Table of contents

1 Introduction

This project deals with a payment system DTUpay. The system services merchants and customers. The architecture is a microservice architecture with microservices for Token management, Payment management, Account management and finally a facade to easen communication with each microservice from a client.
The microservices are deployed through docker and the codebase managed through git. A Jenkins server is setup to test and deploy when changes happen to the repository. The microservices communicate through a mix of synchronous and asynchronous communication. Synchronous communication happens via REST, while a message queue, specifically RabbitMq, is used for asynchronous communication. The message queue is used as an event queue that each microservice can subscribe to relevant events. We append our events with correlation ids to know the subject of the event. We have used a messaging utility (provided by Hubert) with small modification that allows strongly typed event handlers. We use a single queue and use the specific event's class as routing key. We make use of dependency inversion were where we though it was relevant, and make heavy use of beans dependency injection management.

2 Event Storming Process and DDD

To figure out how the system should be built and to be able to design and develop a clean and maintainable Domain Driven system, we used the eventstorming model and after going through multiple iterations we found our our domain events, aggregates which gave us an overview and a an idea of the design and which services should exist and what the flow through the different services should look like.

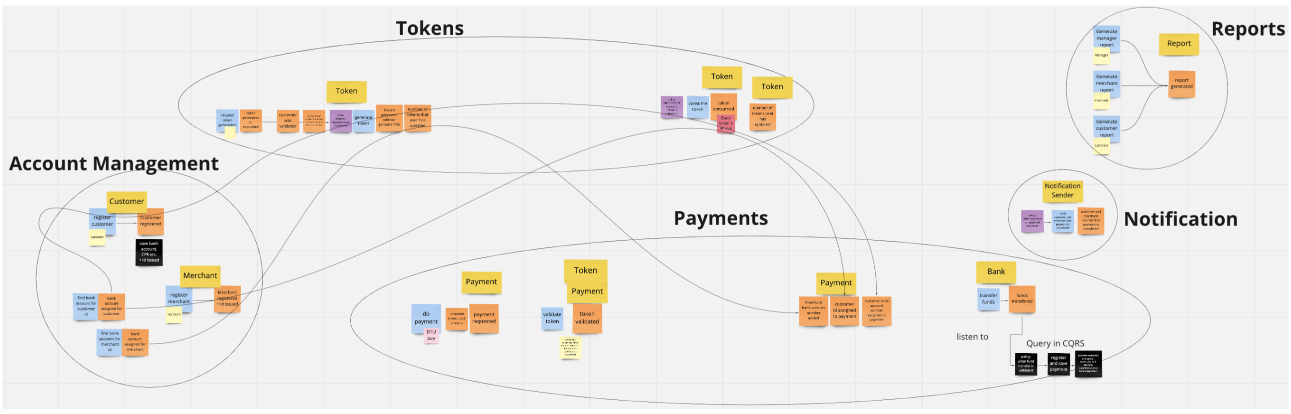

Figure 1: Event Storming Process snippet from Miro

3 Architecture

The final application ended up as seen in the diagram below, where the system consists of 4 microservices. When the application is structured this way, it ensures that it is maintainable and testable, it is loosely coupled and can be deployed independently, allowing for different implementations of the same microservice to be used, if needed without the other services breaking. DTUPay is a facade and has knowledge of all the other microservices.

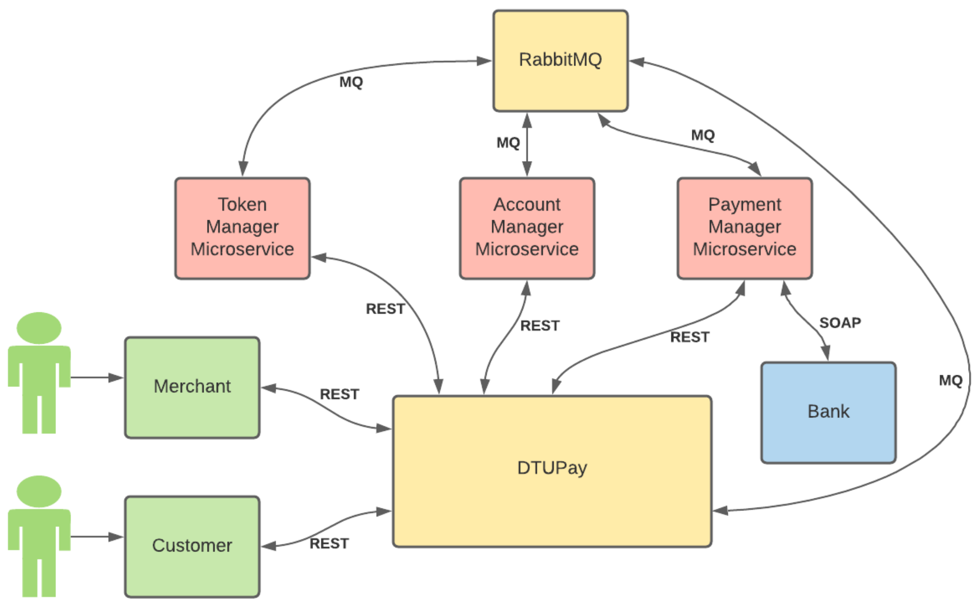
RabbitMQ
MQ
MQ
MQ
Token Manager Microservice
Account Manager Microservice
Payment Manager Microservice
REST
REST
REST
SOAP
MQ
Merchant
REST
DTUPay
Bank
Customer
REST

Figure 2: Architecture

3.1 REST API Design

We argue that our system has "level 2 maturity", according to Leonard Richardson's *REST API maturity model*[3]. In this model, there are three levels of API design to achieve the "glory of rest", taking full advantage of the architecture behind http and what made the world wide web one of the most successful distributed systems. Level 2 maturity is about using resources and http verbs.
How we use resources
The use of HTTP verbs is also part of REST level 2 design, especially when we used the idempotent methods GET and DELETE in addition to POST which is not idempotent.
"Level 3" is about using hypermedia controls to guide a consumer incrementally into how a system should be used. Like visiting a web page and gradually discovering resources you can click on as you advance through the site map. However, in our system that level of design was deemed not practical to achieve as the services do not have an overwhelming amount of resources for consumers to use.
Advantages of REST over message queues. with the facade. Evolutionary design. Easier to set it up, since that is how we started. And according to Sam Newman's *Building Micro-services* book [2], it is always better under conditions of uncertainty to start monolithic and evolve the architecture with time. In the first iteration, the customer API and merchant API were doing REST calls to internal services, but as our system got more complex, we introduced a facade to encapsulate the internal services from the outside. Guided by the Agile principles of KISS and YAGNI, we decided to keep the facade's communication with internal services through REST, instead of refactoring it with a message queue, as it added little-to-none business value.

3.2 Communication and Events

This chart describes the order of messages between microservices in a payment sequence.

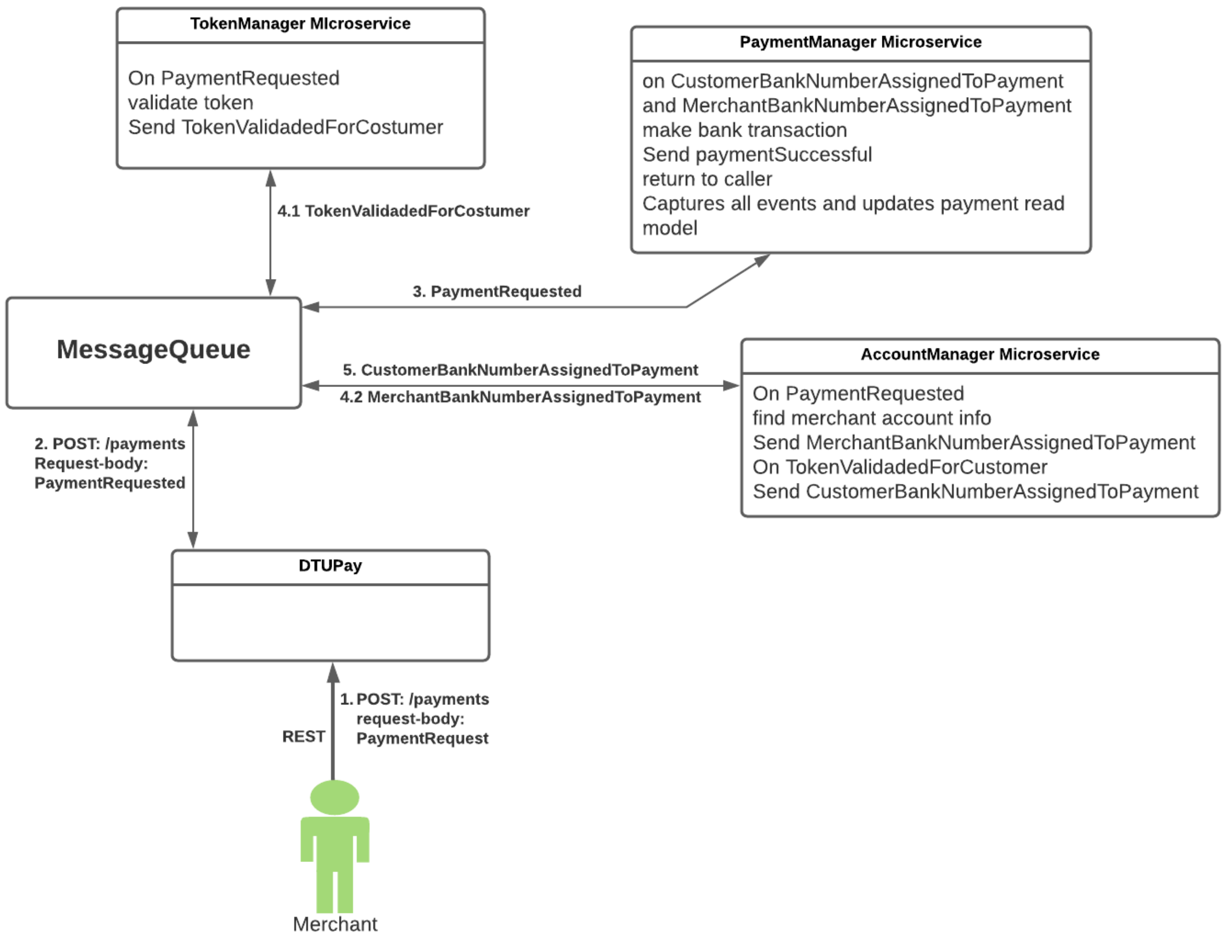
TokenManager MIcroservice
On PaymentRequested
validate token
Send TokenValidadedForCostumer
PaymentManager Microservice
on CustomerBankNumberAssignedToPayment
and MerchantBankNumberAssignedToPayment
make bank transaction
Send paymentSuccessful
return to caller
Captures all events and updates payment read model
4.1 TokenValidadedForCostumer
3. PaymentRequested
MessageQueue
5. CustomerBankNumberAssignedToPayment
4.2 MerchantBankNumberAssignedToPayment
AccountManager Microservice
On PaymentRequested
find merchant account info
Send MerchantBankNumberAssignedToPayment
On TokenValidadedForCustomer
Send CustomerBankNumberAssignedToPayment
2. POST: /payments
Request-body:
PaymentRequested
DTUPay
1. POST: /payments
request-body:
PaymentRequest
REST
Merchant

Figure 3: Message queue

3.3 AccountManager MicroService

This service includes both Customer and Merchant which both contain the attributes: firstName, lastName, cprNumber and bankNumber. The only difference is that Customer pays the Merchant and not the other way around. AccountManager is a REST service that can:

URI Path	Endpoint	Meaning
/customers	POST	Register Customer
/customers/{id}	GET	Get Customer
/customers/{id}	PUT	Update Customer
/customers/{id}	DELETE	Delete Customer
/merchants	POST	Register Merchant

This model indicating the [1] aggregates, values, services and repositories relevant for AccountService.

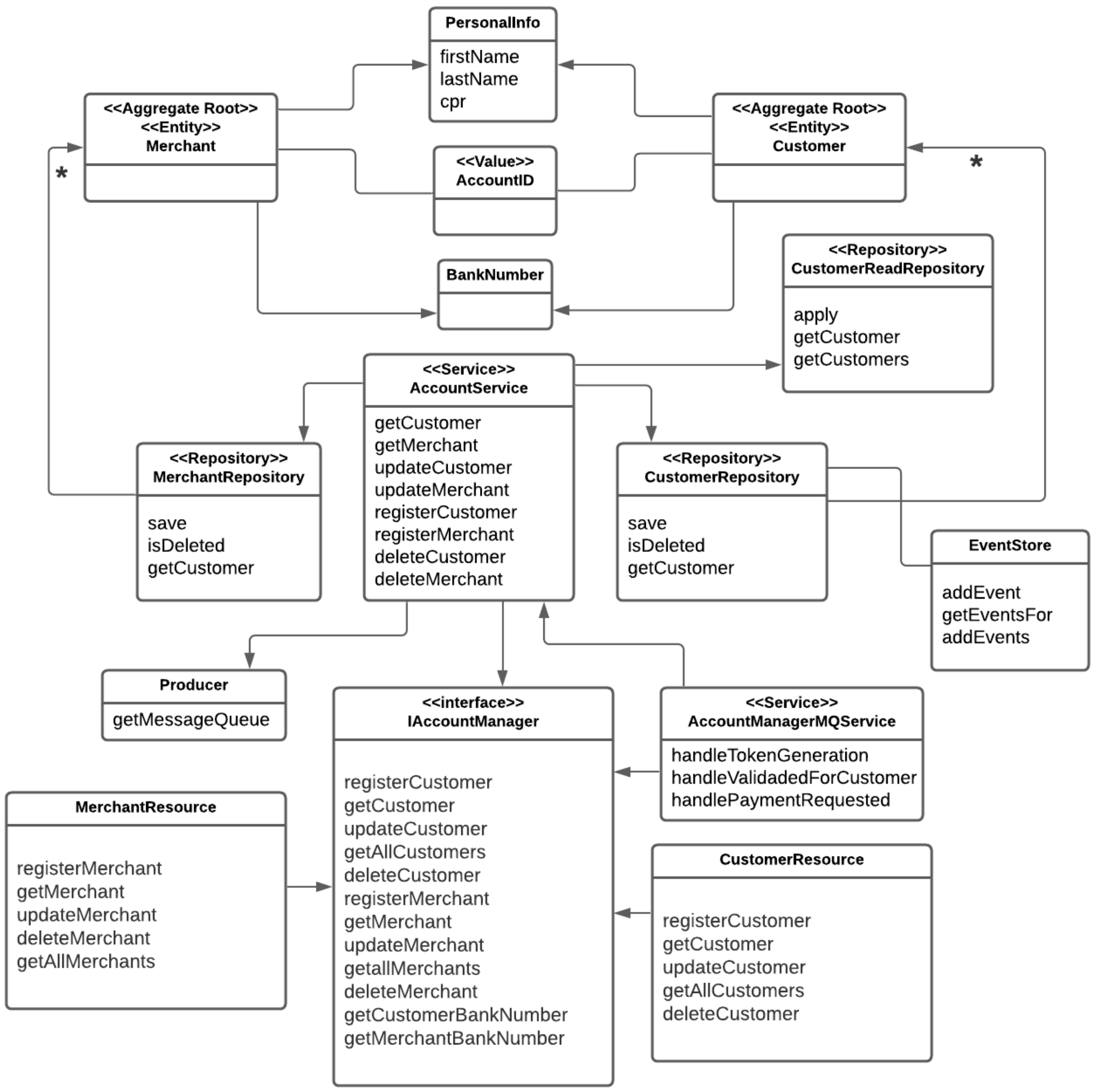

Figure 4: Domain class diagram over Account Service

Due to time limitations only the customer got a repository and have corresponding events to the action that can happen on the customer. The merchant is simply stored in the AccountManager as a map between the id and the rest of the information. The plan was to adapt the customer repository for the merchant.

3.4 PaymentManager MicroService

This service handles payments, the way this works is by a user initiating a payment, and when the payment manager receives the requests it validates that the customer and the merchant exists, and that the custoemr has a valid token, when this is done the paymentmanager publishes a transaction event

URI Path	**Endpoint**	**Meaning**
/payments/customer/{id}	GET	Get Payments For Customer
/payments/merchant/{id}	GET	Get Payments For Merchant
/payments	GET	Get Payments For Manager

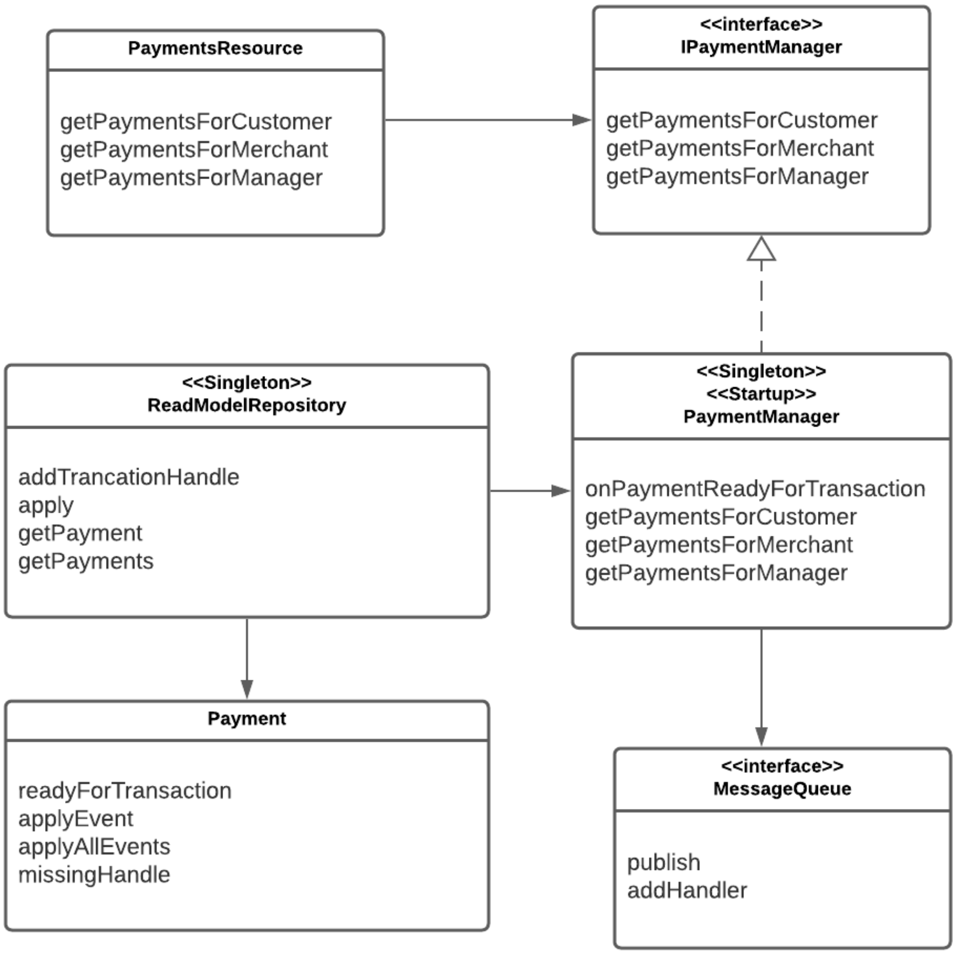

Figure 5: PaymentManager

3.5 TokenManager MicroService

Tokens are used as an object for verification in this app. A Customer can have between 0-6 tokens which are used when paying a Merchant. A Merchant will then ask PaymentManager to validate and execute the transaction. The token is generated using the UUID (universally unique identifier), which makes all tokens unique and completely random. This service is a REST service

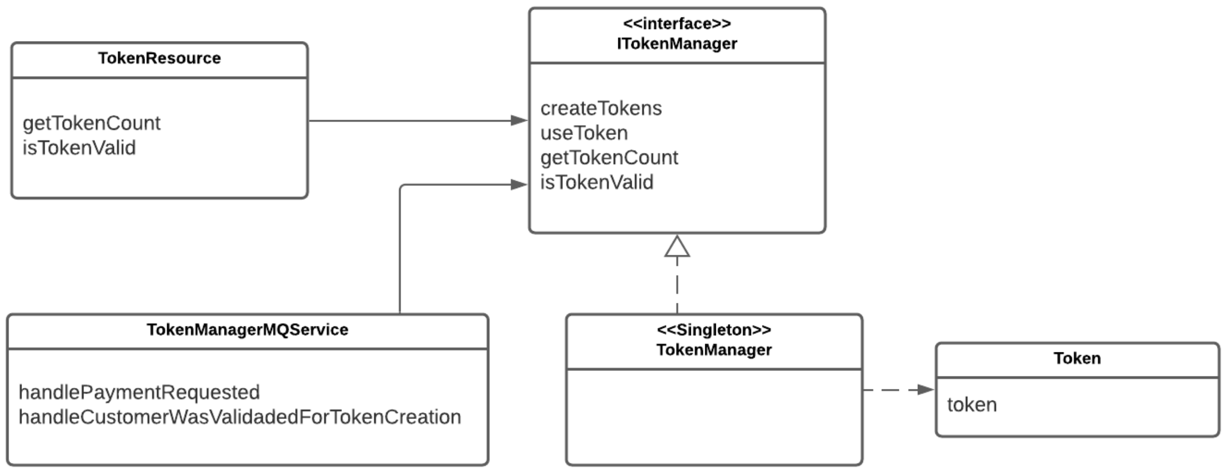

Figure 6: TokenManager

3.6 DTUPay MicroService

The DTUPay microservice acts as the facade of the internal system. It exposes rest resources which are either forwarded to the specific internal service, or translated into the relevant domain events to start an asynchronous task on the internal services. The reason that some functions forwards the rest call to an internal service, was to remove initial friction when we started the project, by allowing direct access to internals and only later proxy them by the DTUPay microservice.

URI Path	**Endpoint**	**Meaning**
/customerAPI/customers	POST	Register Customer
/customerAPI/{id}	GET	Get customer
/customerAPI/customers/{id}	PUT	Update customer
/customerAPI/{id}	DELETE	Delete customer
/customerAPI/tokens	POST	Create Tokens
/customerAPI/payments/{id}	GET	Customer Report
/merchantAPI/merchants	POST	Register Merchant
/merchantAPI/payments	POST	Pay
/merchantAPI/payments/{id}	GET	Merchant Report
/managerAPI/payments	GET	Manager Report

3.7 Error handling

Errors are handled using custom exceptions, message queues and REST API. Starting from the test step definition a REST endpoint is called which is received in the facade DtuPayMs which then forwards it as a REST call to the specified microservice or transforms it into an event to be sent out via message queue. When a REST call action is not successful during one of these flows then an exception is thrown which travels back through the facade to the client where it is being caught. When a message queue event is published on the other hand, and one of the actions or scenarios is not successful then a corresponding event is published. This event is then handled in the facade which throws an exception depending on the event received. The error is then sent back to the client in the same way via a REST call response.

4 Team Workflow

Most of the project have been done remote due to the course being online, however we where able to meet up at DTU Lyngby campus for some days, during the last week of the course. We communicated through Teams in a private group, inside the Software Development of Web Sevices teams channel. A Gitlab repository was created for the code, where we worked with mob programming in the beginning when setting up the DTUpay and when possible we worked in smaller groups and performed pair programming so we could implement more features simultainiously to maintain a good workflow. The reason we chose to use pair and mob programming was to share knowledge and achieve the most learning across the group. All group members have independently implemented at least one cucumber scenario and REST endpoint. Miro was used as a whiteboard for kanban, eventstorming, models and for visualizing architecture. We had a lot of smaller meetings during the day, to talk about updates and what the next task was.

References

[1] Eric Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. Dog Ear Publishing, 2014.

[2] Sam Newman. *Building microservices.* " O'Reilly Media, Inc.", 2021.

[3] *Richardson Maturity Model, Martin Fowler.* `https://martinfowler.com/articles/richardsonMaturityModel.html#level0`. (Accessed on 01/20/2022).