Lab Assignment 1: Design patterns

Software Engineering Methods

Group 30a

Markas Aišparas

Gayeon Jee

Maria Mihai

Jakub Patałuch

Zofia Rogacka-Trojak

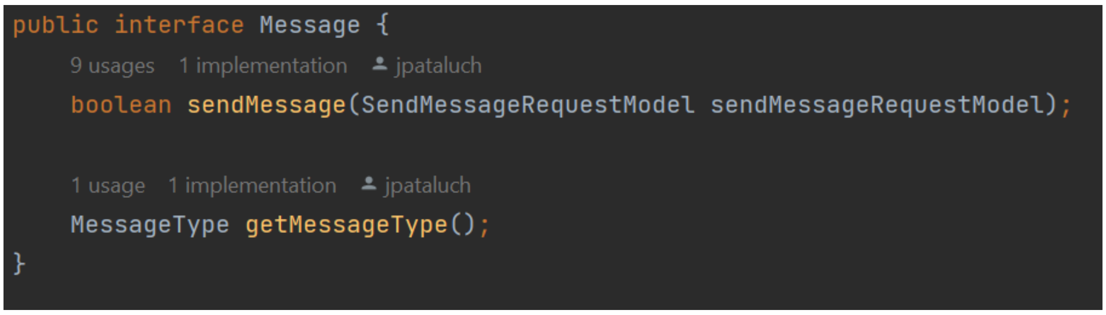
Andrei Simionescu

1) Strategy Pattern in Messaging Microservice

The strategy design pattern is a behavioural software design pattern that enables the encapsulation of a family of algorithms, allowing them to be interchangeable. It provides a way to change the behaviour of an object at runtime by switching its strategy.

In the context of a messaging microservice, the strategy pattern can be applied by defining a common interface for a set of related algorithms, with each algorithm being encapsulated as an object. This interface, referred to as the Message interface, can be implemented by different strategies, such as Email, that provide specific notification functionality.

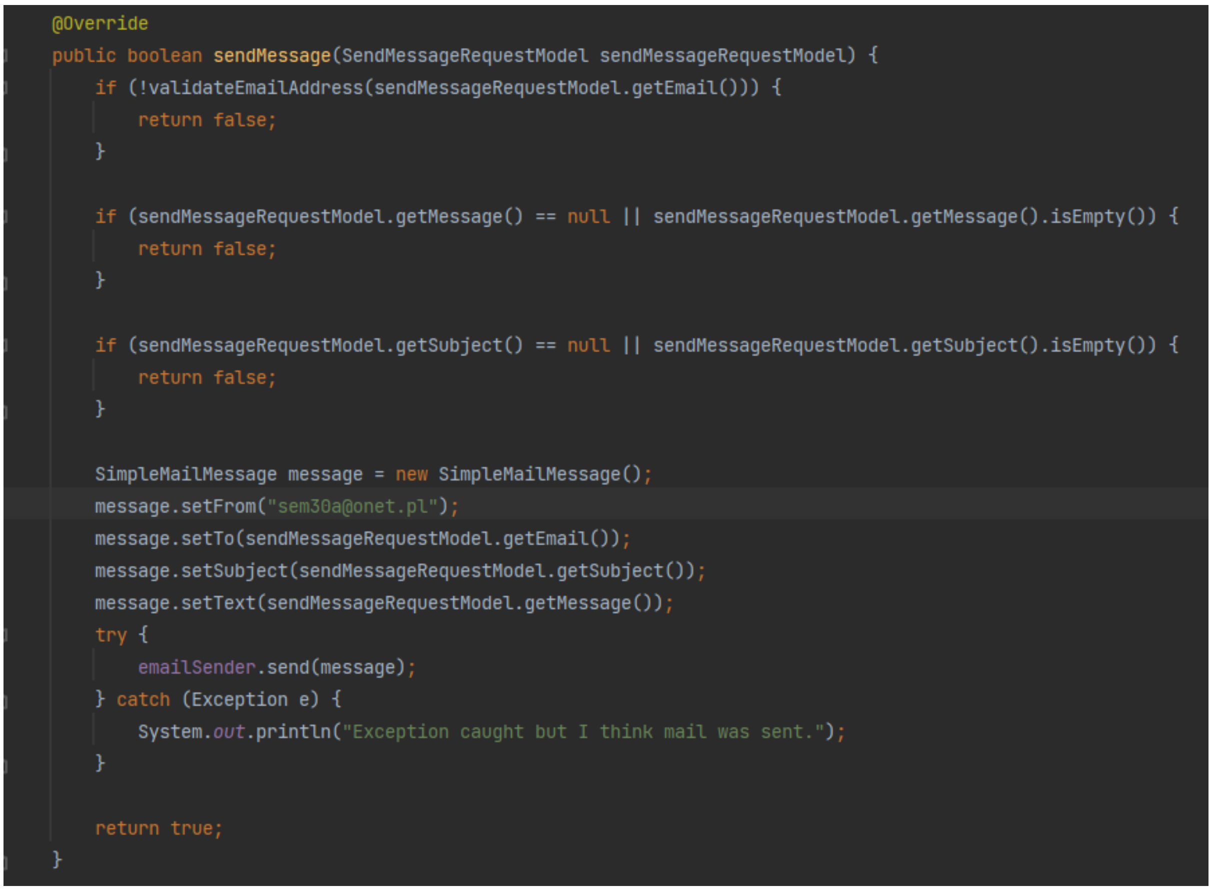
The Message interface in the messaging microservice, located in the package *nl.tudelft.sem.template.messaging.domain.message.entity*, consists of two methods: **sendMessage** and **getMessageType**. These methods define the interface for the algorithms that can be used to send notifications to users.



```
public interface Message {
    9 usages   1 implementation   jpataluch
    boolean sendMessage(SendMessageRequestModel sendMessageRequestModel);

    1 usage   1 implementation   jpataluch
    MessageType getMessageType();
}
```

Figure 1. Message Interface

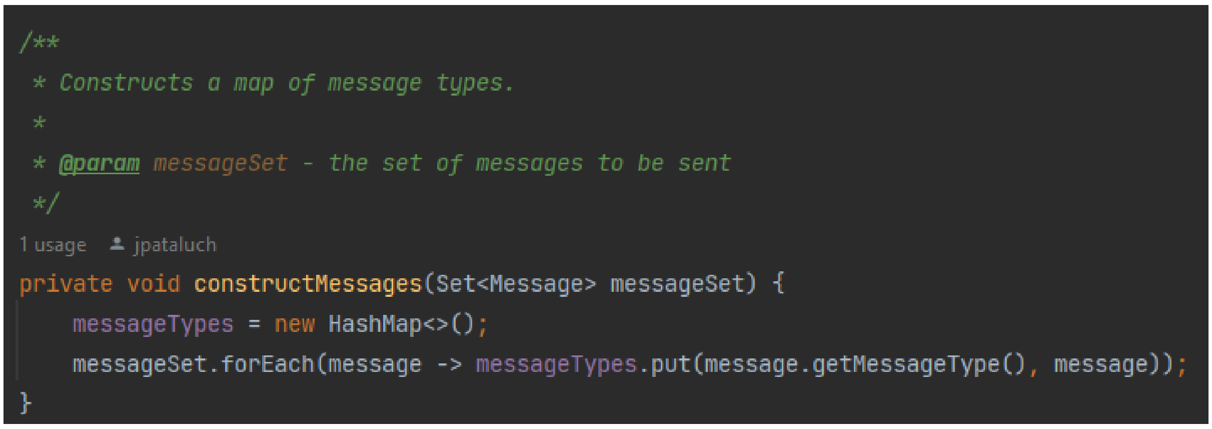
The Email strategy, located in the package *nl.tudelft.sem.template.messaging.domain.message.entity.messages*, is an implementation of the Message interface that sends notifications to users via email.



```java
@Override
public boolean sendMessage(SendMessageRequestModel sendMessageRequestModel) {
    if (!validateEmailAddress(sendMessageRequestModel.getEmail())) {
        return false;
    }

    if (sendMessageRequestModel.getMessage() == null || sendMessageRequestModel.getMessage().isEmpty()) {
        return false;
    }

    if (sendMessageRequestModel.getSubject() == null || sendMessageRequestModel.getSubject().isEmpty()) {
        return false;
    }

    SimpleMailMessage message = new SimpleMailMessage();
    message.setFrom("sem30a@onet.pl");
    message.setTo(sendMessageRequestModel.getEmail());
    message.setSubject(sendMessageRequestModel.getSubject());
    message.setText(sendMessageRequestModel.getMessage());
    try {
        emailSender.send(message);
    } catch (Exception e) {
        System.out.println("Exception caught but I think mail was sent.");
    }

    return true;
}
```

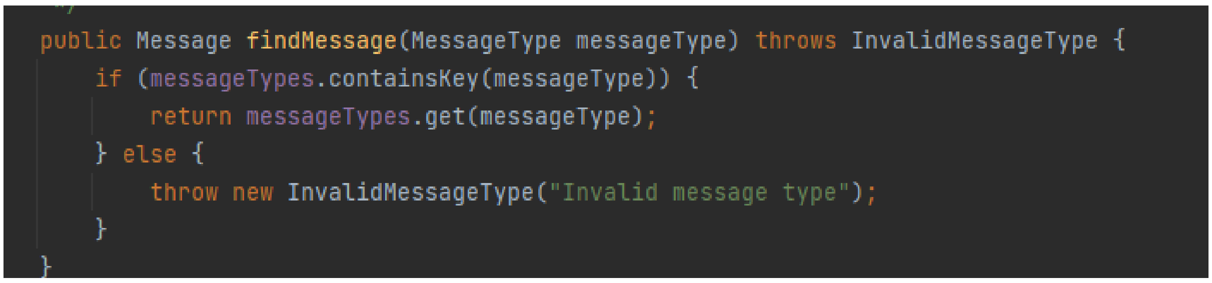
Figure 2. Implementation of Email strategy of sending notification

The MessageFactory, located in package *nl.tudelft.sem.template.messaging.domain.message.factory*, is responsible for creating the actual message, with the strategy previously specified for it.



```java
/**
 * Constructs a map of message types.
 *
 * @param messageSet - the set of messages to be sent
 */
1 usage  jpataluch
private void constructMessages(Set<Message> messageSet) {
    messageTypes = new HashMap<>();
    messageSet.forEach(message -> messageTypes.put(message.getMessageType(), message));
}
```

Figure 3. a. Implementation of constructMessage method in MessageFactory class



```java
public Message findMessage(MessageType messageType) throws InvalidMessageType {
    if (messageTypes.containsKey(messageType)) {
        return messageTypes.get(messageType);
    } else {
        throw new InvalidMessageType("Invalid message type");
    }
}
```

Figure 4. b. Implementation of findMessage method in MessageFactory class

By using the strategy pattern in this way, the messaging microservice can be easily extended with new notification strategies in the future without requiring changes to the existing code. For example, a SMS strategy could be added to send notifications to users via SMS, or a Push Notification strategy could be implemented to send notifications through a mobile app.

Overall, the strategy design pattern allows for flexibility and extensibility in the messaging microservice by enabling the easy addition and swapping of different algorithms as needed.

Class Diagram for Messaging Microservice

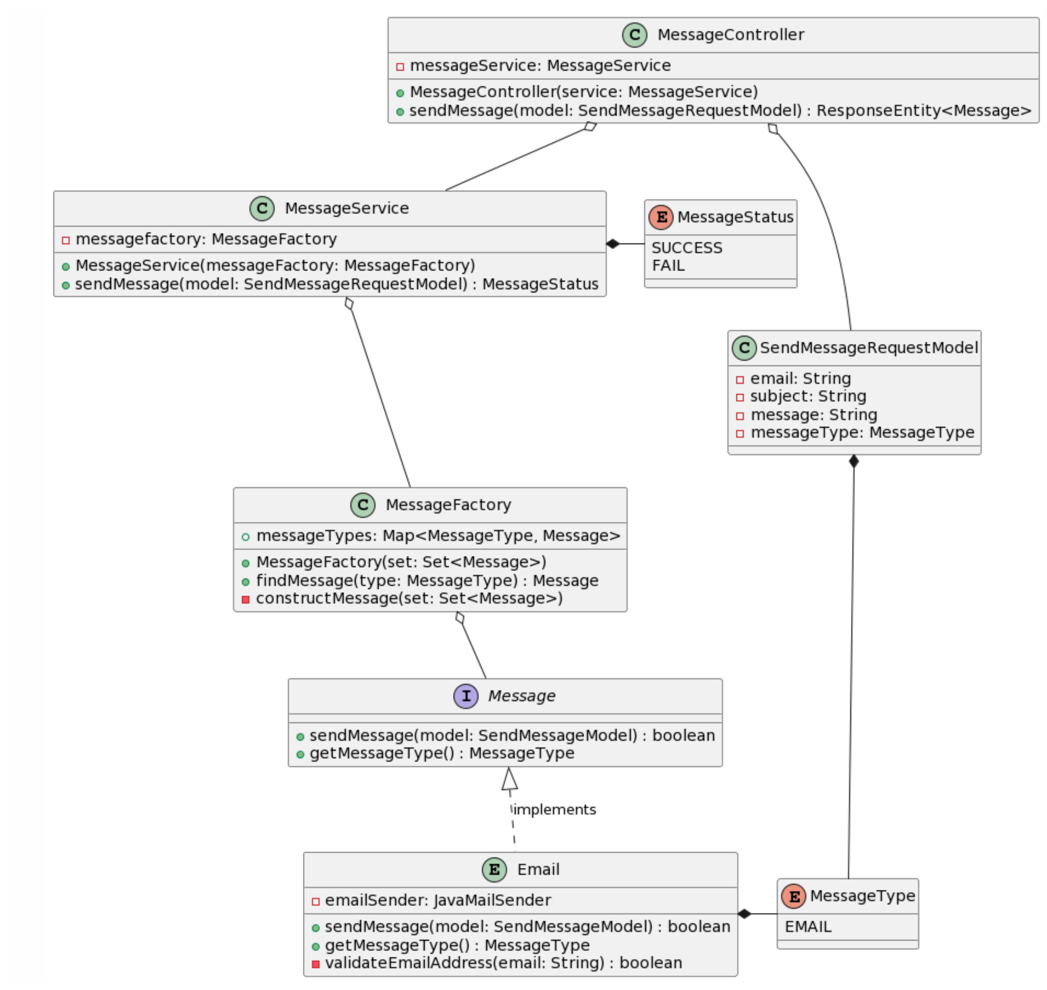

MessageController
messageService: MessageService
MessageController(service: MessageService)
sendMessage(model: SendMessageRequestModel) : ResponseEntity<Message>
MessageService
messagefactory: MessageFactory
MessageService(messageFactory: MessageFactory)
sendMessage(model: SendMessageRequestModel) : MessageStatus
MessageStatus
SUCCESS
FAIL
SendMessageRequestModel
email: String
subject: String
message: String
messageType: MessageType
MessageFactory
messageTypes: Map<MessageType, Message>
MessageFactory(set: Set<Message>)
findMessage(type: MessageType) : Message
constructMessage(set: Set<Message>)
Message
sendMessage(model: SendMessageModel) : boolean
getMessageType() : MessageType
implements
Email
emailSender: JavaMailSender
sendMessage(model: SendMessageModel) : boolean
getMessageType() : MessageType
validateEmailAddress(email: String) : boolean
MessageType
EMAIL

Figure 5. Class diagram of messaging microservice

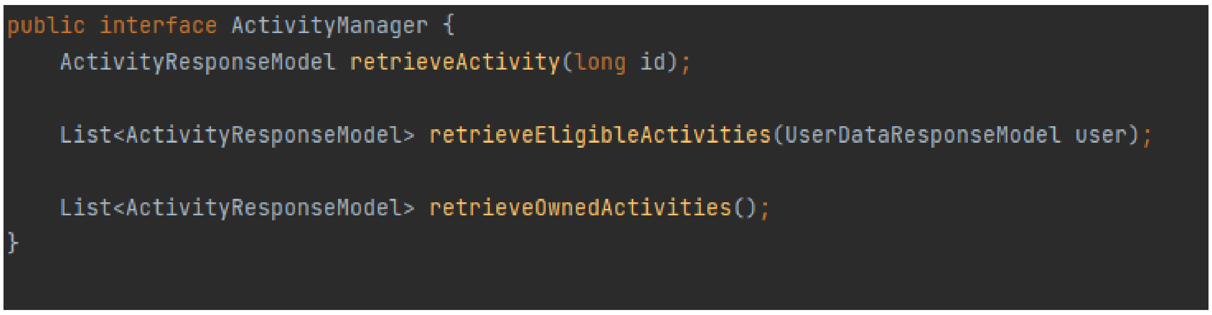
2) Builder Pattern for Planner Microservice

Our planning microservice needs to make requests to the activity microservice to retrieve information about activities. These requests need to be sent to different routes and include different filter query parameters, such as finding activities based on certain roles, availability and many others. Some of these requests might need to include the authentication token some might not.

To avoid code duplication, a naive solution would be to create a method that would take in all the parameters required for the request and send it. However this would make it difficult to support different combinations of filter criteria, or to add or remove filter criteria as needed.

Instead a much better solution would be to make use of the builder creational design pattern. The Builder pattern suggests that you extract the request construction code out of its own class and move it to separate objects called *builders.*
The builder provides a common interface for building and sending requests, and the director uses the builder to choose custom requests as needed. This allows the system to be more flexible and maintainable, as the details of building and sending the request are separated from the logic that uses the request.

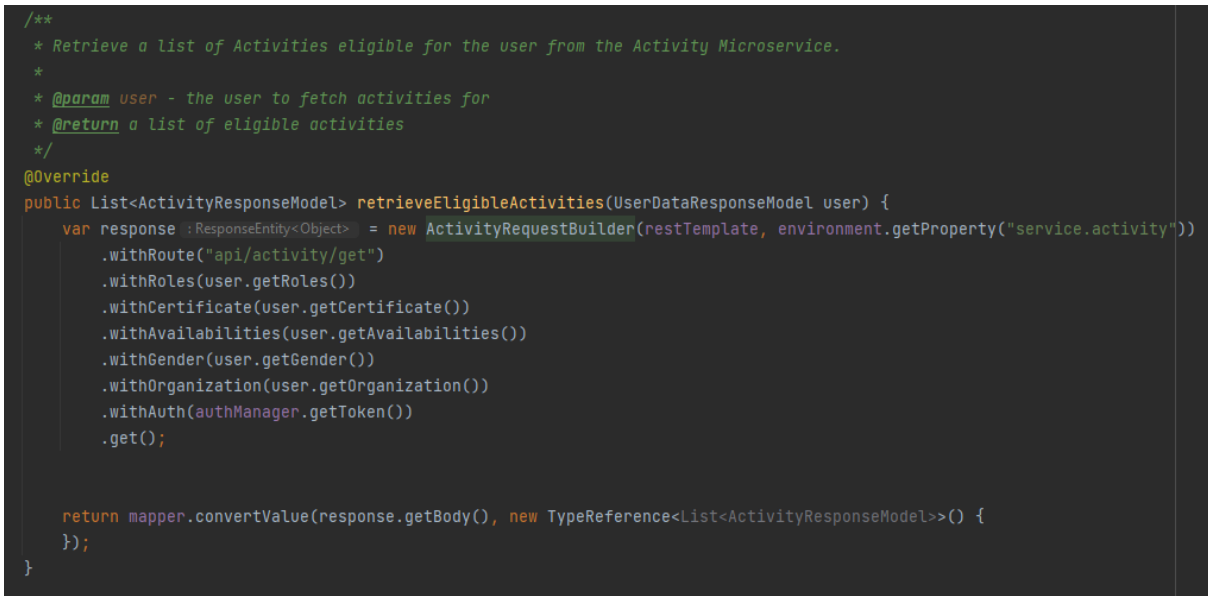
For example, suppose we have an `ActivityManager` interface, located in the `nl.tudelft.sem.template.planning.application.client` package that defines a method for retrieving activities based on certain filter criteria, such as `retrieveEligibleActivities`.



```java
public interface ActivityManager {
    ActivityResponseModel retrieveActivity(long id);

    List<ActivityResponseModel> retrieveEligibleActivities(UserDataResponseModel user);

    List<ActivityResponseModel> retrieveOwnedActivities();
}
```

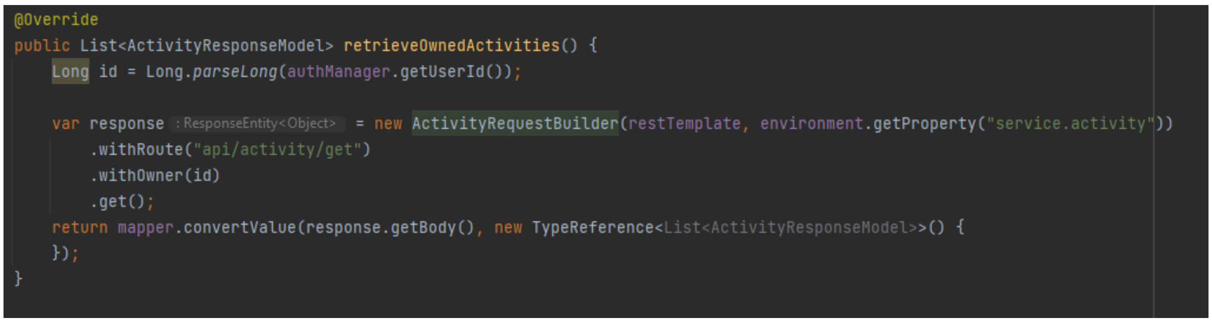
Figure 6 ActivityManager interface.

The `ActivityManagerImpl` class uses an `ActivityRequestBuilder` (located in the `nl.tudelft.sem.template.planning.utils package`) to build and send the requests, allowing the `ActivityManager` to manage the process of retrieving activities in a flexible and maintainable way, without having to directly handle the details of building and sending the requests.



```java
/**
 * Retrieve a list of Activities eligible for the user from the Activity Microservice.
 *
 * @param user - the user to fetch activities for
 * @return a list of eligible activities
 */
@Override
public List<ActivityResponseModel> retrieveEligibleActivities(UserDataResponseModel user) {
    var response : ResponseEntity<Object> = new ActivityRequestBuilder(restTemplate, environment.getProperty("service.activity"))
        .withRoute("api/activity/get")
        .withRoles(user.getRoles())
        .withCertificate(user.getCertificate())
        .withAvailabilities(user.getAvailabilities())
        .withGender(user.getGender())
        .withOrganization(user.getOrganization())
        .withAuth(authManager.getToken())
        .get();


    return mapper.convertValue(response.getBody(), new TypeReference<List<ActivityResponseModel>>() {
    });
}
```

Figure 7. Implementation of retrieveEligibleActivities method in ActivityManagerImpl class

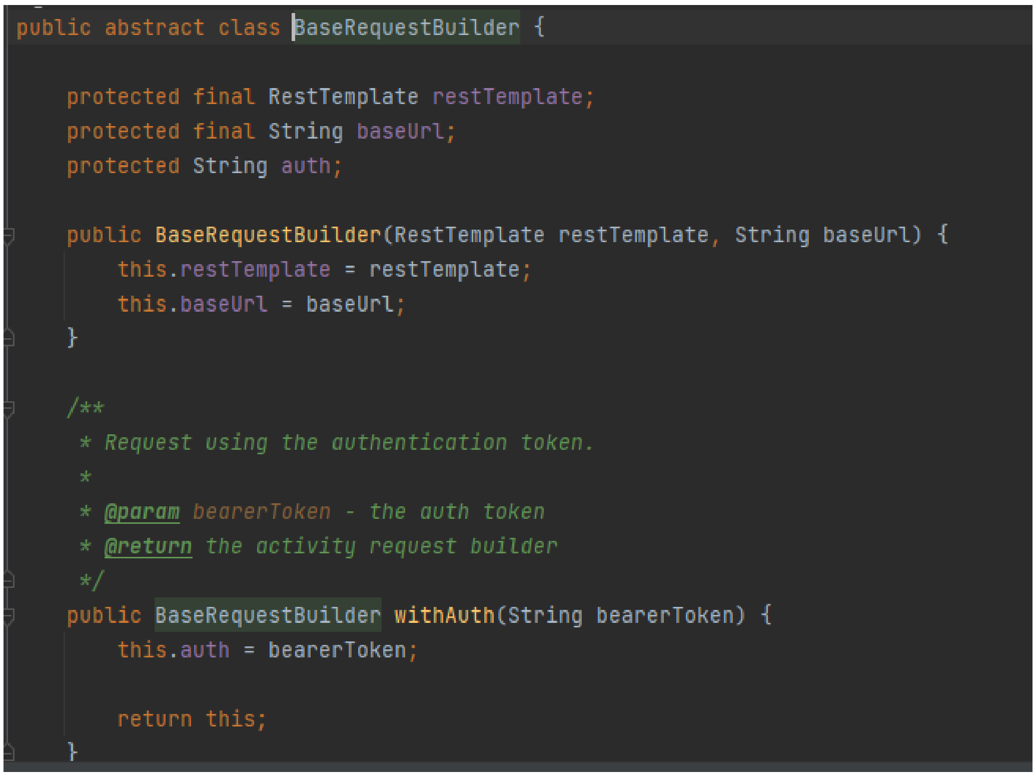


```java
@Override
public List<ActivityResponseModel> retrieveOwnedActivities() {
    Long id = Long.parseLong(authManager.getUserId());

    var response : ResponseEntity<Object> = new ActivityRequestBuilder(restTemplate, environment.getProperty("service.activity"))
        .withRoute("api/activity/get")
        .withOwner(id)
        .get();
    return mapper.convertValue(response.getBody(), new TypeReference<List<ActivityResponseModel>>() {
    });
}
```

Figure 8. Implementation of retrieveOwnedActivities method in ActivityManagerImpl class

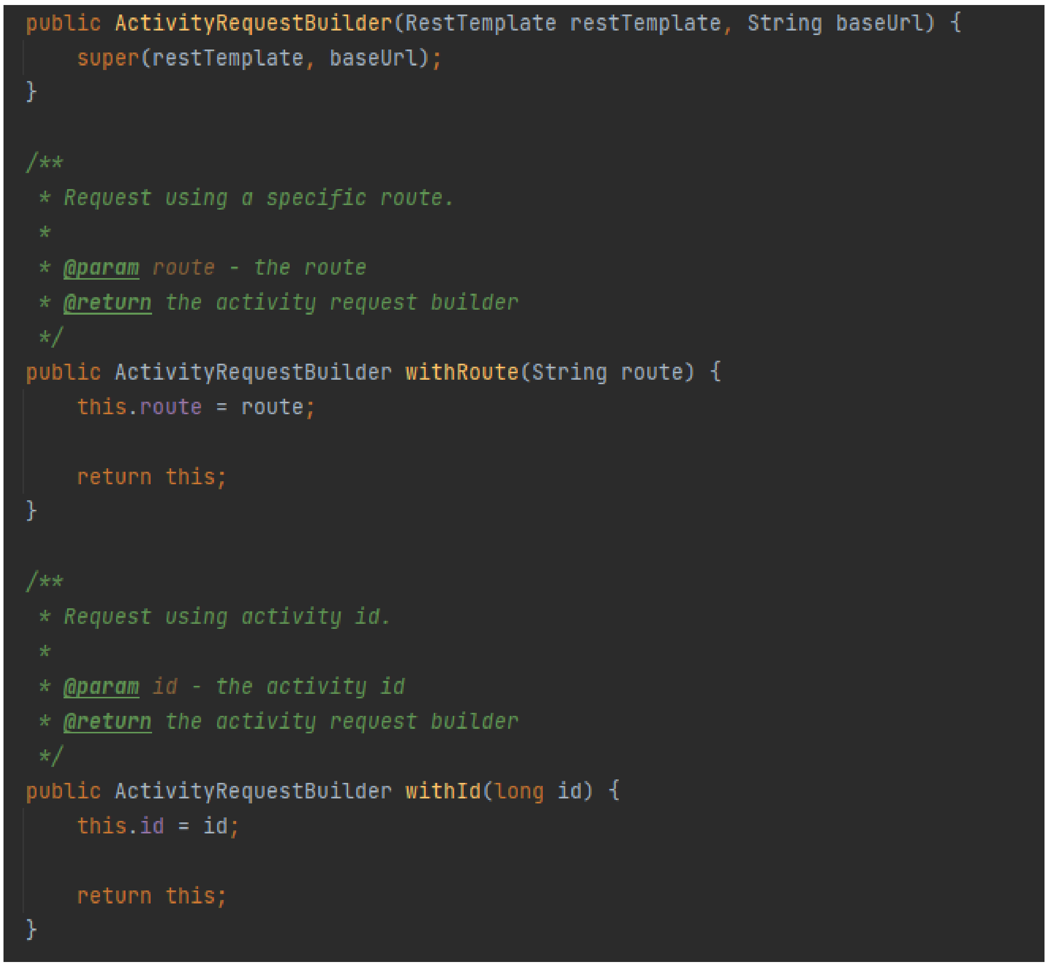
The `ActivityRequestBuilder` provides methods for setting the filter criteria, such as `withRoles`, `withAvailabilities`, or `withCertificate`, allowing the `ActivityManager` to easily customise the request as needed.

Without a builder, it might be difficult to easily modify the filter parameters for a request. For example, suppose you have a method that retrieves a list of activities based on certain filter criteria, such as a starting date, an ending date, and a list of roles. Without a builder, this method would have to manually build the URI and send the request, which means that any changes to the filter criteria would require changes to the method itself. This can make it difficult to support different combinations of filter criteria, or to add or remove filter criteria as needed.



```java
public abstract class BaseRequestBuilder {

    protected final RestTemplate restTemplate;
    protected final String baseUrl;
    protected String auth;

    public BaseRequestBuilder(RestTemplate restTemplate, String baseUrl) {
        this.restTemplate = restTemplate;
        this.baseUrl = baseUrl;
    }

    /**
     * Request using the authentication token.
     *
     * @param bearerToken - the auth token
     * @return the activity request builder
     */
    public BaseRequestBuilder withAuth(String bearerToken) {
        this.auth = bearerToken;

        return this;
    }
```

Figure 9. Abstract class BaseRequestBuilder



```java
public ActivityRequestBuilder(RestTemplate restTemplate, String baseUrl) {
    super(restTemplate, baseUrl);
}

/**
 * Request using a specific route.
 *
 * @param route - the route
 * @return the activity request builder
 */
public ActivityRequestBuilder withRoute(String route) {
    this.route = route;

    return this;
}

/**
 * Request using activity id.
 *
 * @param id - the activity id
 * @return the activity request builder
 */
public ActivityRequestBuilder withId(long id) {
    this.id = id;

    return this;
}
```

Figure 10. Example methods of ActivityRequestBuilder which extends BaseRequestBuilder

Class Diagram for Planner Microservice

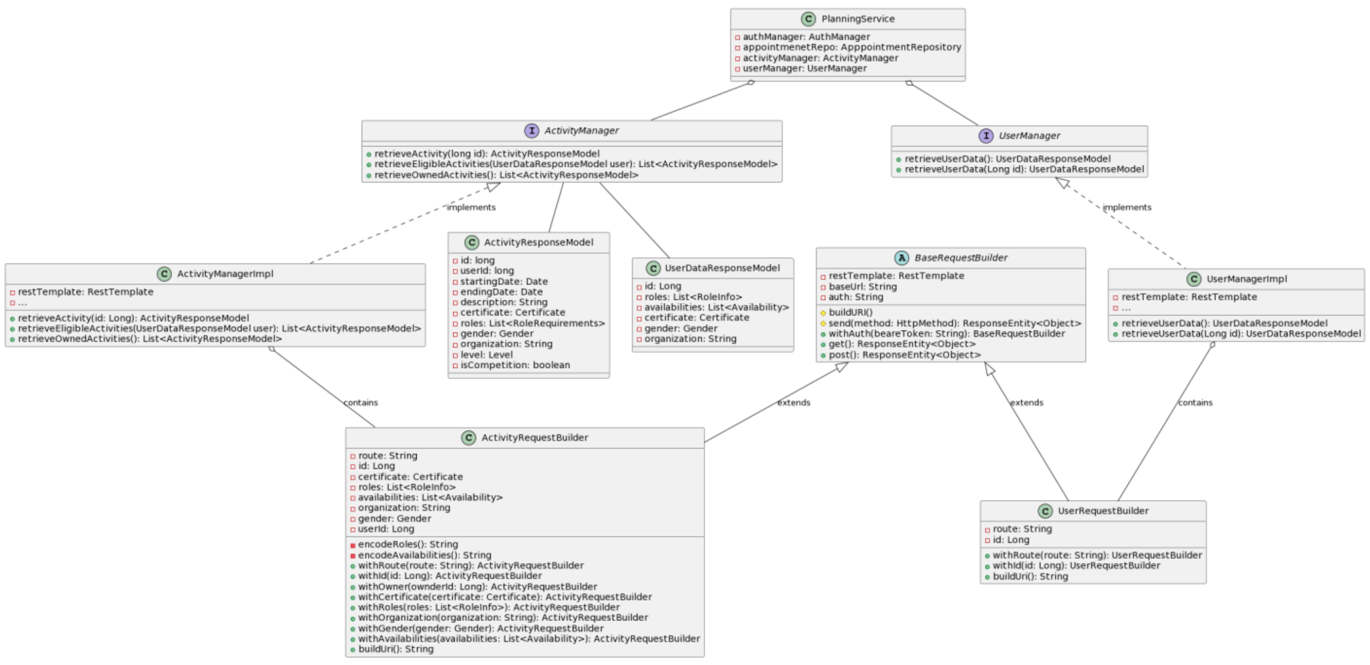

PlanningService
authManager: AuthManager
appointmenetRepo: ApppointmentRepository
activityManager: ActivityManager
userManager: UserManager
ActivityManager
retrieveActivity(long id): ActivityResponseModel
retrieveEligibleActivities(UserDataResponseModel user): List<ActivityResponseModel>
retrieveOwnedActivities(): List<ActivityResponseModel>
UserManager
retrieveUserData(): UserDataResponseModel
retrieveUserData(Long id): UserDataResponseModel
implements
ActivityManagerImpl
restTemplate: RestTemplate
...
retrieveActivity(id: Long): ActivityResponseModel
retrieveEligibleActivities(UserDataResponseModel user): List<ActivityResponseModel>
retrieveOwnedActivities(): List<ActivityResponseModel>
ActivityResponseModel
id: long
userId: long
startingDate: Date
endingDate: Date
description: String
certificate: Certificate
roles: List<RoleRequirements>
gender: Gender
organization: String
level: Level
isCompetition: boolean
UserDataResponseModel
id: Long
roles: List<RoleInfo>
availabilities: List<Availability>
certificate: Certificate
gender: Gender
organization: String
BaseRequestBuilder
restTemplate: RestTemplate
baseUrl: String
auth: String
buildURI()
send(method: HttpMethod): ResponseEntity<Object>
withAuth(beareToken: String): BaseRequestBuilder
get(): ResponseEntity<Object>
post(): ResponseEntity<Object>
implements
UserManagerImpl
restTemplate: RestTemplate
...
retrieveUserData(): UserDataResponseModel
retrieveUserData(Long id): UserDataResponseModel
contains
extends
extends
contains
ActivityRequestBuilder
route: String
id: Long
certificate: Certificate
roles: List<RoleInfo>
availabilities: List<Availability>
organization: String
gender: Gender
userId: Long
encodeRoles(): String
encodeAvailabilities(): String
withRoute(route: String): ActivityRequestBuilder
withId(id: Long): ActivityRequestBuilder
withOwner(ownderId: Long): ActivityRequestBuilder
withCertificate(certificate: Certificate): ActivityRequestBuilder
withRoles(roles: List<RoleInfo>): ActivityRequestBuilder
withOrganization(organization: String): ActivityRequestBuilder
withGender(gender: Gender): ActivityRequestBuilder
withAvailabilities(availabilities: List<Availability>): ActivityRequestBuilder
buildUri(): String
UserRequestBuilder
route: String
id: Long
withRoute(route: String): UserRequestBuilder
withId(id: Long): UserRequestBuilder
buildUri(): String

Figure 11. Class diagram for Planner Microservice