Design Patterns

We decided on two types of design patterns in our system.

- The facade pattern
- The strategy pattern

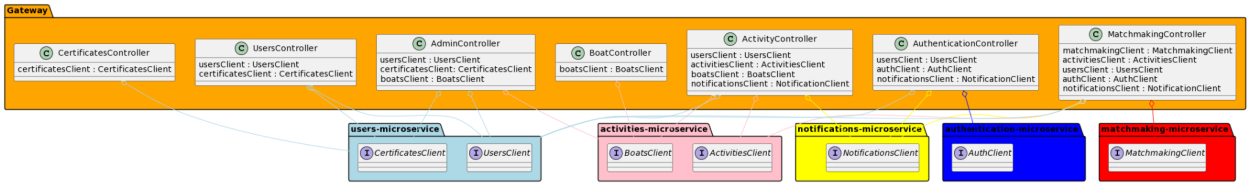
Facade

We implemented the facade design pattern.
This pattern is implemented as the gateway microservice. This microservice makes an interface to interact with the whole system and simplifies a lot of the processes. We choose to use this design pattern to simplify our code and centralize the verification, authentication and interaction of our endpoints.

The gateway serves as the only entrypoint for user requests and thus it is the only place where the authentication system takes effect. Once requests get passed from the gateway to the other microservices they are considered authenticated and validated.

The gateway also greatly simplifies the interaction with our system as now the only endpoints accessible to the outside world are nicely displayed and simplified in order to offer a better experience to the developers or users who will use our API.

Here is a class diagram showing how the gateway controllers interact with the different microservices. The functions from the clients have been omitted for simplicity.



This diagram shows that all the connections between different microservices are done on the gateway layer; this means that all authentication takes place during the gateway.

The design pattern is implemented by the gateway-microservice and has multiple different controllers handling different kinds of functionality.

We have the:

- CertificateController which handles certificates and calls the CertificateClient that is an interface for the certificates from the user-microserviced.
- UsersController this controller handles most user account interactions besides authentication and registration. This endpoint combines the client for certificates with the one for users.
- AdminController holds all the admin endpoints. In order to call these endpoints the logged in user needs to be an admin.
- The boat controller handles boat interactions.

- ActivityController handles most interactions with activities(training and competition). Allows for creating them and managing.
- The authentication controller handles all authentication, registration and password changing.
- The matchmaking controller is what allows a user to find an activity to join. It allows for a number of different searching strategies and filtering criteria.

Here are some snippets of how some of the main controllers look like:
The authentication controller:

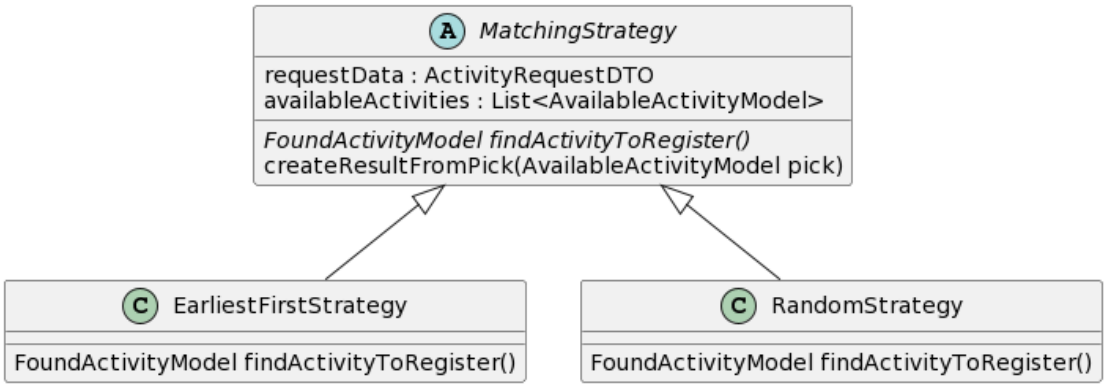
```java
@{ ... }
public class AuthenticationController {

    2 usages
    @Autowired
    private transient UsersClient usersClient;

    6 usages
    @Autowired
    private transient AuthClient authClient;

    1 usage
    @Autowired
    private transient NotificationClient notificationsClient;

    /** The register endpoint. ... */
    1 usage  David Dinucu-Jianu
    @PostMapping("/register")
    public ResponseEntity<String> register(@RequestBody @Valid CreateUserModel createUserModel) { ... }

    /** The authentication endpoint. ... */
    1 usage  David Dinucu-Jianu
    @PostMapping("/authenticate")
    public ResponseEntity<String> authenticate(@RequestBody @Valid AuthenticateUserModel authenticateUserModel) { ... }

    /** The reset password with previous endpoint. ... */
    1 usage  David Dinucu-Jianu
    @PostMapping("/reset_password_with_previous")
    public ResponseEntity<Void> resetPasswordWithPrevious(
            @RequestBody @Valid ResetPasswordModel resetPasswordModel) { ... }

    /** Reset password by email. ... */
    1 usage  David Dinucu-Jianu
    @PostMapping("/reset_password_with_email")
    public ResponseEntity<Void> resetPasswordWithEmail(@RequestBody @Valid Username username) { ... }

    /** Reset password from email token. ... */
    1 usage  David Dinucu-Jianu
    @PostMapping("/reset_password")
    public ResponseEntity<Void> resetPassword(@Valid @NotNull @RequestParam("token") String resetToken,
                                              @Valid @NotNull @RequestBody String newPassword) { ... }
}
```

The clients are interfaces for the microservices. They are implemented using the Feign spring library and provide a really elegant way to connect our microservices and simplify our code. They are used extensively as part of the gateway microservice.

Strategy

We implemented the strategy pattern in our activity finder microservice in order to be able to interchange different types of searching strategies. The activity finder has 2 main features: a user can register in an activity of their choosing or they can let the system find an activity for them. We used the strategy design pattern in the

latter's implementation. Our current implementation supports two searching strategies: earliest activity first and random activity.



Our system allows adding any number of new strategies in the future.
Some possible strategies that are not present in our system at the moment would be:

- Shortest Activity First Strategy - Selects the shortest activity if you do not have much time available but would still like to participate in activities.
- Most Popular Activity First Strategy - This strategy will prioritize activities that have the most number of rowers already enrolled this way the user can participate in a big training or competition activity.
- Closest to me First Strategy - This strategy would use our location field to determine how close a user is to an activity and search based on that.

The possibilities are endless! And our system allows easy integration of any new type of strategy.

Below we share some screenshots of how this design pattern was implemented. Starting with the base class:

Matching Strategy Abstract Class

```
@Data
@AllArgsConstructor
@RequiredArgsConstructor
public abstract class MatchingStrategy {
    private ActivityRequestDTO requestData;
    private List<AvailableActivityModel> availableActivities;

    public abstract FoundActivityModel findActivityToRegister();

    protected FoundActivityModel createResultFromPick(AvailableActivityModel pick) {
        ActivityRegistrationRequestDTO requestDTO =
                ActivityRegistrationRequestDTO
                    .builder()
                    .userName(requestData.getUserName())
                    .activityId(pick.getActivityDTO().getId())
                    .boat(pick.getBoat())
                    .boatRole(pick.getRole())
                    .build();

        return FoundActivityModel.builder()
                .registrationRequestDTO(requestDTO)
                .activityDTO(pick.getActivityDTO())
                .build();
    }
}
```

The random strategy:

Here we show the earliest first strategy implementation: