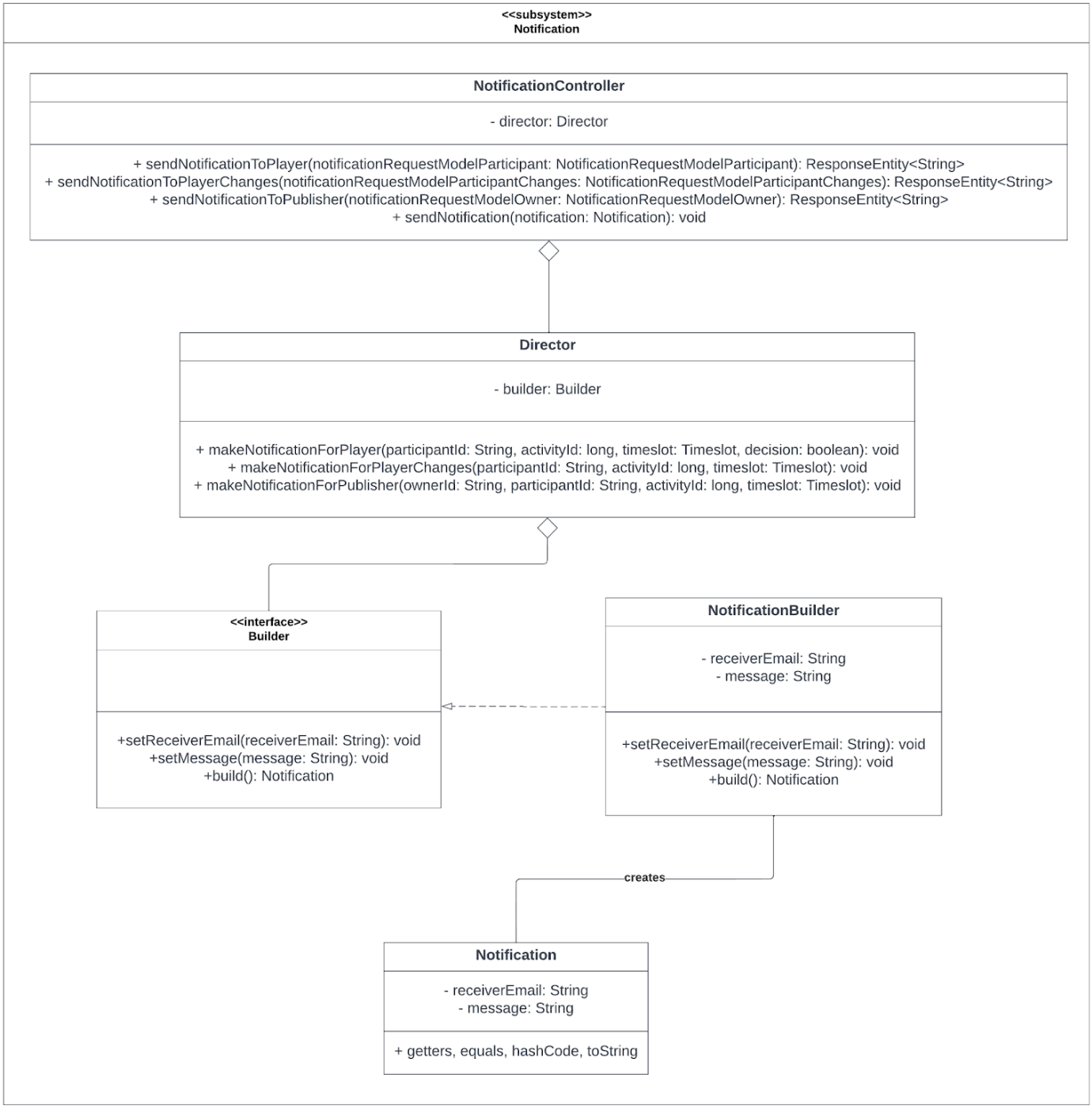
Consider the construction of a notification. A notification is the final end product that is to be returned as the output of the construction process and sent towards a user. As of now, the building part has two steps: setting the receiver and constructing the message, methods defined in the **Builder** interface and implemented in the **NotificationBuilder**. Using the same process, one can build notifications with different types of messages for the scenarios described in the "Architecture" section, paragraph "Notifications Microservice".

In order to create and send a specific type of notification, there is a **Director** who calls its **makeNotification** method implemented for that notification in particular (there are currently three methods of this type), then calls the **build** method on its own builder to generate the final product, and finally sends the notification by calling its **sendNotification** method.

The greatest advantage of this design pattern is that it makes the notification construction process independent of the parts that it consists of (receiver and message) and how they're assembled: only their internal values count as differences. As a follow-up advantage, this pattern handles abstraction and complexity well, because the same construction process can be used for any representation of a notification.

Another idea of design pattern for this microservice would have been the "Abstract Factory", since it handles families of complex objects properly and provides a fine, abstract, construction process. The reason we did not choose to implement it is the following: currently our application's goal is to send notifications in a single "family" format - email. If it was necessary to send notifications in another format as well, for example SMS, then the "Abstract Factory" could show its main purpose; for now it is not the case.

Task 2.2
Class Diagram



Task 2.3

According to the API request that is made, a new notification builder is created and the director of the NotificationController is created with that builder. The director calls the appropriate makeNotification method, then the notification is being built and sent:

```
/**
 * Sends email to publisher of an activity when a user wants to sign up for it in a certain timeslot.
 *
 * @param notificationRequestModelOwner request body format
 * @return if email is sent successfully returns 200 OK,
 *      otherwise 422 Unprocessable Entity and the exception message
 */
@PostMapping(value = "/publisher",
            consumes = {MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE},
            produces = {MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
public ResponseEntity<String> sendNotificationToPublisher(
            @RequestBody NotificationRequestModelOwner notificationRequestModelOwner) {
    try {
        Builder builder = new NotificationBuilder();
        director = new Director(builder);
        director.makeNotificationForPublisher(notificationRequestModelOwner.getOwnerId(),
                notificationRequestModelOwner.getParticipantId(),
                notificationRequestModelOwner.getActivityId(),
                notificationRequestModelOwner.getTimeslot());
        sendNotification(builder.build());
        return ResponseEntity.ok("Email sent successfully.");
    } catch (MailException mailException) {
        return ResponseEntity.unprocessableEntity().body(mailException.getMessage());
    }
}
```

```java
/**
 * Sends a notification through email to a player when he receives a decision from the owner of
 * the activity he signed up for.
 *
 * @param notificationRequestModelParticipant the request body format
 * @return if email is sent successfully returns 200 OK,
 *          otherwise 422 Unprocessable Entity and the exception message
 */
@PostMapping(value = "/participant",
            consumes = {MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE},
            produces = {MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
public ResponseEntity<String> sendNotificationToPlayer(
            @RequestBody NotificationRequestModelParticipant notificationRequestModelParticipant) {
    try {
        Builder builder = new NotificationBuilder();
        director = new Director(builder);
        director.makeNotificationForPlayer(notificationRequestModelParticipant.getParticipantId(),
                notificationRequestModelParticipant.getActivityId(),
                notificationRequestModelParticipant.getTimeslot(),
                notificationRequestModelParticipant.isDecision());
        sendNotification(builder.build());
        return ResponseEntity.ok("Email sent successfully.");
    } catch (Exception exception) {
        return ResponseEntity.badRequest().body(exception.getMessage());
    }
}
```

```java
/**
 * Sends a notification through email to a player when the activity he was approved for
 * gets deleted or if its details are changed.
 *
 * @param notificationRequestModelParticipantChanges the request body format
 * @return if email is sent successfully returns 200 OK,
 *      otherwise 422 Unprocessable Entity and the exception message
 */
@PostMapping(value = "/activity-changed",
            consumes = {MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE},
            produces = {MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
public ResponseEntity<String> sendNotificationToPlayerChanges(
            @RequestBody NotificationRequestModelParticipantChanges notificationRequestModelParticipantChanges)
    try {
        Builder builder = new NotificationBuilder();
        director = new Director(builder);
        director.makeNotificationForPlayerChanges(notificationRequestModelParticipantChanges.getParticipantId(),
                notificationRequestModelParticipantChanges.getActivityId(),
                notificationRequestModelParticipantChanges.getTimeslot());
        sendNotification(builder.build());
        return ResponseEntity.ok("Email sent successfully.");
    } catch (Exception exception) {
        return ResponseEntity.badRequest().body(exception.getMessage());
    }
}
```

The director corresponding to the builder design pattern decides which scenario to use when building concrete Notification object:

```java
/**
 * The director corresponding to the builder design pattern, it decides which scenario to use
 * when building a concrete Notification object.
 */
@AllArgsConstructor
public class Director {
    private transient Builder builder;

    /**
     * Sets values of the builder when constructing a notification for the player
     * in case he has a response from the owner of the activity he signed up for.
     *
     * @param participantId email of the participant
     * @param activityId id of the activity
     * @param timeslot start and end time of the activity
     * @param decision decision from activity owner
     */
    public void makeNotificationForPlayer(String participantId, long activityId,
                                          Timeslot timeslot, boolean decision) {
        String message;
        String dash = " - ";

        if (decision) {
            message = "Congratulations! You have been accepted for activity " + activityId
                    + ". You are expected to be there between " + timeslot.getStartTime().toString()
                    + dash + timeslot.getEndTime().toString() + ".";
        } else {
            message = "Unfortunately, you have been denied for activity " + activityId
                    + ", happening between " + timeslot.getStartTime().toString()
                    + dash + timeslot.getEndTime().toString()
                    + ". We advise you to not give up and try another timeslot or activity.";
        }

        builder.setReceiverEmail(participantId);
        builder.setMessage(message);
    }
```

```
     * Sets values of the builder when constructing a notification for the player
     * in case the activity he signed up for gets cancelled or its details are changed.
     *
     * @param participantId email of the participant
     * @param activityId id of the activity
     * @param timeslot start and end time of the activity
     */
    public void makeNotificationForPlayerChanges(String participantId, long activityId,
                                         Timeslot timeslot) {
        String dash = " - ";

        builder.setReceiverEmail(participantId);
        builder.setMessage("Unfortunately, the details for activity " + activityId
                + ", happening between " + timeslot.getStartTime().toString()
                + dash + timeslot.getEndTime().toString()
                + "have been changed and you have been unenrolled. "
                + "We advise you to try another timeslot or activity.");
    }

    /**
     * Sets values of the builder when constructing a notification for the owner
     * of an activity in case a player requested to sign up for it.
     *
     * @param ownerId email of the owner
     * @param participantId email of the participant
     * @param activityId id of the activity
     * @param timeslot start and end time of the activity
     */
    public void makeNotificationForPublisher(String ownerId, String participantId,
                                             long activityId, Timeslot timeslot) {
        String dash = " - ";

        builder.setReceiverEmail(ownerId);
        builder.setMessage("You have a new request: user "
                + participantId + " wants to participate in activity " + activityId
                + " between " + timeslot.getStartTime().toString() + dash
                + timeslot.getEndTime().toString() + ". Please decide as soon as possible"
                + " whether you accept this request or not.");
    }
```

These are the Builder interface and the concrete **NotificationBuilder class:**

```java
/**
 * Builder interface to be implemented by any concrete builder class.
 */
public interface Builder {
    /**
     * Setter for receiver email.
     *
     * @param receiverEmail email of the person who receives the notification
     */
    public void setReceiverEmail(String receiverEmail);

    /**
     * Setter for message.
     *
     * @param message the body of the notification
     */
    public void setMessage(String message);

    /**
     * Build method specific to this design pattern.
     *
     * @return a new Notification object
     */
    public Notification build();
}
```

```java
/**
 * NotificationBuilder is a concrete builder for notifications.
 */
public class NotificationBuilder implements Builder {
    private transient String receiverEmail;
    private transient String message;

    /**
     * Setter for receiver email.
     *
     * @param receiverEmail email of the person who receives the notification
     */
    @Override
    public void setReceiverEmail(String receiverEmail) { this.receiverEmail = receiverEmail; }

    /**
     * Setter for message.
     *
     * @param message the body of the notification
     */
    @Override
    public void setMessage(String message) { this.message = message; }

    /**
     * Build method specific to this design pattern.
     *
     * @return a new Notification object
     */
    @Override
    public Notification build() { return new Notification(receiverEmail, message); }
}
```

And this is the **Notification object**:

```java
/**
 * Notification object containing the email address of the receiver and its body (message).
 */
@AllArgsConstructor
public class Notification {
    private transient String receiverEmail;
    private transient String message;
```

Chain of Responsibility - Authentication microservice

Task 2.1

The authentication service is implemented using **the Chain of Responsibility design pattern**. In simple words, this pattern splits a large task into distinct subtasks which are assigned to individual components called handlers. These handlers are linked together in a chain where each one calls the next after it finishes its task.

Workings

The ***"Authentication" microservice*** has two main tasks: allowing clients to authenticate themselves so they can use the other microservices and facilitating the registration of new users. To perform these tasks the microservice uses four handlers. These are constructed (using the ChainCreator) by the Controller class that handles the incoming requests.

When an authentication request comes in, the **SanitizeCredentials handler** is called and the client's credentials are passed to it. This first handler sanitises the credentials and confirms that the userId is an email-address. It also hashes the password. If no errors are reported, the first handler calls the next in the chain and again passes it the credentials. The second handler is called VerifyCredentials. It does what the name suggests and checks if the supplied credentials match the credentials in the database. When the credentials are verified, the last handler is called. The CreateToken handler takes the now verified credentials and creates a JSON Web Token (JWT). This token is what the client can use to easily authenticate themselves at the other microservices. The last handler doesn't call any other handler, but instead stores the created JWT as a class variable. When the chain is complete, the Controller can access the JWT from the CreateToken handler and send it to the client.

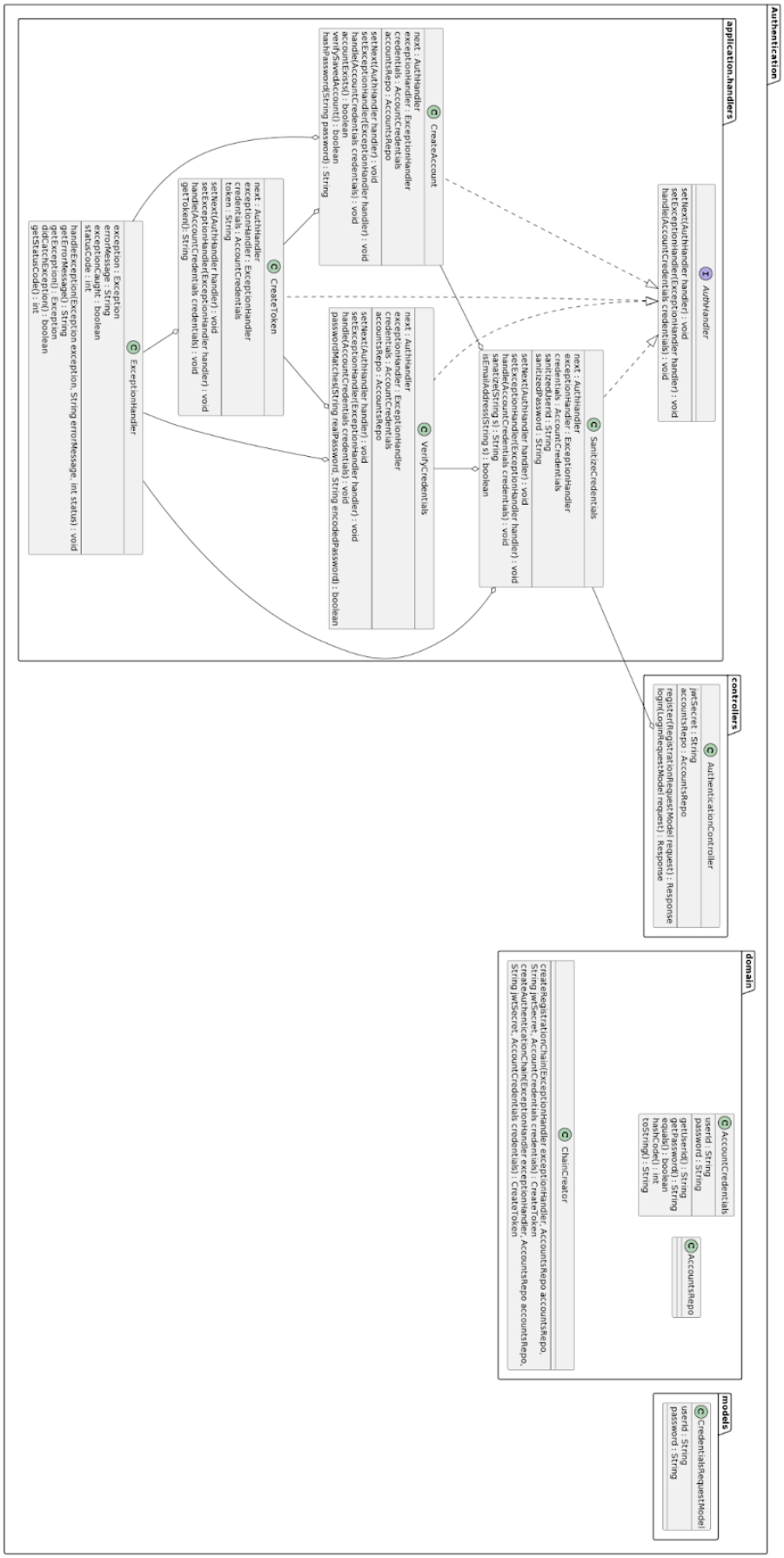
The procedure for the registration is very similar. The Controller once again constructs the chain of handlers. The only difference compared to authentication is in the second handler. Instead of verifying the credentials, a handler called CreateAccount is used. This handler checks if an account with the supplied userId doesn't already exist and adds a new account. The first and last handlers in this chain are the same as before: the credentials are sanitized and at the end a JWT is created. The Controller again completes the request by sending the client the created token.

Additionally, these two chains use an extra handler called the ExceptionHandler. This handler is different from the other ones, because it is only called when an exception or mistake occurs. The ExceptionHandler is set in all the handlers in the chain. When an exception occurs in a handler, it reports the exception to the ExceptionHanler and breaks the chain by stopping its execution. This procedure is used for genuine Java exceptions, but also for events that cause the chain's task to fail. For example, when the client supplied a username that is not an email-address when registering or an incorrect password when authenticating. In such cases, the handlers can also set a custom error message and HTTP status code (400 or 500) in the ExceptionHandler. When the execution of the chain has ended, the Controller first checks the ExceptionHandler to see if an exception has occurred. If this is the case, it gets the error message and status code from the handler and sends this information back to the client. Only when the ExceptionHandler hasn't detected an error, the Controller gets the JWT and completes the request with success.

Advantages

The implementation with the chain of responsibility pattern has some great advantages: an authentication process always involves different steps that can easily be split into handlers. This allows for clean code where each class has one clear goal. Furthermore, exceptions can be handled cleanly without disrupting the operation of the entire service. A large plus is that code can be easily reused. In the implementation, two of the handlers are used in both the registering and authenticating chains. But the biggest advantage is scalability. This pattern makes it incredibly easy to insert new handlers into the chain or to swap out existing ones. Our relatively simple authentication service could be easily upgraded by creating new handlers and adding them to the chains. It is also very easy to create new different chains using the existing handlers.

Task 2.2


Authentication
application.handlers
AuthHandler
setNext(AuthHandler handler) : void
setExceptionHandler(ExceptionHandler handler) : void
handle(AccountCredentials credentials) : void
CreateAccount
next : AuthHandler
exceptionHandler : ExceptionHandler
credentials : AccountCredentials
accountsRepo : AccountsRepo
setNext(AuthHandler handler) : void
setExceptionHandler(ExceptionHandler handler) : void
handle(AccountCredentials credentials) : void
accountExists() : boolean
verifySwedAccount() : boolean
hashPassword(String password) : String
SanitizeCredentials
next : AuthHandler
exceptionHandler : ExceptionHandler
credentials : AccountCredentials
sanitizedUserId : String
sanitizedPassword : String
setNext(AuthHandler handler) : void
setExceptionHandler(ExceptionHandler handler) : void
handle(AccountCredentials credentials) : void
sanitize(String s) : String
isEmailAddress(String s) : boolean
CreateToken
next : AuthHandler
exceptionHandler : ExceptionHandler
credentials : AccountCredentials
token : String
setNext(AuthHandler handler) : void
setExceptionHandler(ExceptionHandler handler) : void
handle(AccountCredentials credentials) : void
getToken() : String
VerifyCredentials
next : AuthHandler
exceptionHandler : ExceptionHandler
credentials : AccountCredentials
accountsRepo : AccountsRepo
setNext(AuthHandler handler) : void
setExceptionHandler(ExceptionHandler handler) : void
handle(AccountCredentials credentials) : void
passwordMatches(String realPassword, String encodedPassword) : boolean
ExceptionHandler
exception : Exception
errorMessage : String
exceptionMessage : String
statusCode : int
handleException(Exception exception, String errorMessage, int status) : void
getErrorMessage() : String
getException() : Exception
didCatchException() : boolean
getStatusCode() : int
controllers
AuthenticationController
jwtSecret : String
accountsRepo : AccountsRepo
register(RegistrationRequestModel request) : Response
login(LoginRequestModel request) : Response
domain
AccountCredentials
userId : String
password : String
getUserId() : String
getPassword() : String
equals() : boolean
hashCode() : int
toString() : String
AccountsRepo
ChainCreator
createRegistrationChain(ExceptionHandler exceptionHandler, AccountsRepo accountsRepo, String jwtSecret)
createAuthenticationChain(ExceptionHandler exceptionHandler, AccountsRepo accountsRepo, String jwtSecret, AccountCredentials credentials) : CreateToken
models
CredentialsRequestModel
userId : String
password : String

Task 2.3

The handlers in a chain are all an implementation of the **AuthHandler interface**.

```
Interface for Handlers in the Authentication chain of responsibility.

21 usages  4 implementations  Mitchell van Pelt
public interface AuthHandler {

    Main method in each handler. The method performs all the tasks the handler has to deal with. When the
    method is completed with success, it calls the handle method on the next handler with the credentials.
    When the method cannot complete its task, it calls the ExceptionHandler and aborts.
    Params: credentials – The client's credentials needed for the authentication process.

    4 implementations  Mitchell van Pelt
    public void handle(AccountCredentials credentials);

    Sets the handler that should be called after at the end of the handle method. Must be set before handle()
    is called in implementations that are not the last in the chain.
    Params: handler – The next handler in the chain.

    4 implementations  Mitchell van Pelt
    public void setNext(AuthHandler handler);

    Sets the ExceptionHandler that is called when the handle method encounters an error. Must always be set
    before handle() is called.
    Params: handler – The ExceptionHandler used in the chain.

    4 implementations  Mitchell van Pelt
    public void setExceptionHandler(ExceptionHandler handler);
}
```

The four main handlers that were used:

SanitizeCredentials

```
public SanitizeCredentials() {}

    Checks if the credentials are complete and don't contain any illegal characters. Also hashes the password.
    This method will fail if the credentials are incomplete or contain illegal characters, or if the userId is not an
    email-address. The method will abort if the next handler or ExceptionHandler are not set.
    Params: credentials – The client's credentials that will be sanitized.
    Mitchell van Pelt
@Override
public void handle(AccountCredentials credentials) {
    if (next == null || exceptionHandler == null) {
        return;
    }
    this.credentials = credentials;
    try {
        if (credentials.getUserId() == null || credentials.getUserId().equals("")) {
            exceptionHandler.handleException(new IllegalArgumentException(), "Please provide a user id.", 400);
            return;
        }
        if (credentials.getPassword() == null || credentials.getPassword().equals("")) {
            exceptionHandler.handleException(new IllegalArgumentException(), "Please provide a password.", 400);
            return;
        }
        sanitizedUserId = sanitize(credentials.getUserId());
        sanitizedPassword = sanitize(credentials.getPassword());
        if (sanitizedUserId.length() < credentials.getUserId().length()) {
            exceptionHandler.handleException(new IllegalArgumentException(), "Your user id contains illegal"
                    + " characters. Please only use letters, numbers and the following characters: "
                    + "!#$%&()*+,-./:;<=>?@^_`{|}~", 400);
            return;
        }
        if (sanitizedPassword.length() < credentials.getPassword().length()) {
            exceptionHandler.handleException(new IllegalArgumentException(), "Your password contains illegal"
                    + " characters. Please only use letters, numbers and the following characters: "
                    + "!#$%&()*+,-./:;<=>?@^_`{|}~", 400);
            return;
        }
        if (!isEmailAddress(sanitizedUserId)) {
            exceptionHandler.handleException(new IllegalArgumentException(), "Your user id must be an email address.",
                    400);
            return;
        }
        AccountCredentials newCredentials = new AccountCredentials(sanitizedUserId, sanitizedPassword);
        next.handle(newCredentials);
    } catch (Exception e) {
        exceptionHandler.handleException(e);
    }
}
```

CreateAccount

```java
public class CreateAccount implements AuthHandler {

    3 usages
    private transient AuthHandler next;
    5 usages
    private transient ExceptionHandler exceptionHandler;
    3 usages
    private transient AccountCredentials credentials;
    4 usages
    private transient AccountsRepo accountsRepo;

    Constructs a CreateAccount handler. Handles the creation of new accounts.
    Params: accountsRepo – The repository where the accounts are saved.

    3 usages  Mitchell van Pelt
    public CreateAccount(AccountsRepo accountsRepo) { this.accountsRepo = accountsRepo; }

    Creates an account and saves it in the accounts' repository. This method will fail if an account with the
    supplied userId already exists. The method will abort if the next handler or ExceptionHandler are not set.
    Params: credentials – The client's credentials (with a hashed password) to be saved in the database.

    Mitchell van Pelt
    @Override
    public void handle(AccountCredentials credentials) {
        if (next == null || exceptionHandler == null) {
            return;
        }
        this.credentials = credentials;
        try {
            if (accountExists()) {
                exceptionHandler.handleException(new SQLException(), "An account with this user id already exists."
                        + " Please choose a different user id.", 400);
                return;
            }
            String hashedPassword = hashPassword(credentials.getPassword());
            AccountCredentials hashedCredentials = new AccountCredentials(credentials.getUserId(), hashedPassword);
            accountsRepo.save(hashedCredentials);
            if (!verifySavedAccount(credentials.getPassword())) {
                exceptionHandler.handleException(new SQLException(), "There was an error while saving your account."
                        + " Please try again later", 500);
                return;
            }
            next.handle(credentials);
        } catch (Exception e) {
            exceptionHandler.handleException(e);
        }
    }

    Sets the handler that should be called after at the end of the handle method. Must be set before handle()
    is called.
    Params: next – The next handler in the chain.

    Mitchell van Pelt
    @Override
    public void setNext(AuthHandler next) { this.next = next; }
```

VerifyCredentials:

```
Constructs a VerifyCredentials handler.
Params: accountsRepo – The repository the accounts are stored in.
3 usages  Mitchell van Pelt
public VerifyCredentials(AccountsRepo accountsRepo) { this.accountsRepo = accountsRepo; }

Retrieves known credentials from the database and matches them with the supplied credentials. This
method will fail if the supplied credentials do not match any credentials found in the database. The
method aborts if no next handler or ExceptionHandler has been set.
Params: credentials – The client's credentials to be verified.
Mitchell van Pelt
@Override
public void handle(AccountCredentials credentials) {
    if (next == null || exceptionHandler == null) {
        return;
    }
    this.credentials = credentials;
    try {
        Optional<AccountCredentials> foundAccount = accountsRepo.findById(credentials.getUserId());
        if (foundAccount.isEmpty()) {
            exceptionHandler.handleException(new SQLException(), "UserId or password incorrect. Please try again.",
                    401);
            return;
        }
        AccountCredentials foundCredentials = foundAccount.get();
        if (!foundCredentials.getUserId().equals(credentials.getUserId())
                || !passwordMatches(credentials.getPassword(), foundCredentials.getPassword())) {
            exceptionHandler.handleException(new SQLException(), "UserId or password incorrect. Please try again.",
                    401);
            return;
        }
        next.handle(credentials);
    } catch (Exception e) {
        exceptionHandler.handleException(e);
    }
}

Sets the handler that should be called after at the end of the handle method. Must be set before handle()
is called.
Params: next – The next handler in the chain.
Mitchell van Pelt
@Override
public void setNext(AuthHandler next) { this.next = next; }

Sets the ExceptionHandler that is called when the handle method encounters an error. Must always be set
before handle() is called.
Params: exceptionHandler – The ExceptionHandler used in the chain.
Mitchell van Pelt
@Override
public void setExceptionHandler(ExceptionHandler exceptionHandler) { this.exceptionHandler = exceptionHandler; }
```

CreateToken:

```java
AuthHandler that handles the creation of a JSON Web Token. After the handle method has succeeded, the
token can be extracted using the getToken method.

23 usages  Mitchell van Pelt
public class CreateToken implements AuthHandler {

    1 usage
    public static final long JWT_TOKEN_VALIDITY = 30 * 60 * 1000;

    2 usages
    private transient String jwtSecret;
    1 usage
    private transient AuthHandler next;
    3 usages
    private transient ExceptionHandler exceptionHandler;
    1 usage
    private transient AccountCredentials credentials;
    2 usages
    private transient String token;

    Constructs a CreateToken handler that handles the creation of a JWT token.
    Params: jwtSecret – The secret that will be used to create the JWT signature.
    4 usages  Mitchell van Pelt
    public CreateToken(String jwtSecret) { this.jwtSecret = jwtSecret; }

    Creates a JSON Web Token (JWT) from the userId. This method will fail if no ExceptionHandler has been
    set.
    Params: credentials – The client's credentials used to create the JWT.
    Mitchell van Pelt
    @Override
    public void handle(AccountCredentials credentials) {
        if (exceptionHandler == null) {
            return;
        }
        this.credentials = credentials;
        try {
            Map<String, Object> claims = new HashMap<>();
            long time = Instant.now().toEpochMilli();
            Date issued = new Date(time);
            Date expires = new Date(time + JWT_TOKEN_VALIDITY);
            this.token = Jwts.builder()
                    .setClaims(claims)
                    .setSubject(credentials.getUserId())
                    .setIssuedAt(issued)
                    .setExpiration(expires)
                    .signWith(SignatureAlgorithm.HS512, jwtSecret)
                    .compact();

        } catch (Exception e) {
            exceptionHandler.handleException(e);
        }
    }
```

The handlers all use an **ExceptionHandler.** Note that this one is different and does not implement AuthHandler.

```
    public ExceptionHandler() {
        this.caughtException = false;
        this.errorMessage = "An unexpected error has occurred. Please try again.";
        this.statusCode = 500;
    }

    Handles an exception by storing it, so it can be accessed later.
    Params: exception – The exception that was thrown.
    7 usages  Mitchell van Pelt
    public void handleException(Exception exception) {
        this.caughtException = true;
        this.exception = exception;
    }

    Handles an exception by storing it, and storing a custom error message and http status code.
    Params: exception – The exception that was thrown.
            errorMessage – A custom error message for the client.
            statusCode – The http status code to be used in the response.
    12 usages  Mitchell van Pelt
    public void handleException(Exception exception, String errorMessage, int statusCode) {
        this.caughtException = true;
        this.exception = exception;
        this.errorMessage = errorMessage;
        this.statusCode = statusCode;
    }

    Gets the exception that was handled.
    Returns: The handled exception. Null if no exception was handled.
    2 usages  Mitchell van Pelt
    public Exception getException() {
        return exception;
    }

    Gets the custom error message. Will be a default message if no specific message was registered.
    Returns: The custom error message.
    12 usages  Mitchell van Pelt
    public String getErrorMessage() {
        return errorMessage;
    }

    Says if an exception was handled or not.
    Returns: True if an exception was handled. False otherwise.
    24 usages  Mitchell van Pelt
    public boolean didCatchException() {
        return caughtException;
    }

    Gets the http status code to be used in the response.
    Returns: The http status code associated with the exception.
    8 usages  Mitchell van Pelt
    public int getStatusCode() { return statusCode; }
}
```

The **ChainCreator class** provides static methods to create a chain. This class was created to encapsulate the many lines needed for chain creation.

```java
public class ChainCreator {

    /**
     * Creates and starts a chain of handlers for registering new accounts.
     * Params: exceptionHandler – The ExceptionHandler user by all handlers in the chain.
     *         accountsRepo – The repository where the accounts are stored.
     *         jwtSecret – The secret String used to sign a JWT.
     *         credentials – The client's credentials to register.
     * Returns: The last handler in the chain.
     */
    8 usages  Mitchell van Pelt
    public static CreateToken createRegistrationChain(ExceptionHandler exceptionHandler, AccountsRepo accountsRepo,
                                                       String jwtSecret, AccountCredentials credentials) {
        SanitizeCredentials sanitizeCredentials = new SanitizeCredentials();
        CreateAccount createAccount = new CreateAccount(accountsRepo);
        CreateToken createToken = new CreateToken(jwtSecret);

        sanitizeCredentials.setExceptionHandler(exceptionHandler);
        createAccount.setExceptionHandler(exceptionHandler);
        createToken.setExceptionHandler(exceptionHandler);

        sanitizeCredentials.setNext(createAccount);
        createAccount.setNext(createToken);

        sanitizeCredentials.handle(credentials);
        return createToken;
    }

    /**
     * Creates and starts a chain of handlers for authenticating new accounts.
     * Params: exceptionHandler – The ExceptionHandler user by all handlers in the chain.
     *         accountsRepo – The repository where the accounts are stored.
     *         jwtSecret – The secret String used to sign a JWT.
     *         credentials – The credentials the client provided.
     * Returns: The last handler in the chain.
     */
    2 usages  Mitchell van Pelt
    public static CreateToken createAuthenticationChain(ExceptionHandler exceptionHandler, AccountsRepo accountsRepo,
                                                        String jwtSecret, AccountCredentials credentials) {
        SanitizeCredentials sanitizeCredentials = new SanitizeCredentials();
        VerifyCredentials verifyCredentials = new VerifyCredentials(accountsRepo);
        CreateToken createToken = new CreateToken(jwtSecret);

        sanitizeCredentials.setExceptionHandler(exceptionHandler);
        verifyCredentials.setExceptionHandler(exceptionHandler);
        createToken.setExceptionHandler(exceptionHandler);

        sanitizeCredentials.setNext(verifyCredentials);
        verifyCredentials.setNext(createToken);

        sanitizeCredentials.handle(credentials);
        return createToken;
    }
}
```