DESIGN PATTERNS

1. Strategy

Since we have decided to utilize 2 ways of sending notifications, to minimize code duplication, we have opted for the *Strategy* design pattern, which will design when the system should add the notification to the platform and when it should send the message to the registered email. This decision will also allow further changes if more notification-sending options are wanted in the future. The design pattern can be found in the *notification microservice* (see *Diagram 1 - Strategy class diagram* for reference).

The *notification* entity will be able to change strategies, meaning that it will transform the way it deals with notifications. In our idea of implementation, both options will be used. Both approaches will have the same implementation idea, only with slight changes.

Strategy will be an interface. It will make sure that the "Notification" class knows how to handle one notification and what outcome it should give. The *platform strategy* will also append a message on the user's notification list, while also creating a message with all necessary information. *Email strategy* will have the same message, but it will be prepared to be sent to a registered email using a third-party emailing system. This will give flexibility to future development to choose whatever mailing subsystem feels better for our product.

```
public interface Strategy {

    Method that will send notification by different ways
    Params: user – the user that will receive the notification
            event – the event the notification is about
            outcome – the answer from the event: accept or reject
    Returns: a formatted message
    String sendNotification(User user, Event event, Outcome outcome);
}
```

Picture 2 - Structure of "Strategy"

Notification will contain only one value: a strategy. This will permit changing strategies without specifically hardcoding the way the outcomes are being produced. The setter for the strategy will permit the program to choose the method it wants to tackle messages.

```java
@Service
public class Notification {

  private Strategy strategy;

  public Notification(){
  }

  Method for setting the strategy the notification will use
  Params: strategy – the type of strategy
  public void setStrategy(Strategy strategy) { this.strategy = strategy; }

  The method of sending the notification using the "nl.tudelft.sem.template.Strategy"
  design pattern
  Params: user – the user that will receive it
          event – the event the notification is about
          outcome – accept/reject
  Returns: a formatted message
  public String sendNotification(User user, Event event, Outcome outcome){
    return strategy.sendNotification(user, event, outcome);
  }
}
```

Picture 3 - Notification architecture

The controller will have the most important action: processing event data and transforming them into a message & using it to create the so-called notifications. Due to our decision, this API will use both strategies, but this doesn't mean that, whenever the client wants, we cannot change how a notification looks like.

```java
@PostMapping(path = "/{eventId}/{netId}/")
public ResponseEntity<String> sendNotification(@PathVariable ("eventId") Long id,
                                               @PathVariable("netId") String netId,
                                               @RequestParam("outcome") Outcome outcome) {
  this.client = WebClient.create();
  Mono<User> response = client.get().uri( uri: "http://localhost:8084/api/user/netId/?netId=" + netId)
          .retrieve().bodyToMono(User.class).log();
  if (Boolean.FALSE.equals(response.hasElement().block())) {
    return new ResponseEntity<>(HttpStatus.NOT_FOUND);
  }
  User user = response.block();

  Mono<Event> responseEvent = client.get().uri( uri: "http://localhost:8083/" + id).retrieve().bodyToMono(Event.class).log();
  if (Boolean.FALSE.equals(responseEvent.hasElement().block())) {
    return new ResponseEntity<>(HttpStatus.NOT_FOUND);
  }
  Event event = responseEvent.block();

  notification.setStrategy(new PlatformStrategy());
  String message = notification.sendNotification(user, event, outcome);

  notification.setStrategy(new EmailStrategy());
  message += "\n" + notification.sendNotification(user, event, outcome);
  return ResponseEntity.ok(message);
}
```

Picture 4 - Implementation of Notification controller

When calling this API by *localhost:8085/api/notification/1/1/?outcome=ACCEPTED*, the outcome will look like this:

"Bob Bobbie, you have been accepted to Bob's training - TRAINING from 12:00 until 14:00 in week 1, on MONDAY.

Email has been sent to Bob@b.ob with the message:
Bob Bobbie, you have been accepted to Bob's training - TRAINING from 12:00 until 14:00 in week 1, on MONDAY."

The first sentence is the message produced by *PlatformStrategy*, while the second one can easily be recognized as the *EmailPlatform*. It has taken the name of the event, and the type of event it is (either training or competition), and has displayed the interval it is being organized.

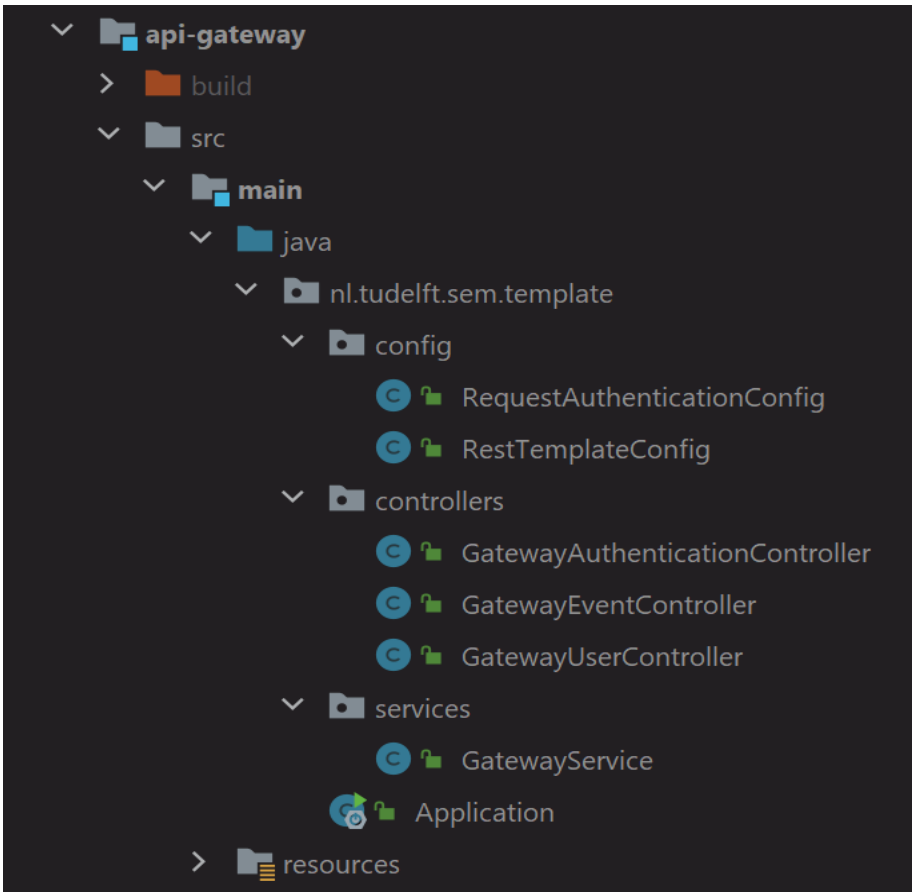
2. Facade

The facade pattern is used in your system to provide a unified interface to a set of interfaces in a subsystem. It allows us to hide the complexity of the underlying microservices and present a simplified interface to the client. When we saw that none of netflix.zuul, or spring.cloud gateways didn't work with our project we decided to implement everything manually to also include the facade in our project.

In our system, the gateway acts as the facade, providing a single entry point for client requests. The gateway then communicates with the various microservices (such as the Event Microservice, User Microservice, etc.) to fulfill the client's request. This allows the client to interact with the system in a more straightforward and intuitive way, without needing to know the details of how each individual microservice works or how to communicate with them directly. Of course knowing the ports of each microservice authenticated user can communicate with it directly, but this was used mainly for debugging purposes, as all real users will only know about the gateway entry-point.

By using the facade pattern, you can decouple the client from the individual microservices, making it easier to modify or replace individual microservices without affecting the client. This can also make it easier to maintain the system, as you can change the implementation of a microservice without having to update the client.

The structure of our code looks as follows (see *Diagram 2 - Façade class diagram* for an overview):



Picture 5 - Gateway architecture

We have created separate controllers for each microservice to split the work and not rely only on one Controller.

```
public class GatewayService {

    private static final String apiPrefix = "http://localhost:";
    private static final String userPath = "/api/user";
    private static final String eventPath = "/api/event";

    @Autowired
    private transient RestTemplate restTemplate;
```

Picture 6 - GatewayService components

The GatewayService is responsible for routing the requests to the actual microservices. Before any action is being performed, to optimize time and runtime speed, we

have decided that this will also make sure that the users have the permission to do it. It will use the pathes of the other microservices and use parts of the original HTTP request in order to redirect to the fitting Gateway controller (due to our design, it being either event, authentication or user).

Every action performed by a user will go throught the matching Gateway controller. The user will make all requests to the url (**localhost 9000:api/**{*microservice it will access*}/{*other parameters*}) the *Gateway* is being linked to , without directly interacting with the other microservices.

The *GatewayControllers* themselves will look identical to the original ones, with one minor difference - the API's contents. This will not contain any proper functionality of the action we want the client to output, bu t will redirect the request to the service that contains all the methods used by APIs. The methods will use Rest Template in order to make auxiliar calls that will be received by one of the main microservices. Those will make in the end the final action.

```
Gets all events.
@GetMapping("/all")
public ResponseEntity<List<Event>> getEvents() {
    try {
        return ResponseEntity.ok(gatewayService.getAllEvents());
    } catch (ResponseStatusException e) {
        throw e;
    } catch (Exception e) {
        return ResponseEntity.badRequest().build();
    }
}
```

Picture 7 - GatewayEventController API contents

```
public List<Event> getAllEvents() {
    return restTemplate.getForObject( url: apiPrefix + MicroservicePorts.EVENT.port + eventPath
            + "/all", List.class);
}
```

Picture 8 - GatewayService method example

For example, if a person wants to fill a profile, behind the scenes the system will first call the *Gateway* which will redirect the call to the *GatewayUserController* which will call the service that calls user-microservice to fulfill adequate request. In the case of filling user profile, it will go directly to the API request for filling up a profile and call the HTTP request inside the original *UserController*, which will do the final computations and give the results.

Diagram 1 - *Strategy* class diagram

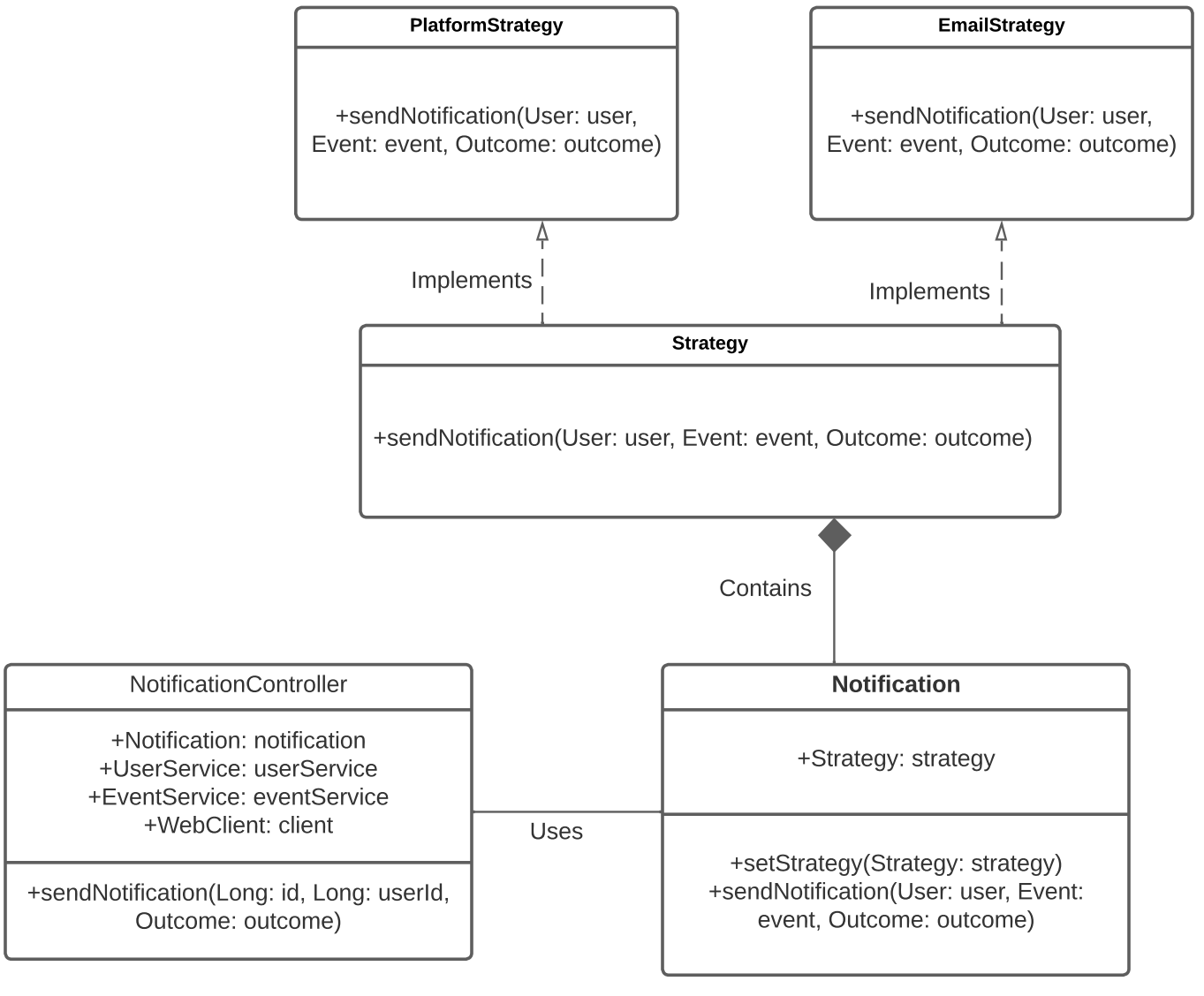


Diagram 2 - *Façade* class diagram