Danmarks Tekniske Universitet

DTU

DTU Pay
02267 Software Development of Web Services

Group 1

Student No.	Authors:
s184477	Frederik Emil Schibelfeldt
s212434	Kári Sveinsson
s010219	Peter Tran
s222961	João Afonso Alves Henriques E Silva
s223186	João Luís Gonçalves Mena
s222963	Tiago Azevedo Rodrigues Silverio Machado
s212953	Þorfinnur Pétursson

Teachers:

Hubert Baumeister

Contents

List of Figures

List of Tables

1 Introduction

In this project we showcase and explain our implementation of the backend of DTU Pay, a mobile payment service for customers and merchants. The objective of this project is to put into practise the development, test and documentation of different web services in a service oriented architecture by working in a team using agile methods, as well as building and deploying it in an application server or a cloud host. The main challenges presented were the design and implementation of new services taking into account their coordination and security according to the requirements presented.

The functionality of the resulting application from this project should allow for registration of users, both as customers and merchants, the creation of payments by the merchant that are validated through a user token, and the generation of reports. For this, we developed four different services: the Account Management service, the Payment service, the Token Management service, and the Reports service, as well as a facade project that includes the customer and merchant facades, which represent the respective mobile applications, and a reports facade to be used by a manager.

The web services in this project were developed using Java's JAX RS for the server side and REST interfaces, Cucumber and JUnit for the behaviour tests, RabbitMQ for message queueing, which allowed communication between services, Jenkins for Continuous Integration and testing, and Docker for the deployment.

The design of the communication between services was possible through event storming, as described further in the report. The use of Hexagonal Architecture allowed us to achieve fault isolation within each service and improve the scalability of the solution.

1.1 Team work

During the project the team had daily meetings where each member updated the group on what they had done the previous day and what they were currently working on or going to work on. To have an overview of what each member was working on we used a Miro board where each member had a task list with two columns, in progress and done. At the beginning of the project the team went through the project description and broke the project down to small tasks and created a pool of tasks that team members could then assign to themselves. The team was also split into 3 groups and each group worked closely together and on similar parts of the project in an effort to keep everyone involved with the different services and all the pieces that were involved in the development.

2 Event Storming

Our team used a technique called Event Storming to explore and understand the events that occurred within the system as well as capture the interactions between the different agents. One of the advantages of Event Storming was that it enabled us to identify potential issues and challenges early on in the development process which in turn helped us design and implement a more robust solution. As a result of the Event Storming sessions a comprehensive visual representation of the events and their interactions was created which allowed for a clear understanding of the workflow. The results are shown in the following figures.

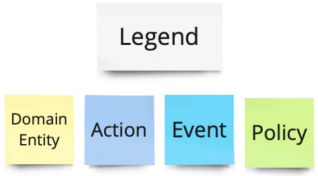

Figure 1: Event Storming Legend

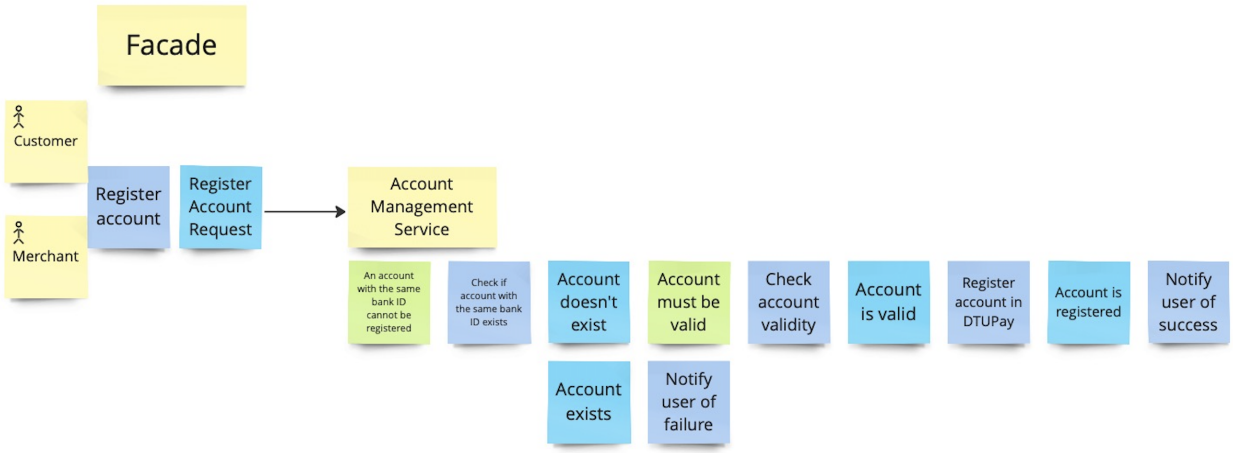

Figure 2: Event Storming diagram for the Account Management registering service

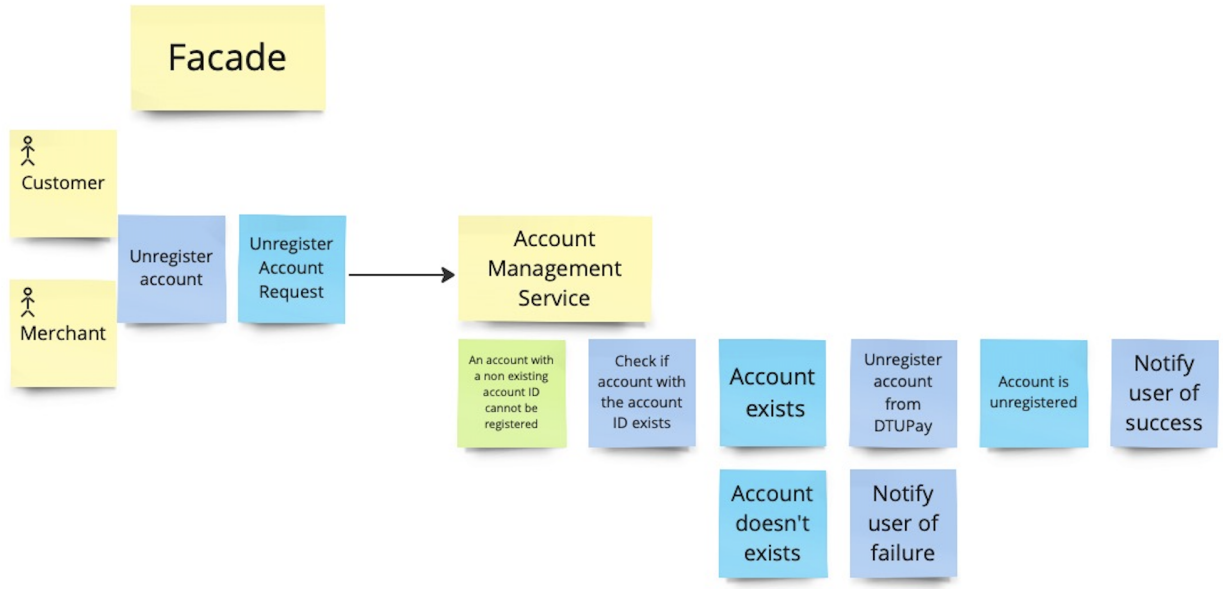

Figure 3: Event Storming diagram for the Account Management unregistering service

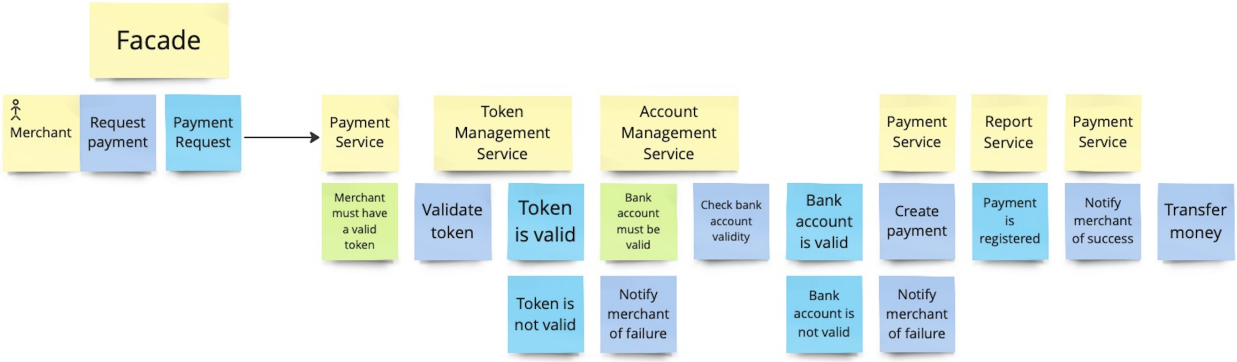

Figure 4: Event Storming diagram for the Payment service

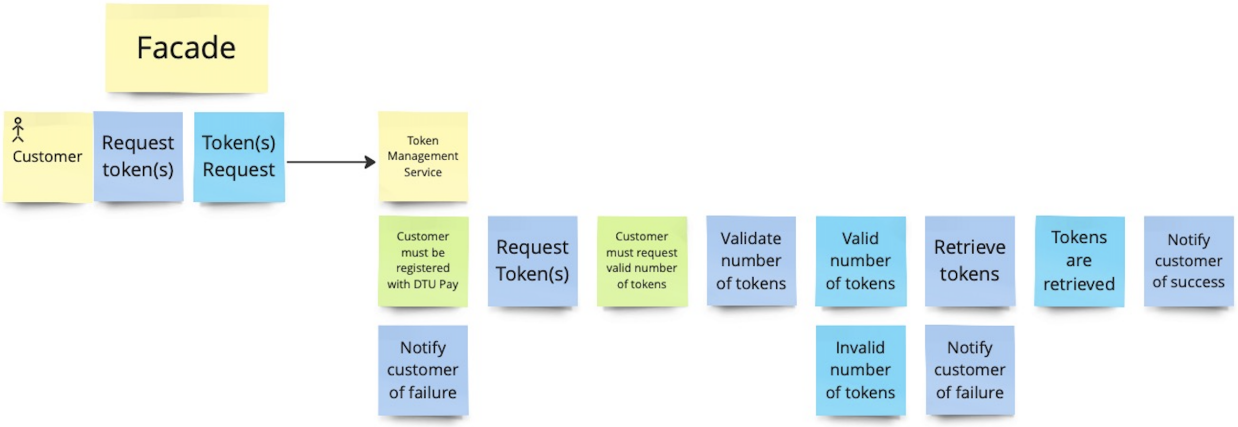

Figure 5: Event Storming diagram for the Token Management service

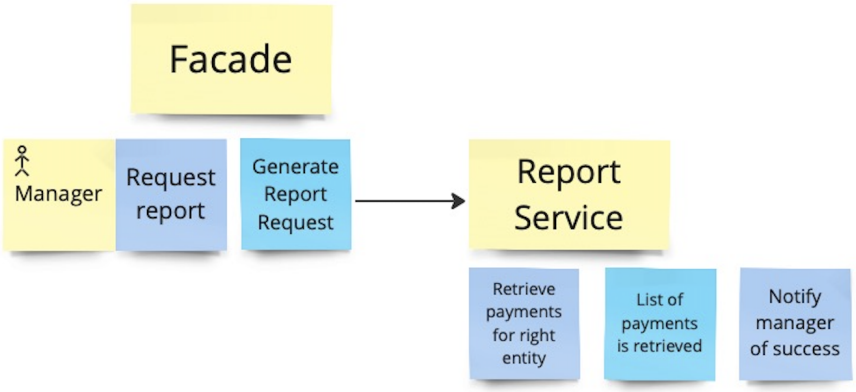

Figure 6: Event Storming diagram for the Report service

3 Design

3.1 Hexagonal Architecture

The DTU Pay web service was designed using a hexagonal architecture and an overview of it can be seen in 7.

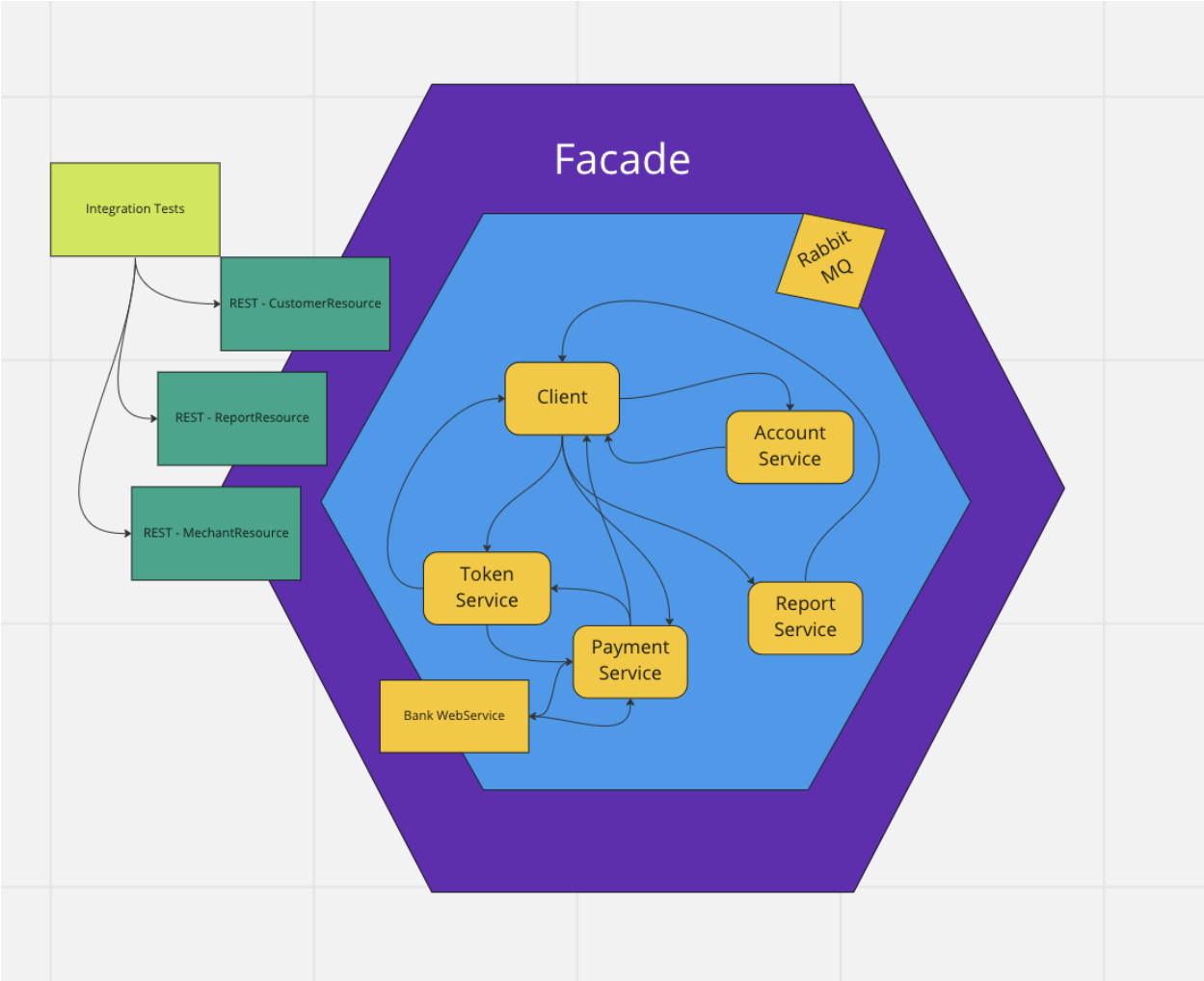

Figure 7: Diagram of the Hexagonal Architecture

The services in the inner hexagon all operate independently. The integration tests test the whole system and the micro services all have their own tests as well. The microservices communicate between themselves through Rabbit MQ messages. The Rabbit MQ is a port that connects the REST resources to the services.

3.2 Account Management Microservice

The account service is responsible for the registration, deletion and storage of customer and merchant accounts. To register with DTUPay, a user must associate a bank account, however, the validity of that bank account is not checked on our end, as that is left for the bank's own business logic and policies. In terms of event handling, the account management ser-

vice processes account registration requests from the facade and requests from the payment service to get an account in order to validate it. It also sends a request to the token service to create a new token user upon the registration of a new customer.

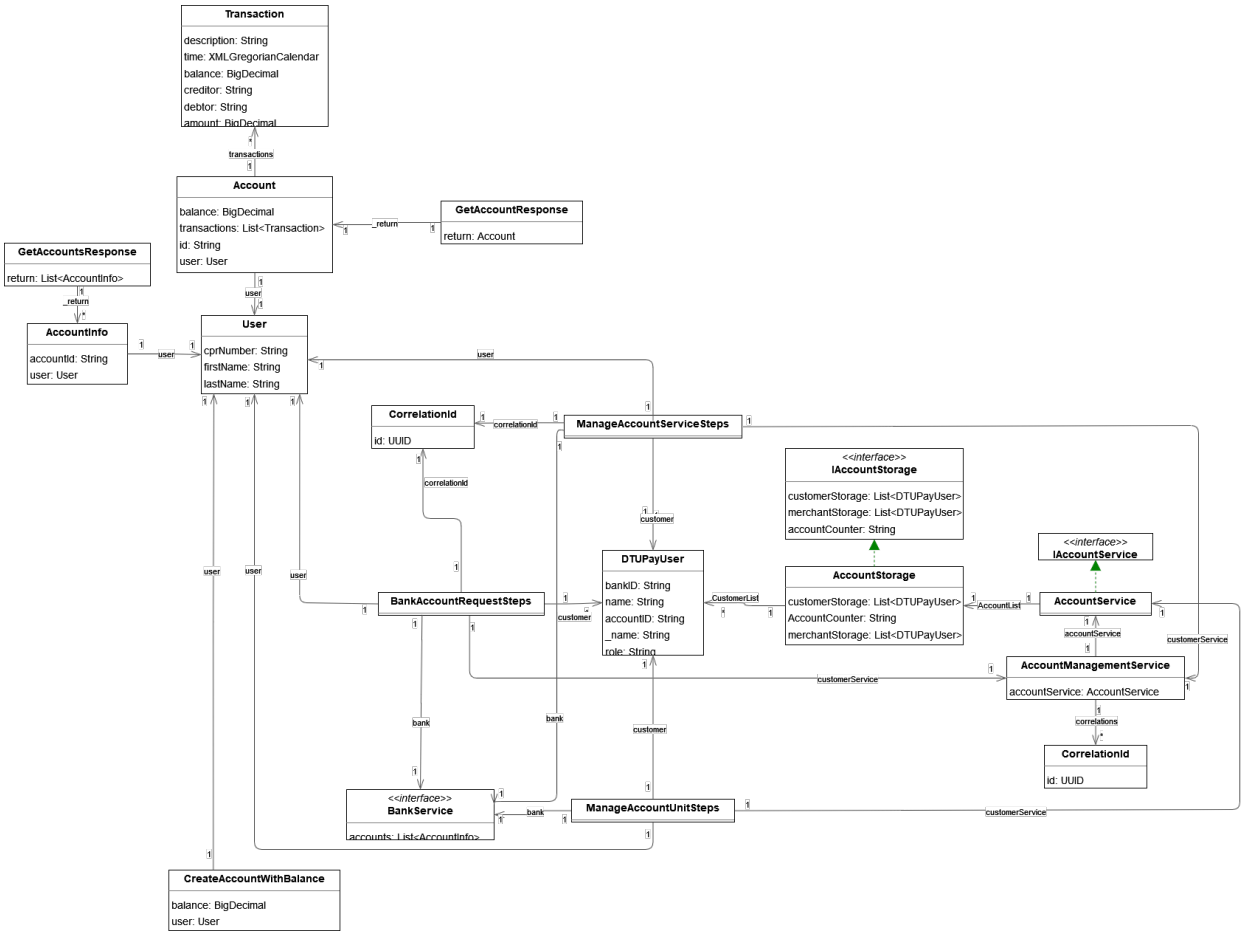

Figure 8: Class Diagram of the Account Service

3.3 Payment Microservice

The payment microservice is responsible for handling payments in the system. The payment service communicates with the bank service to register payments. When a payment is initiated by the merchant, the payment validates the token received with the payment and if the customer has a valid token and the payment goes through the bank then the payment is registered and a record of the payment is sent to the reporting service.

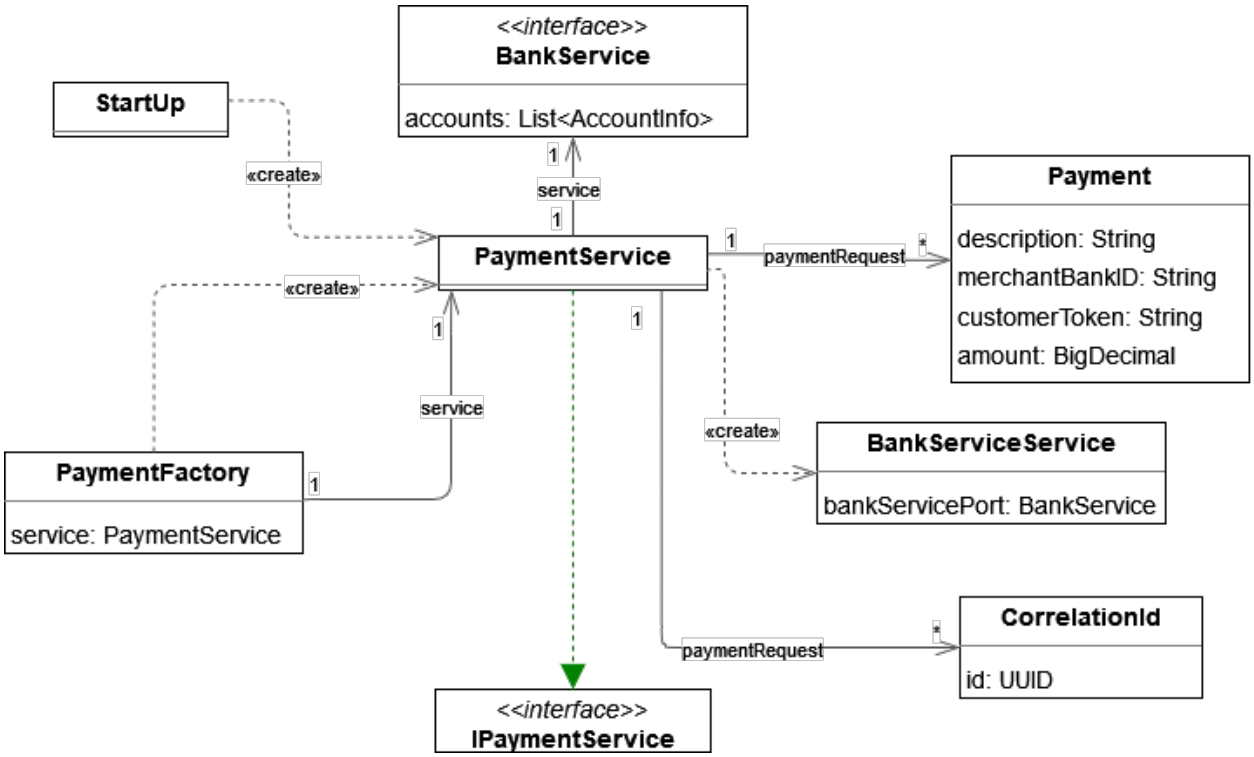

Figure 9: Class Diagram of the Payment Service

3.4 Token Microservice

The token micro service handles the creation and validation of tokens. A customer can request 1-5 tokens and if the customer does not exist in the token service, then he is created. The Token object has two variables, the customer id (user) and a list of tokens. To keep track of the users and tokens a list of Token objects is used. it is The service keeps track of used tokens in a separate list. When a token is validated and used in a payment, then it is removed the Token list and added to the used token list. The used token list is used to make sure that two identical can not be created. The class diagram for the token micro service can be seen in this diagram 10.

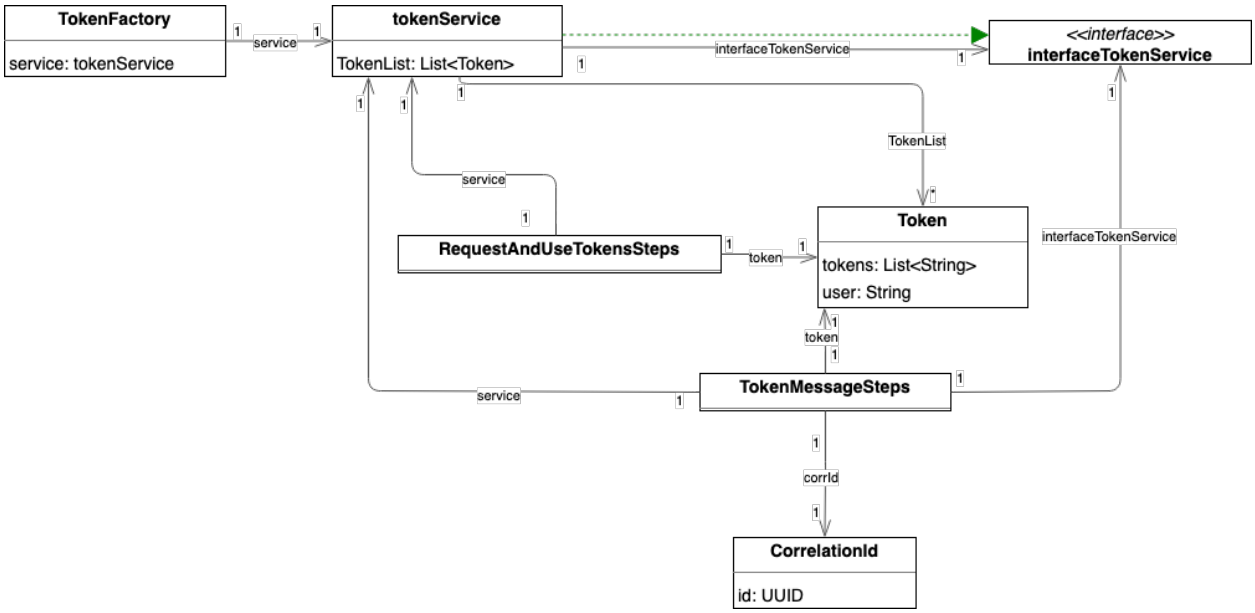

Figure 10: Class Diagram of the Token Service

3.5 Reports Microservice

The report microservice is responsible for generating payment reports. Whenever a payment is completed, the report service receives a message from the payment service and stores the payment information. Customers, merchants and managers can request payment reports that are then handled by the report service. The generated reports are suited for the role and id of the client asking for the report. Customers only get a report with their payments. Merchants only gets reports of payments they have been involved in and managers get all reports stored in the repository.

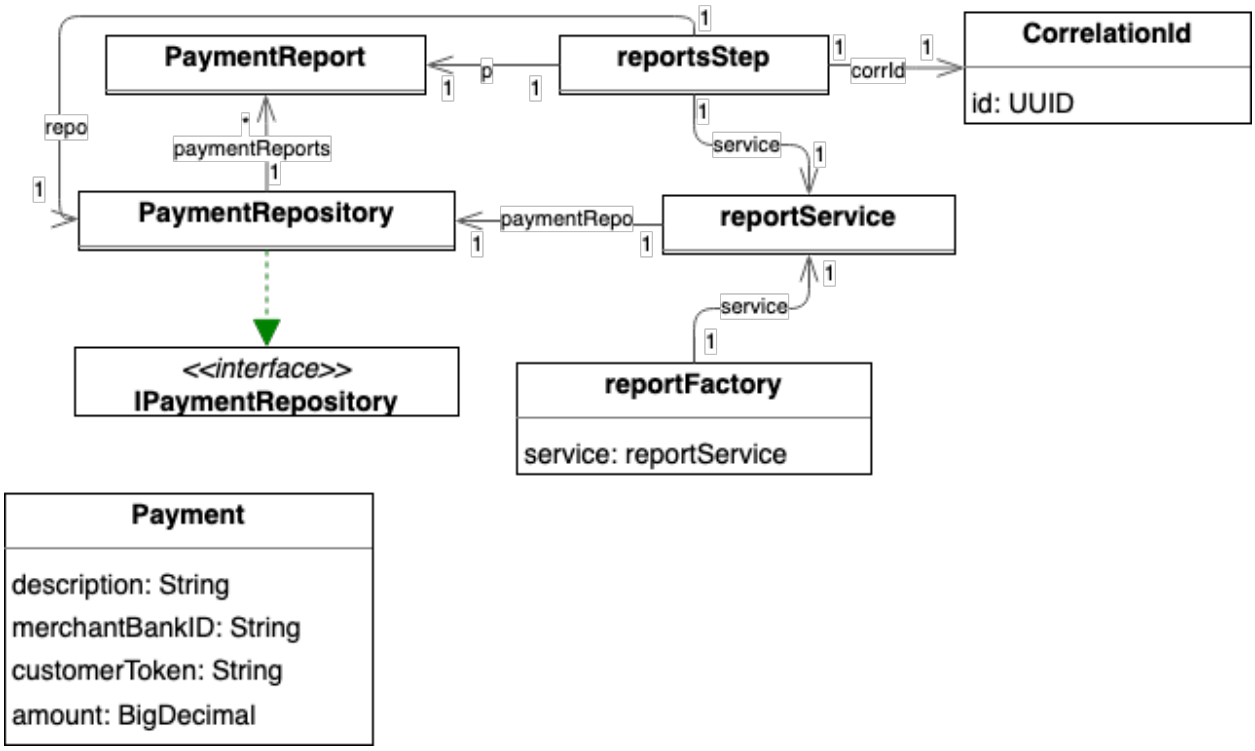

Figure 11: Class Diagram of the Report Service

3.6 Facade

3.6.1 Rest interface

Table 1: Customer REST interface

URI Path / Resource	HTTP Method	Functionality
/customer/report	GET	Get the list of the payments made by the customer
/customer/getTokens	POST	Get tokens for transaction use
/customer/register	POST	Register customer to DTU Pay
/customer/unregister	POST	Un register customer from DTU Pay

Table 2: Merchant REST interface

URI Path / Resource	HTTP Method	Functionality
/merchants/report	GET	Get the list of payments made for the merchant
/merchants/register	POST	Register merchant to DTU Pay
/merchant/unregister	POST	De register merchant from DTU Pay
/merchants/pay	POST	Execute the given payment

Table 3: Report REST interface

URI Path / Resource	HTTP Method	Functionality
/manager/report	GET	Get the list of all the payments made on DTU Pay

4 Features

description of the major features/scenarios * What are these features, what do these features do and how are they realized within your architecture. For example, how is the feature initiated, what messages are exchanged between which Microservices, etc. * You need to show, how a given feature/scenario is realized in your architecture. The idea is, that, in principle, from that description, a reader would be able to implement the feature him/herself.

We consider the major scenarios from our project are the facade tests because it's on the facade that all the requests will merge and we need to be sure that everything is working well:

- Create customer DTU Pay account
- Get tokens as a customer
- Successful payment requested
- Customer asks for report

As the scenarios *Merchant asks for report* and *Merchant asks for report* are quite similiar, we think it's not worth it to talk about them.

4.1 Create customer DTU Pay account

Given the "customer" "John" "Johansen" has a bank account

When the user registers at DTUPay

Then the event "RegisterAccountRequest" is published

When the account is received from account management

Then the account is created

This scenario aims to test the request the facade has to send for the Account Management to create a DTU Pay account.

The scenarios starts just storing the information about the customer which wants to register on DTU Pay, providing his first and last name, and also bank account. After that the request to register the customer in DTU Pay system is sent for the Customer Management service. After that our scenario checks the sending of the request to the service. Finally , when the customer is registered successfully, the account id assigned to the customer is received from the Account Management and after this the account is succesfully created.

So this scenario only requires the communication between the facade and the Account Management Service.

4.2 Get tokens as a customer

Given the customer "10980564" creates account

When the customer asks for 5 tokens

Then the event "RequestToken" is published asking for tokens

When the tokens are received from the account management

Then the tokens are received

The scenario aims to test the customer requesting tokens. The scenario starts with a customer creating an account, then the customer requests 5 tokens. It is then checked if the tokens were created and if the customer received the tokens. When the tokens are successfully created, the customer and the token service store a copy of them.

4.3 Successful payment requested

Given a Customer registered

And a Merchant registered

When the merchant initiates a payment for 10 kr by customer

Then the "MerchantPaymentRequest" is published requesting payment

When the payment service notifies the success of the payment

Then the payment was successful

This scenario has the goal to test the payment functionality, when the merchant requests the payment.

The scenario considers the customer and the merchant to already haver valid DTU Pay accounts. After that, the merchant starts the payment and in this moment the request is sent for the Payment service. The service will check if the payment is possible to make, checking the validity of the token with the Token Management service and will also check the DTU Pay accounts with the Account Management. Then it's checked if the payment request was sent and finally, the Payment service notifies the merchant about the success of the payment and the payment was successfully done.

The payment action is more complex because it requires the communication of the Payment service with Account Management and also Token Management.

4.4 Customer asks for report

Given the user has a DTUPay account with id "123"

When the customer asks for a report

Then the event "generateCustomerReport" is published asking for report

When the costumer report is received from the report service

Then the report is created

The scenario aims to test the customer request for his report. This report is the historic of all his payments.

The scenario considers there is already a DTU Pay account and provides the id assigned to it. Then the customer asks for a report and it's checked if the

5 Contribution

Tiago Machado

- Code: Account Management, Client, Facade
- Tests: Account Management tests, Client tests, Facade tests
- Report: Features

Kári Sveinsson

- Code: Token Service
- Tests: Token Service tests
- Report: Token class diagram (10), Token Microservice, REST interface (1, 2, 3)
- swagger / openAPI
- installation chapter in installation guide

Frederik Schibelfeldt

- Code : PaymentService, ReportService, Facade, Client,
- Docker
- RabbitMQ
- Payment class diagram

João Mena

- Code: Account Management, Facade
- Report: Introduction, Account Management, Facade

João Silva

- Tests: Account Management tests
- Report: Event Storming, Account class diagram (8)

Peter Tran

- Tests: Report Service tests
- Report: Installation Guide, Reports Microservice, Report class diagram (11)
- Jenkins

Þorfinnur Pétursson

- Code: Token Service
- Tests: Token Service tests
- Report: Introduction, Team Work, Hexagonal Architecture, Token Microservice
- Jenkins