02267
Software Development of Web Services

Group 16

Friday 21st January, 2022

Josephine Mellin
s163313

Florian Kesten
s202771

Gunn Hentze
s145494

Bingkun Wu
s210318

Bence Mány
s210278

Tamás Paulik
s212569

Contents

Link for GitLab Repository:

`https://gitlab.gbar.dtu.dk/s202771/Exam-project`

1 Introduction

Author: Bence

The aim of this project is to incorporate knowledge gathered in course 02267 to create, using the microservice architecture, and test a functioning payment system: DTU Pay. This system seeks to give the necessary means in executing transactions between the two types of actors of the system, the customer and merchant. When the customer and merchant agree on a purchase, the customer provides a unique token to the merchant. The merchant then forwards this unique identifier along with the details of the transaction (`merchantID` and amount to be transferred), whereupon the DTU Pay system retrieves customer data from the customer token, and initiates a transaction between their respective bank accounts. The process is destined to make the customer experience faster and easier.

2 Event Storming

Author: Bence

The payment process is the summary of many events. In order to create a fault tolerant system, the group had to take into consideration as many scenarios as possible. A flow of events could be the following for a successful transaction:

1. Consumer and Merchant bank accounts created
2. Consumer and Merchant DTU Pay accounts created
3. Tokens requested and received
4. Transaction initiated by the merchant
5. Token validated
6. Used token deleted
7. Forward transaction request to bank
8. Successful payment
9. Report updated

There are some other scenarios which are not connected to the payment scenario but still important parts of the system, such as Account deletion and Report retrieval.

Before starting the implementation phase, the group did a Domain-Driven Design phase. First, all the considered possible events were put on a board and ordered as a sequence of events, after which the commands for the events and policies were added. The next step was to define aggregates

and bounded contexts on the events. After these steps, the group had a good understanding of the domain, and a figure which would help us design the domain class diagram and give an overview of how the functionality would be split among microservices, and which microservices would need to communicate between each other.

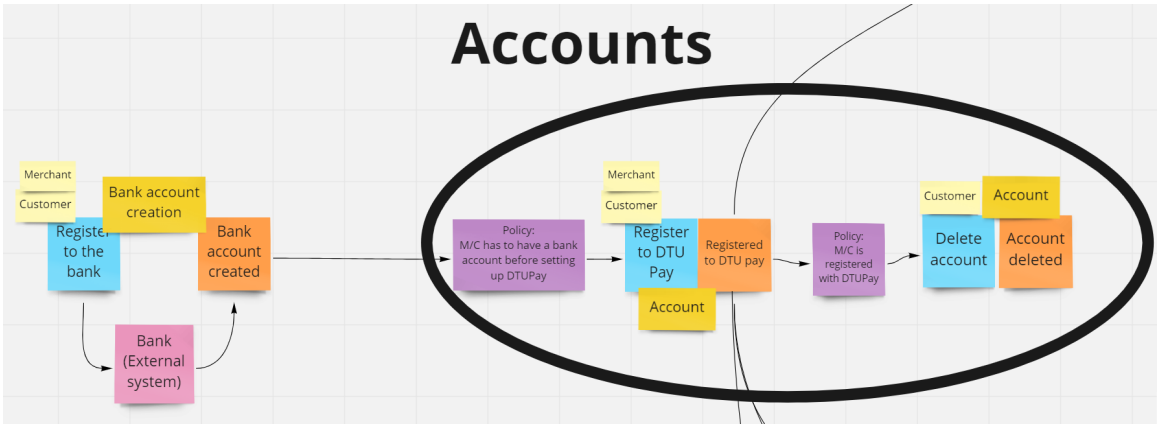

Figure 1: Account part of the DDD.

As a prerequisite to registering at DTU Pay, the merchant or customer first needs to have a bank account and provide the bank account number. An interaction is needed between the Tokens and the Transactions services for a successful payment. Finally, the successful payment is saved in the Reports. The customer and the merchant have to both ask through the account service for the reports.

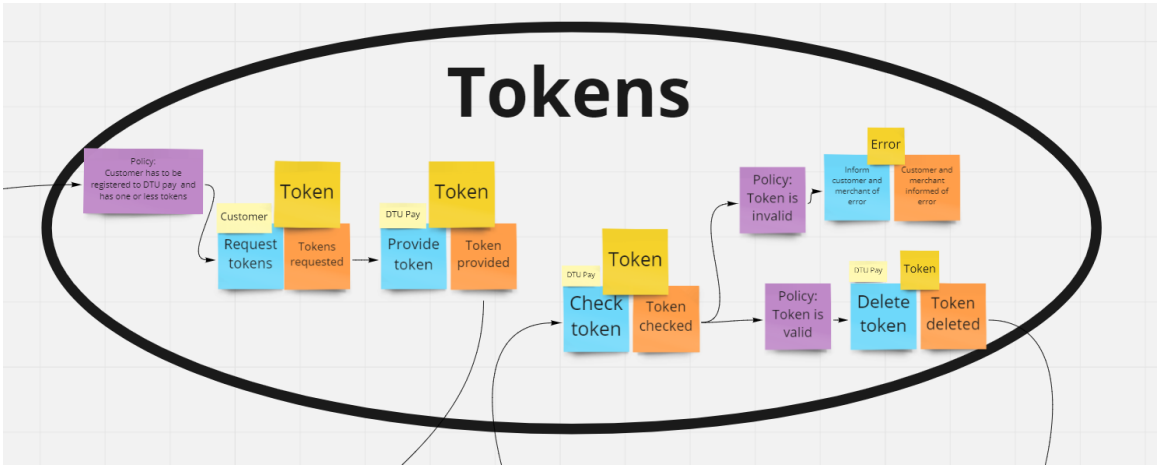

Figure 2: Tokens part of the DDD.

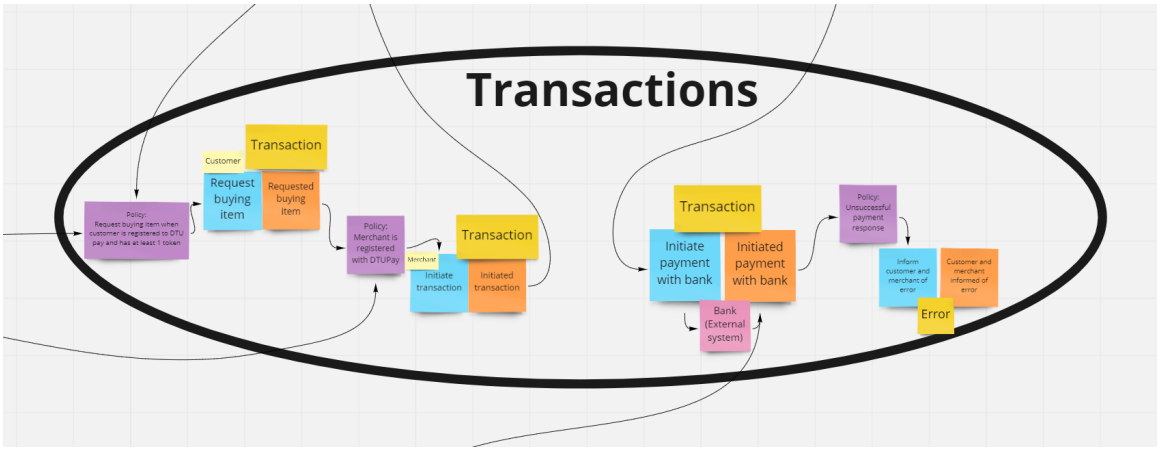

Figure 3: Transactions part of the DDD

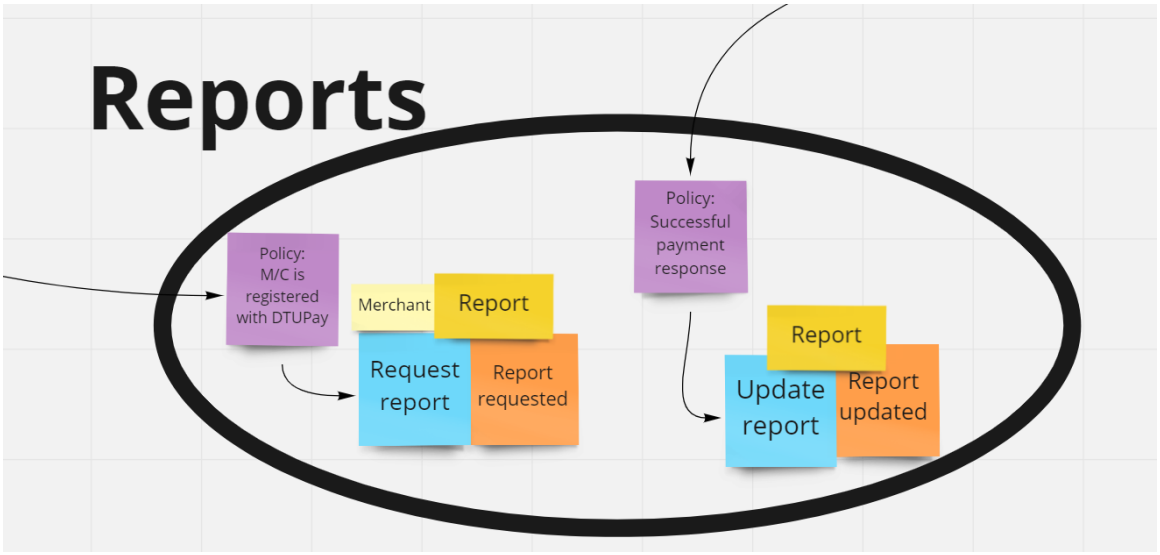

Figure 4: Reports part of the DDD.

After the event storming session, a domain class diagram was created showing the classes and their corresponding events and aggregates, which can be seen on figures 5, 6, 7 and 8.

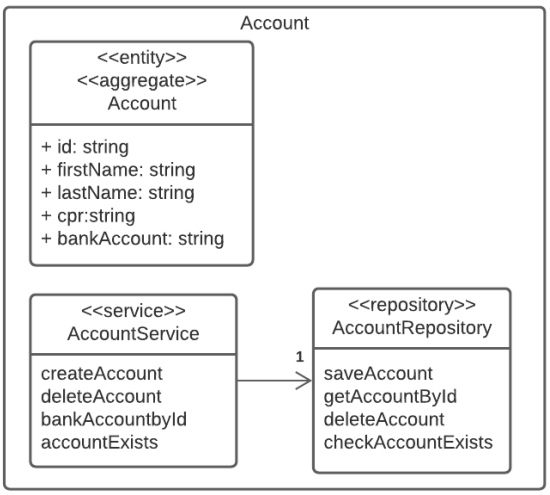

Figure 5: Domain Class Diagram of the Account.

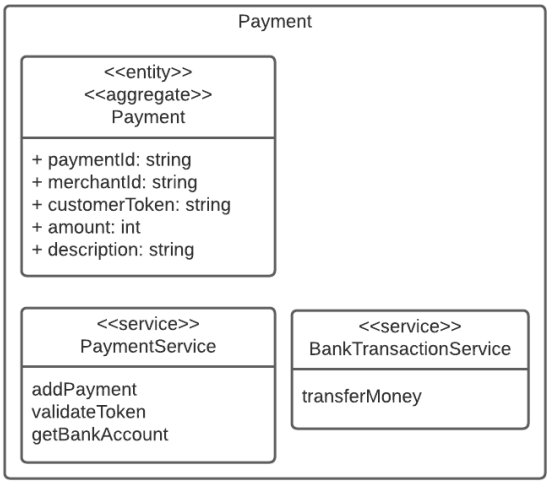

Figure 6: Domain Class Diagram of the Payment.

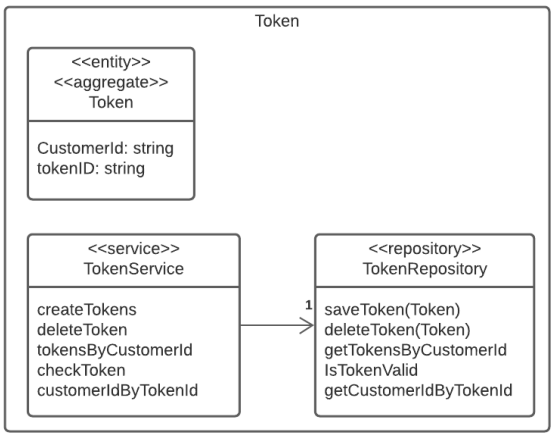

Figure 7: Domain Class Diagram of the Token.

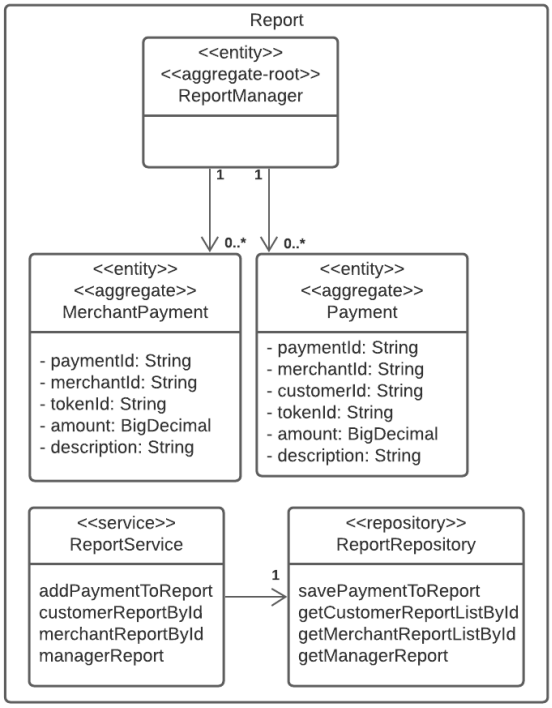

Figure 8: Domain Class Diagram of the Report.

3 Architecture

3.1 Microservices

Author: Bence

There are four microservices that are maintaining and executing certain payment scenarios. These four are: `Account`, `Token`, `Payment`, and `Report`. Over this, we have the DTU-pay microservice that communicates with the outside world through the REST API access points.

In this regards the rabbitMQ open-source message-broker software is used to communicate between our microservices and our Data Transfer Objects for these transmissions. In the DTU Pay service, the rest adapters for the merchant and the customer end-points are separate. The Payment service communicates with the bank API through SOAP. Further a service for end-to-end testing is created, which simulates the calls of the clients.

A successful payment can be done by first having a registered account in the DTU Pay system (and certainly one in the bank), then initiating a transfer via a valid token which can be requested from and checked by the Token service. If all conditions are positive: Available accounts, sufficient balance, valid token, then the transfer will be successful. Eventually, all payments are stored in the Report service so one's transaction history can be requested at any time. In the following section, each microservice is going to be discussed - both their business logic and the communication between them.

3.1.1 Account

Author: Gunn

First of all, every participant has to have an account in the system. The required information during the registration process contains the participant's full name and CPR number. Different people can not have the same CPR number. If somebody tries to register with an already existing CPR number, the process will be denied. The locally stored "accounts" map contains all the created accounts. Using that, there is also a possibility for checking if a user has an account in the system. This is important during the payment process: if a participant's account is invalid, an error has to be thrown. And finally the `deleteAccount` function takes care of retiring the accounts which are not needed anymore.

3.1.2 Token

Author: Tamas

The Token service has three important functions: `createToken`, `checkToken`, `deleteToken`. When a customer wishes to initiate a transfer he needs to possess at least one token. Every token can be used only once, therefore the customer has to request new tokens at a certain point. However, there are some basic rules:

- A user can only have maximum 6 tokens at the time.
- A user can only request new tokens when having 0 or 1 token.

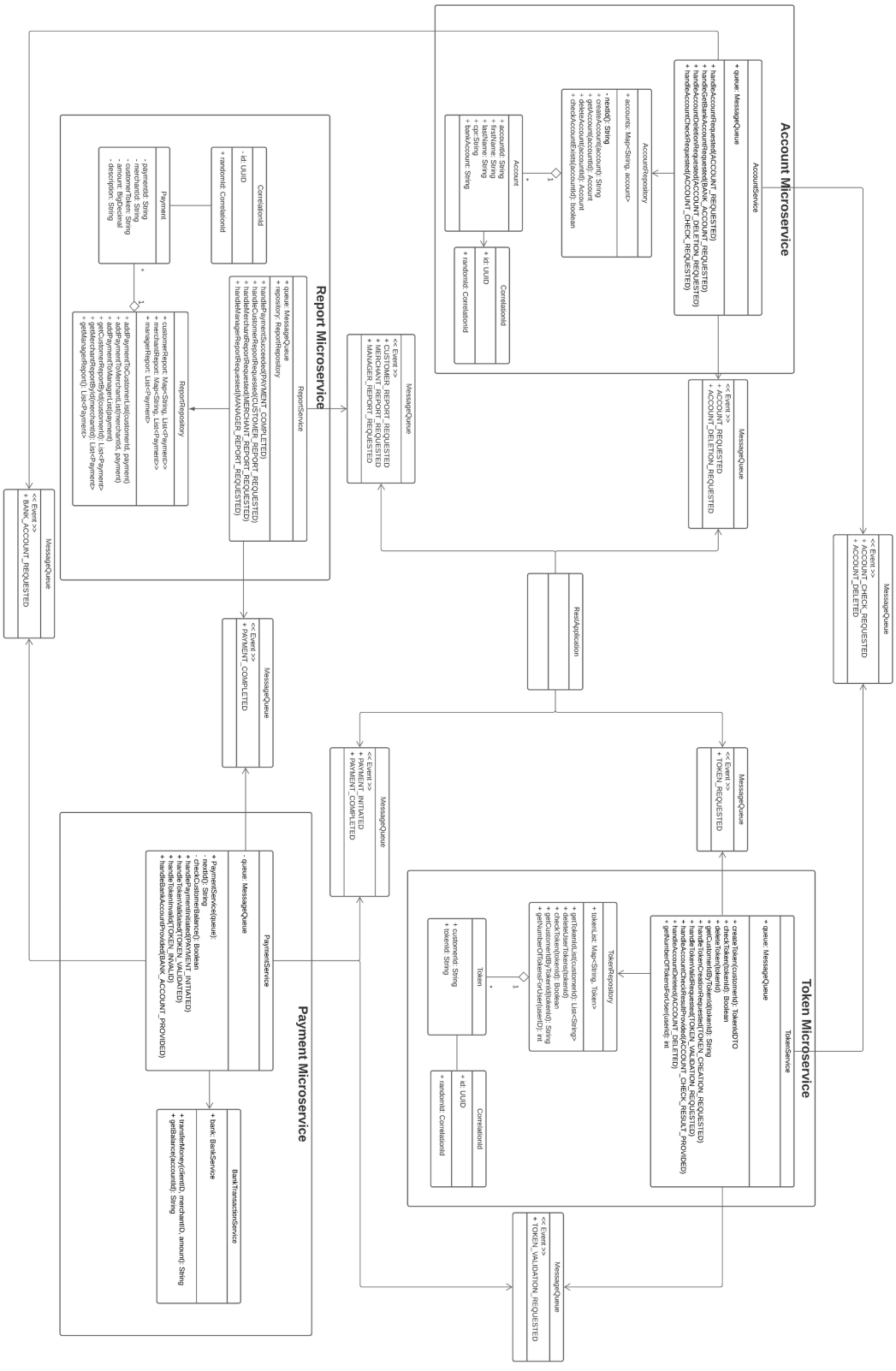

Figure 9: Class diagram of the architecture, excluding the facade

Therefore the function for token creation first checks the amount of tokens possessed by the user - if it is less than 2, it will create the amount of tokens requested up to user, with the total number not exceeding 6. The tokens are also stored locally in the service so they can be validated later. However, noted that not the full token object is being returned to the customer, only a list of `tokenIDs`. This is due to privacy: our Token objects contain the customer ID, and the merchant should not know who the customer is. A token is built up as the following: {`String customerID`, `String tokenID`}
If the token creation is successful, the list of `tokenIDs` is being sent as a Data Transfer Object (`TokenDTO`), inside an event. This event is published by the service as the following:

```
Event("TOKEN PROVIDED", new Object[] { tokenIdDTO, correlationId })
```

When a customer purchases something the merchant forwards the `tokenID` alongside the rest of the payment request, and the token gets verified and the respective account id for the user retrieved . It is done by looking up the stored tokens - if the list contains the given `tokenID` then it is valid. After it is being used, the token has to be deleted from the list so if a user wants to reuse it, the validation is going to fail.

3.1.3 Payment

Author: Bingkun
The payment process is the core of the whole system. It has to handle many cases and possible failure scenarios. In order to complete a payment, the following conditions need to be met: the system has to check if both the customer and merchant have valid accounts, both the customer and merchant have to have correct bank accounts, the customer needs to have a valid token, the bank accounts need to have sufficient balance. Failing any of these conditions will mean that the payment fails and an appropriate error response is sent back.
Hereto, the following tests are considered for failure scenarios:

- Non existent DTU Pay accounts (customer, merchant)
- Invalid bank accounts (customer, merchant)
- Invalid token
- Insufficient funds

3.1.4 Report

Author: Josephine
Usually the customer likes to know how and where he spent his money recently. The report service helps solving this problem by storing all the transfers executed through the system. They are stored as report objects with the following components:

- `paymentID`
- `customerID`
- `merchantID`

- tokenID
- amount
- description

Both the merchant and customer can request a list of their transactions, however the merchant can't see the customerID, only the customer's tokenID (due to privacy reasons). So the main functionalities of this service is addPayment and getReport. The prior is called every time when a successful transaction has been executed, the latter can be called anytime the customer or the merchant wishes to do so. The customer and merchant payments are stored separately and they can only access their own payment history. There is also a third party: manager. He can see all the transactions done in the past.

3.2 REST interfaces

Author: Bingkun

The REST endpoints and resources are divided into three groups. The customer resource handles the customer accounts, tokens and reports.

A customer can send an HTTP request to the customer endpoint to create new accounts. It applies a POST request which carries a accountDTO object including all the fields needed (name, cpr, bankAccount, etc.). The customer can also delete the account by its ID. The customer can request new tokens by ID and the response contains a list of tokens in a TokenIdDTO object.

URI Path = Resource	HTTP Method	Function
/customer/accounts	POST	Register an account
/customer/accounts/{id}	DELETE	Unregister an account
/customer/tokens/{id}/tokens/{tokenAmount}	GET	Get tokens for customer
/customer/reports/{customerId}	GET	Get reports for customer

Table 1: Customer Resource

Correspondingly, the services for merchant are aggregated into merchant resource. Merchant is able to create new payments using POST request. The body of the POST request is a paymentDTO object which contains all the fields required (id, tokenId, amount, etc.).

URI Path = Resource	HTTP Method	Function
/merchant/accounts	POST	Register an account
/merchant/accounts/{id}	DELETE	Unregister an account
/merchant/payments/{id}/payments	POST	Create a new payment
/merchant/reports/{id}	GET	Get reports for merchant

Table 2: Merchant Resource

URI Path = Resource	HTTP Method	Function
/manager/reports	GET	Get all reports

Table 3: Manager Resource

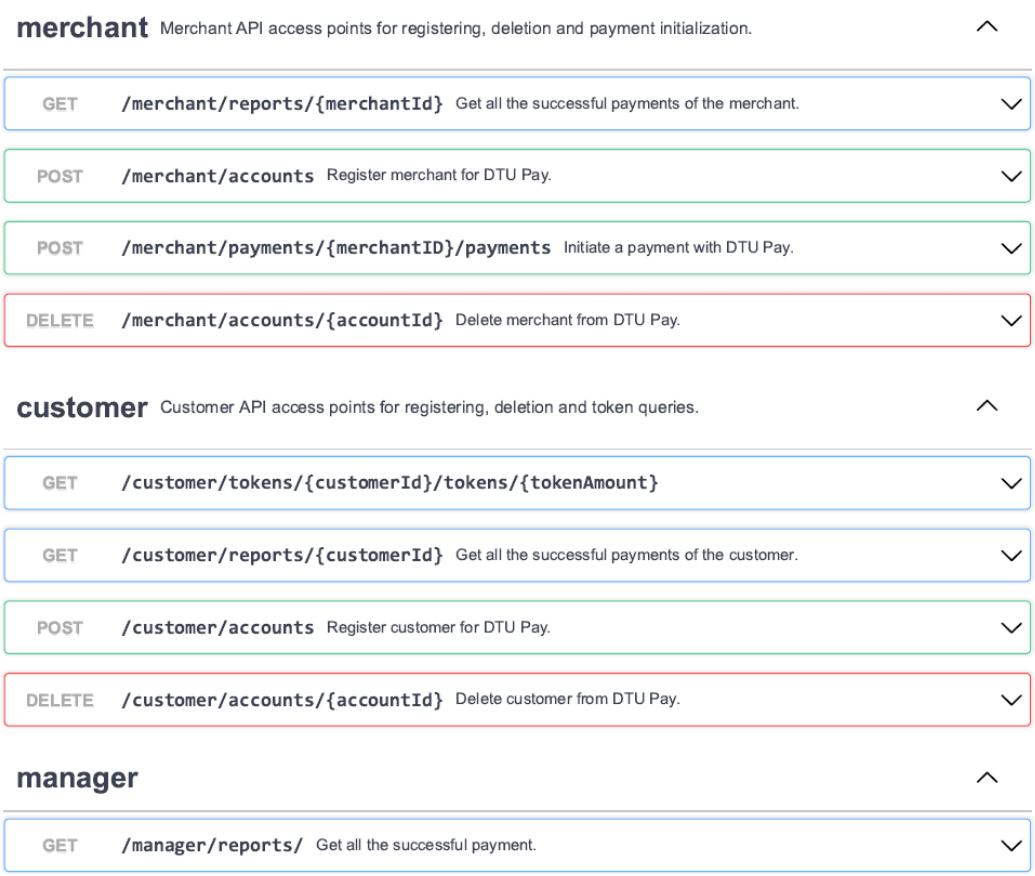

Figure 10: Swagger diagram of the system

3.3 Communication

Author: Florian

As mentioned in section 3.1, there are four different microservices regarding maintaining and executing different payment scenarios: `Account`, `Token`, `Payment` and `Report`, and one microservice, `DTU pay`, for handling the communication to the outside world. Each microservice uses the message queue provided by Hubert in the message-utilities-3.3, and using that implementation, each "event" has its own topic, i.e. routing key. Note, that also the Factory-pattern is used from Hubert's own example code. When an event is send out e.g., `TokenRequested`, we send out a message for a topic specific for `TokenRequested`, and similarly, when we add a handler for the event `TokenRequested`, we are listening for messages for that topic.

For an overview of how services are communicating using the message queues, consult figure Figure 9. We also have cases of Remote Procedure Calls, e.g. when a customer requests tokens, the facade sends out a `TokenRequested` event with the customer ID. The only service listening for that event is the Token service, which before issuing the tokens, sends out an event `AccountCheckRequested`. The Account Service listens to that event, and then checks if the is customer registered, and sends out a `AccountCheckResultProvided` event with the result. Only after that event is received by the Token service, it continues processing the `TokenRequested` event, and then returns the appropriate

response. The `DTU pay` microservice handles the communication to the outside world through HTTP requests. Here the client can send requests to `DTU pay` via the resource paths defined in tables 1, 2 and 3.

4 Group Work

Author: Gunn & Bingkun

At the beginning of the project the team started planning the system together. As a first step, the main events and scenarios which need to be implemented had to be decided on.

The Event storming was done on a Miro board where it was quite comfortable to draw an easy-to-understand event map. After a successful Event storming session the team moved forward with mob programming: Florian was writing the skeleton of the project on his own computer while the others were following him through Teams and giving advice, discussing unclear questions.

Later on the team split up into smaller groups to work on the individual microservices. Josephine and Gunn were working on the Account and Report services, Bingkun and Florian were creating the Payment service and finally Tamás and Bence were doing the Token microservice. Test scenarios have been created for all the individual services and when they all passed, the integration phase began, again via mob programming, conducted by mainly Florian with inputs from the group. At the end of the day the integration was verified by end-to-end tests where everybody took their share of writing different scenarios.

The team work was made effortless by using Jira, where tasks could be easily defined and followed so members could work effectively. The structure of the system was documented as an UML diagram which was created on Lucidchart.

In term of the continuous integration, we also configured the CI/CD pipelien on gitlab to have a direct impression during development. The CI runner of gitlab is also placed on the virtual machine and the CI is divided into two stages: build and test.

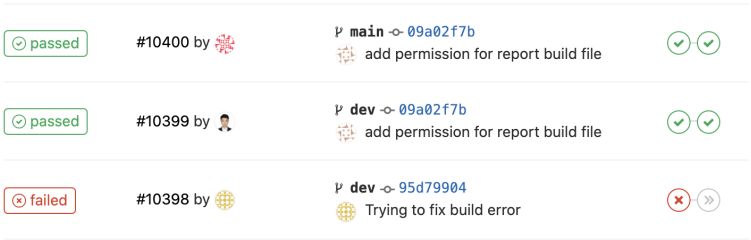

Figure 11: Gitlab CI Pipelines

5 Future Considerations

Author: Josephine & Florian

Seen in retrospective and if the group had the chance to implement any improvements, multiple tweaks would have been considered. In this regards, in the case of a token creation being unsuccessful, a `TokenDTO` with an empty list will be return. This could be improved by including an error message,

which could be returned to client based on the reason for the tokens not being created. Further, improvements could be made by instead of deleting tokens, they get stored in a list of used tokens, and security could be improved by adding expiration time on tokens. A customer could then get a more detailed response in case the token does not work, e.g., token already used and token expired responses.