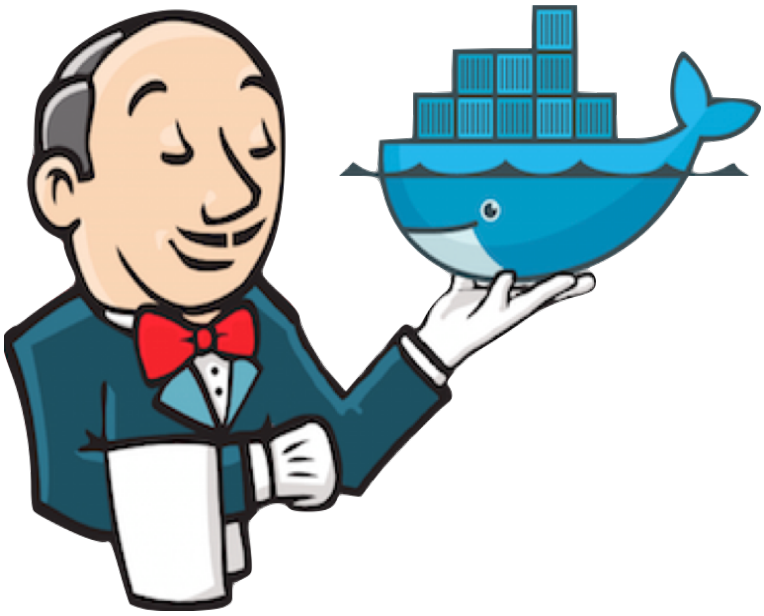
02267 Software Development of Web Services

Group 21 - Project Description



Authors:

Nikos Karavasilis	Melina Siskou	Erik Aske Laforce	Thomas Spyrou
s213685	s213158	s194620	s213161

Verconica De Santos Bernal	Jacob Kølbjerg Hejlsberg
s210098	s194618

Date: January 21, 2022

Table of content

List of Figures

1 Introduction

The focus of this report is to present a possible solution to the DTU Pay exercise as a final project for the course 02267 Software Development of Web Services.

2 DomainDrivenDesign (DDD)

Throughout this project DDD has been kept in mind. Before the project started we were first introduced to the domain via a smaller version of DTUPay where we got to learn what it needed to do. The domain of this project ended up being a payment system keeping merchants and customers stored so they could execute transactions between each other. The domain further included the possiblity of safe transactions via tokens and reporting of the transactions so users as well as system administrators could keep track of what money should have gone where in case of errors. This domain in itself was later split of into smaller contexts later made into individual micro services. These contexts were:

- Account Management
- Payment
- Token Management

While Account and Payment both were included from the beginning, the Token management was included later for safer transactions. These services were only able to do the business logic and were not meant to communicate with an app or website but rather only each other. Users of our web service would instead go through a different context "FACADE", which as the name implies is a form of interface for communication. With the domain now split into smaller bits which had defined purposes, each context could be developed by itself. Having used DDD in this way was what allowed us to implement Token Management easily into the system, since it could get its own business logic developed alone and then payment could be updated to take advantage of the new Token Management. Further the idea of DDD was used to give each of the micro services their own Service acting like the interface for communicating with each other. They also hold their own entities which are kept safe by only being accessible to other micro services through the Service allowing for controlled sharing of information. For each micro service the Service could be used for testing it without the need for the other Contexts to be up and running, and when it came to testing the entire system integration tests could use the facade as an interface leading to insurance that each individual part works alone and together.

3 Hexagonal Architecture

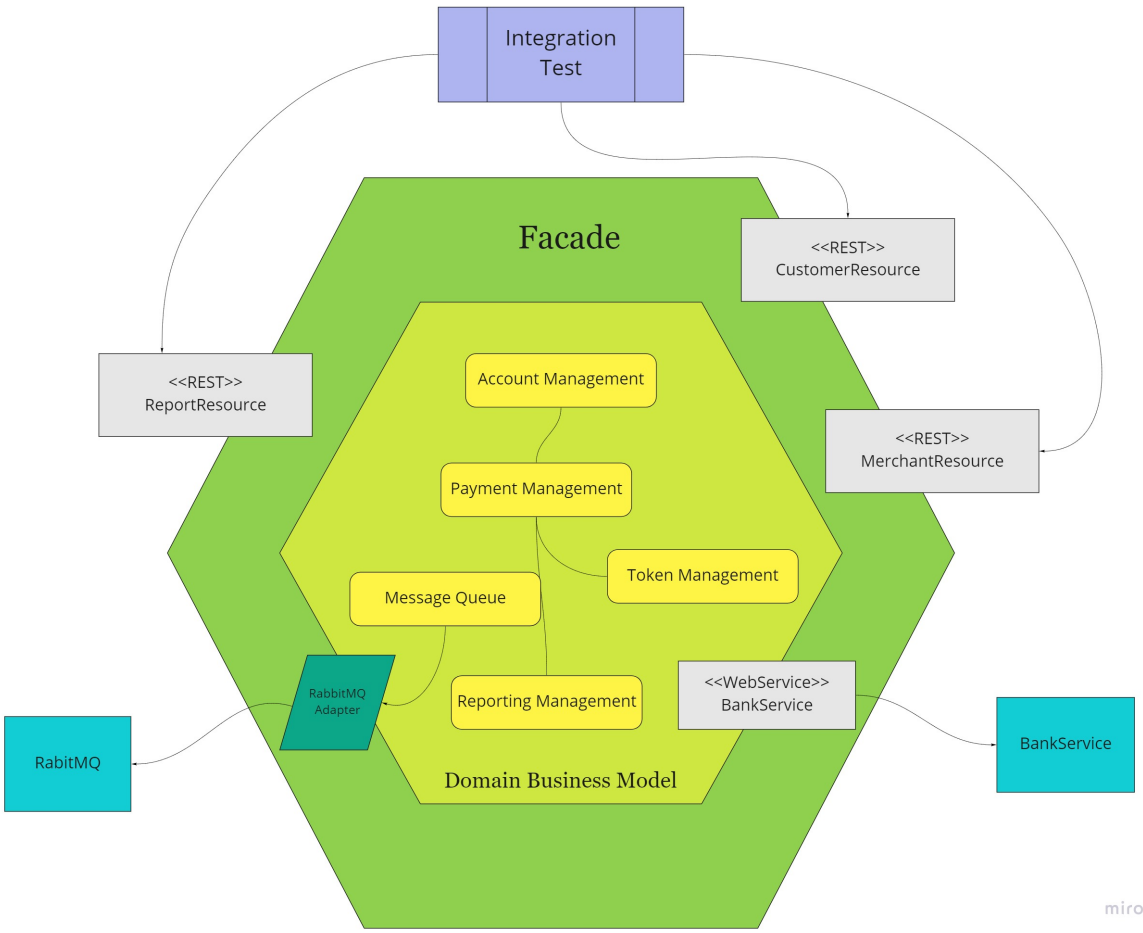


Figure 1: Hexagonal Architecture

Our system was designed in an hexagonal architecture showed in Figure 1, which is an efficient methodology in the form of ports and adapters that protect the business logic.

The business logic in our system consists in all the microservices, each of them is a standalone service that are independent but can interact to each other.

The business logic layer is enclosed inside the inner hexagon, all classes and objects are isolated and they do not depend on the outside world. They can only be accessed by the primary port, the Facade.

Then from the outside we have inbound and outbound adapters.

Three REST inbound adapters - CustomerResource, MerchantResource and ReportResource.

One SOAP WebService inbound adapter BankService - which allows us to access the bank.

The RabbitMQ adapter that allow us to send messages to an exchange, consume messages from RabbitMQ queues and listen for messages in a queue in exchanges.

And the outbound adapter - The Integration Test which is used to test the whole system.

4 Event Storming Process

Event Storming is a brainstorming exercise that work on an application - in our case, DTU Pay. To identify the various domain events and processes that can happen within the system.
Everyone wrote on sticky notes the every event that occurs in the application.

The Event Storming legend commands are the following:

- **Domain Event** Something the occurred that is written as a verb in past tense and it is relevant for the domain.
- **Action** A decision taken by a user or by a piece of software
- **Information** Resulted information from a domain event.
- **People** Someone that interacts with the system or takes decisions.
- **Consistent Constraint** Business constraints or rules.
- **Policy** A reaction of the system to a given event.
- **Hot-spots** Pain point or bottle neck. Anything that needs further clarification.
- **External System** A system maintained by another team or association.
- **Microservices** Small self contained application.
- **Aggregates** Grouped events and commands around. Each aggregate represents a specific business concept that has a local responsibility.

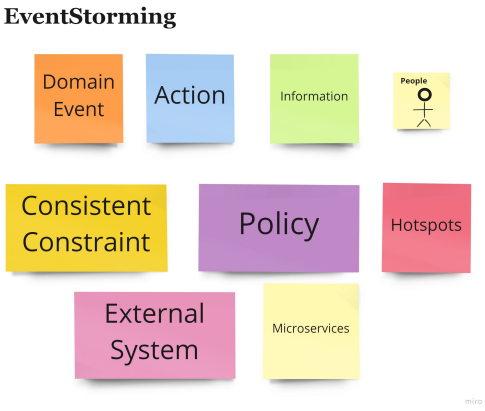


Figure 2: Event Storming Legend

Figures 3 and 4 show the result of the event storming performed by the team in Miro. It was one of the first collaborative exercise for the team to explore the business flow of the project. We started with the domain events, moving through the model to identify the actions and identify the multiple micro-services and how they would interact to each other.

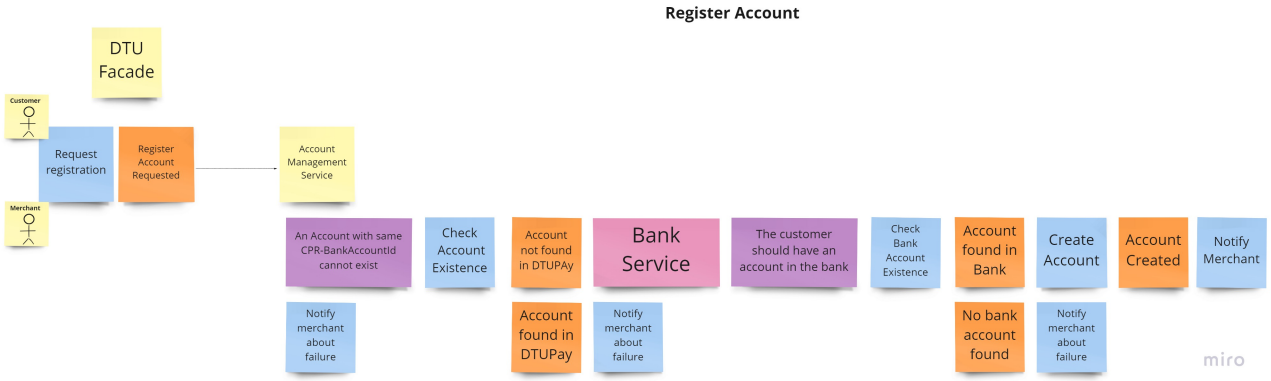


Figure 3: Event Storming Result Account Registration

Both Customer and Merchant can register an account in our system. An event "Register account requested" from the Facade is sent to the Account Management Service.
Once the event is published the Account Management service checks if the account already exists, as a policy an account with the same CPR-BankAccountId cannot exist.
If the account in DTUPay already exists, the failure notification is sent, if the account is not found, the Bank Service is used to check if the customer/merchant have a bank account. The account registration to DTUPay is not possible if the user doesn't have an existing bank account, if that is the case, an event "No bank account found" is sent. On the contrary, provided that the user has a bank account, an event "Account found in bank" is sent, the account is created and the notification is sent.

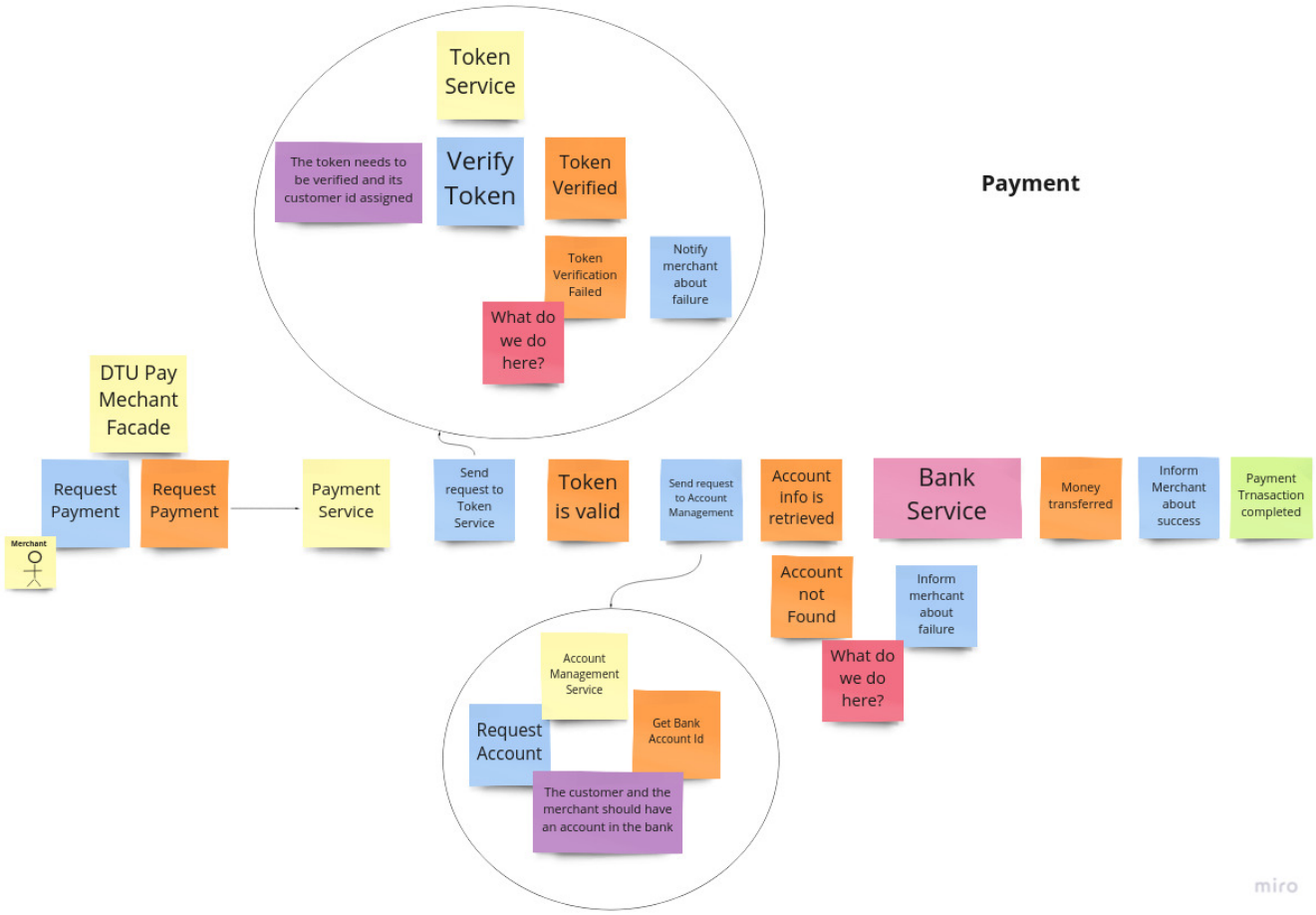


Figure 4: Event Storming Result Payment

When a merchant receives a request for a payment from a customer, the DTUPay system receives a "Payment Requested" message. DTUPay then contacts the Payment Service to begin the payment process. Firstly, the provided customer token needs to be validated. Payment Service sends a "Verify Token" message to Token Management service and waits for a verification response. If verification succeeds, Payment service proceeds by sending first Customer and then Merchant account ids for identification to the Account Management service. Account Management service identifies accounts and sends back a successful response together with the respected bank account ids. The payment can now be performed by contacting the Bank service.

Upon a successful bank transfer, Payment service informs DTUPay that the payment has been successfully handled and the merchant gets a notification message. If however, the token/accounts could not be verified or the bank transfer fails, Payment service informs DTUPay of a bad request and merchant gets a failure notification along with an error message explaining what went wrong.

5 Description of Architecture

5.1 General Overview

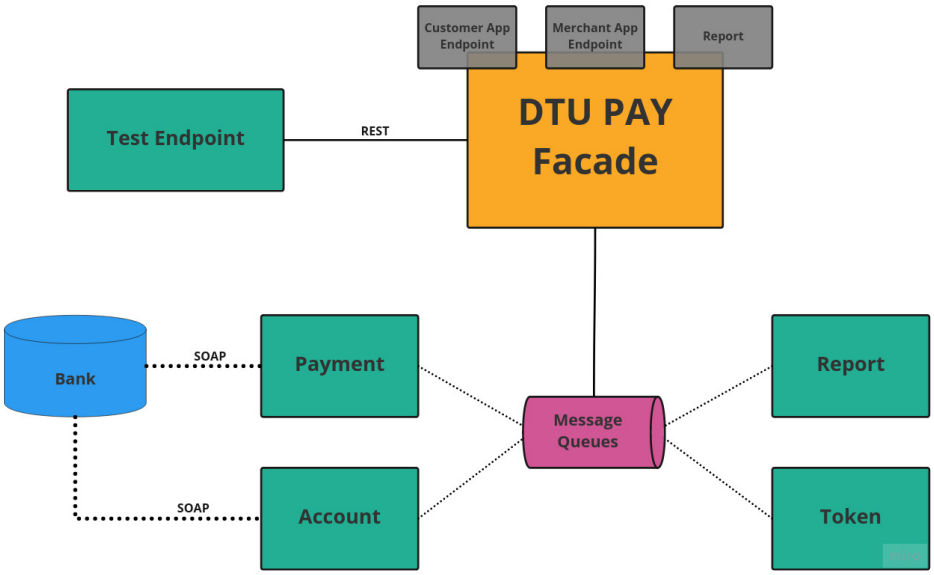


Figure 5: DTU Pay Overview

The architecture of the project consists of four different microservices each one for a specific functionality, the facade which basically is the exposed rest endpoints for the outer world and the last part is the end-to-end test endpoints. Two of the four microservices communicate with the Bank by using the SOAP messaging protocol. The communication among the microservices and the facade is accomplished via queues by using RabbitMQ topics.

5.2 Microservices

As mentioned previously, we have four different microservices each one for a specific functionality. All the microservices do not communicate among them (e.g. the report does not send any message to the token).

The *Account Management* microservice is responsible for managing users accounts. The service can register an account for either a customer or a merchant based on the type provided in the payload. A customer or a merchant will call the REST endpoint in the *Facade* which will place a message in the queue and only the *Account Management* service can consume this message and send a response back. The microservice also consumes messages from the *Payment* to check if an account exist so the payment can occur. The account model representation will contain an account ID auto-generate by DTU-PAY system and the bank account id retrieved from the bank.

The *Payment Management* microservice is used to accomplish the payments. The Facade produces a message in the MQ and the Payment service will consume it. The *Payment* service will communicate with the bank via SOAP, which is responsible to transfer the money. However a payment cannot be done

if the *Token* that the customer has is invalid, once the *Payment* receives the data from the *Facade* will place a message in the queue for the *Token* microservice in order to validate the received token. If the *Token* is valid the flow will proceed to find the bank account numbers, if not it will return a message through the queue in the *Facade*. If the *Account* microservice responses successfully the *Payment* will call the bank to complete the payment.

The *Token* microservice is responsible for generating tokens. A user can request tokens the service will consume the message from the *Facade* and first it will check if the amount of the requested tokens is eligible. If this is the case, the service will generate the tokens and return them to the *Facade* through the queue. In addition, the service consumes tokens when the token are about to use in the *Payment*. If a token is used in a transaction the *Payment* service will sent it to the *Token* to be verified and it will return the account ID of the customer.

The *Report* microservice is the simplest comparing to the other three in terms of logic. Report service only functionality is to retrieve transactions from the *Payment* and return them to the *Facade*. There are three categories to request transactions the manager, the customer and the merchant. The manager can retrieve all the transactions with all the fields that are stored in the *Payment*. In contrast, the customer and the merchant will only retrieve transactions based on their bank account ID.

5.3 REST interfaces

It is not possible to reach any of the microservice directly, but only through the facade by HTTP requests. The URI-Paths and HTTP Methods are specified in the table below.

Customer

URI Path / Resource	HTTP Method	Function
customer/register	POST	Registers a customer user in DTUPay
customer/getToken	POST	Gets up to 5 tokens for a user
customer/{id}	GET	Gets a DTUPay customer account by id

Merchant

URI Path / Resource	HTTP Method	Function
merchant/register	POST	Registers a merchant user in DTUPay
merchant/payment	POST	Excecutes a given transaction
merchant/{id}	GET	Gets a DTUPay merchant account by id

Report

URI Path / Resource	HTTP Method	Function
report/manager	GET	Gets a list of all transactions
report/customer/{id}	GET	Gets a list of transactions containing the user specified by id
report/merchant/{id}	GET	Gets a list of transactions containing the merchant specified by id

Other than the shown HTTP Requests the buisness logic for functions such as deleting accounts and consumption of tokens without payments are in place but were not implemented in the facade due to time constraints. Because of this we only ever use POST and GET. We have also decided to specify different paths for all functions so the path itself tells what the function would be. This could have been replaced by using multiple HTTP Methods on one path (e.g. Customer registration and deletion could both happen through the path "customer" with a POST and DELETE Method). Doing so would be helpful for keeping the paths short when a lot of functions need to get implemented, but since the amount of functions is small we decided against this in favor of readability.

5.4 RabbitMQ

The *RabbitMq* implementation is based on the messaging-queue demo provided throughout the course by Hubert Baumeister. We have one exchange named *eventsExchange* that is responsible of transferring our events. The events are only transferred to a queue if the queue is subscribed to the exchange. Each time a method is registered as consumer, the *addHandler* creates a new channel with rabbitmq and creates a non-durable, exclusive, autodelete queue and binds the queue to the exchange. Then the queue is binded to the exchange in order to listen for events. Our message queue system has only one topic that queues push and receive messages, which means that all queues will receive all messages. However, a queue will only accept a message if the incoming event matches the type of events that the queue is listening for. In other words, the consumer method for each handler will only process the matching events. The main difference with the provided code is that instead of including the correlationId inside the arguments we have included an extra object inside the *Event* class in order to store the correlationId. The reason behind this choice is that we implemented this before the correlation demo was released. At that time we designed the system, so that the arguments contain only information concerning the events and we used the Event class as a way to transport the correlation Id. Below we describe the internal communication for the registration, the request of tokens and the payment.

When we register either a *customer* or a *merchant* a request event is sent to the exchange. The account microservice will receive the event and try to create the account. Then it will push either a successful or failed event back to the facade.

When the customer requests *tokens*, the facade will push a request event to the token service, which will produce an event back for the facade.

After a payment is submitted, the facade will produce an event for the payment service. The payment service will communicate with the token service in order to retrieve the account Id for the customer. If the token service replies with a failed event then the payment will produce a *payment failed* event that the facade will accept. However, if the token replies with a success message then the payment will create an event for the account management, aiming to get the bank accounts for the customer and merchant. Similarly as before, if the information retrieval fails then a *payment failed* event will be pushed. But upon successful retrieval then the payment service will try to complete the payment.

6 Jenkins

6.1 Our setup

Jenkins has served as an important part of our agile workflow throughout this project as well as making CI/CD possible. The purpose of jenkins was to insure stability of each microservice and the entire DTUPAY application. This was done with the following setup: For each microservice with its own repository, we created a jenkins project with a webhook such that the repository was fetched each time er pull request was made or someone pushed a new commit. The jenkins ran a build file that built the project and ran the local tests ensuring that the change made had not destabilized the service. If this check passed it updated the image of this microservice on the VM running jenkins to ensure that we always have the latest functioning version of each microservice. This also meant that if a build of a microservice failed, we could still run integration tests for the rest while someone worked on fixing that microservice. A successful build triggered the downstream job IntegrationTests that tested the entire application spanning all microservices. Given that we always have the newest functioning version/image of each service, the integration tests simply ran containers based in the present images of all microservices. We then ran end to end test stored in the integration test repository in git to test whether any change had affected the application as a whole. If this succeeded we triggered our release pipeline where we took all current repositories, cloned then and ran the build for each one as well as the integration tests (this was to make sure that we did not have a microservice that has previously failed and not had its image updated) and if all succeeded we could copy all repositories to a release folder storing a snapshot of a working application (As we have to hand in a zip file this could be considered continuous delivery).

6.2 A few notes on using jenkins for CI/CD

Having this jenkins setup along with using the trello board - As mentioned in collaboration - has created a reliable and robust worflow that for us has ensured that we could easily consider one feature and test small changes in context of both the microservice and the entire system going from a working application to a working application. We experienced that this also made any possible errors in intercommunication between the microservice transparent while developing and allowed us to fix them as they arose rather finding out late if all services were not integrated continuously. All in all, jenkins, agile development, CI/CD and the trello board has been the bedrock of a good workflow where we could all work in parallel making small incremental changes to the application and getting complete transparency of the current status of our application.

7 Collaboration

Successful software projects require effective team collaboration across the entire development lifecycle. Since the time to deliver the project was short, we had to integrate in our daily workflow different tools and approaches to be more productive and well aligned. For the task alignment and the status of the development of the project we used a Trello board. Tasks was made based on the project description and assigned to a group of two members of the team, however quite often other members of the team were involved if a problem has arisen. We had a stand-up meeting every morning after the QA session in order to update about the development status discuss and any potential problems. In the evening we usually had a hour-longer meetings to resolve any issues, clarify questions and design the next steps.

For the code distribution several different Git repositories for each microservice were used which can be found on Gitlab (All the repositories can be found in the installation guide). Each two-person team worked on daily basis via teams using the pair programming approach. However, in order all the members of the team have a good understanding of the process we rotated the tasks among the teams. Miro was one of the tools which helped us to design and understand the project requirements better, as well as, it was used for the event storming part.

Given the rules of the project, we decided to split the authors of the code based on the pair that contributed the most. However, most of the time other people were also involved in specific task that a team had. As a result the biggest part of the code is actually the result of the collaboration of two or more people.