Technical University of Denmark

Technical University of Denmark

02267 Software Development of Web Services

DTUPay - Project Description

Tobias Rydberg (s173899)
Daniel Larsen (s151641)
Emil Kosiara (s174265)
Troels Lund (s161791)
Sebastian Lindhard Budsted (s135243)
Kasper Lindegaard Stilling (s141250)

Group 11

January 19, 2021

Contents

1 Introduction

As part of the course in *software development of web services* at DTU, we are tasked by the fictional company DTUPay to develop an application backend that handles mobile payments between customers and merchants who use their mobile application. The backend they wish to have developed must function as a service which the mobile applications make use of. The backend service uses an external service that manages the bank accounts and the money transfers between them to handle the financial elements.

In this project we will use elements taught in the course to develop a solution that fulfills the requirements given to us by the customer. We will propose a solution that is comprised of a set of smaller services. One of these services will expose a REST interface to enable communication with the mobile applications and for managerial purposes. Message queues will be used internally to allow for internal asynchronous communication between services. The external bank service is outside of our control, and communicating with it will happen through a SOAP interface.

The design and development of the application was approached in an agile fashion with test-driven development ensuring that test scenarios are written to comply with the requirements before functionality is implemented in practice.

2 Collaboration

Communication between team members is important and was unfortunately more difficult than usual due to the pandemic. In an optimal scenario we would have met physically to discuss the progress of the project, design ideas and so on. As an alternative to meeting up physically, we had voice and video meetings primarily using the Zoom[7] platform. Announcements, notes and other small discussions were conducted over the Discord[1] text chat.

Morning stand-up meetings were held every morning at 9AM where the overall progress was discussed, what our goals for the day were and team members could raise any questions they needed to have clarified. Additionally ad hoc meetings were held on an as-needed basis, typically when unforeseen problems arose.

To plan and manage the tasks of the project from start to the finished product, a GitHub project was created. The project enabled us to boil the requirements from the customer down to smaller tasks and organize them on a Kanban board and track the overall progress of the development. GitHub was used to manage all of the source code produced by the team as well.

After the project requirements had been boiled down into smaller self-contained tasks, the team was split into three groups of two, and these groups were assigned a set of tasks.

2.1 Contributions

In Table 1 a list displaying which group members have been the main contributors of the various elements in the system.

Element	**Contributors**
REST interfaces	Everyone
Client application	s161791 & s151641
Payment service	s161791 & s151641
Token service	s174265 & s173899
Account service	s141250 & s135243
Reporting service	s174265 & s173899
Deployment setup	s161791 & s151641

Table 1: Group members and their main responsibilities.

Besides the main responsibility of the team members as seen in Table 1, each member also participated in design meetings and influenced, provided corrections and improvements for other parts of the system. This was particularly important for making the communication amongst microservices function as intended. Additionally, each team member also participated in an equal manner in writing this report.

3 Requirements

DTUPay offers a mobile payment option for merchants and customers. From the description of the system given to us by the customer, DTUPay, we construct the following lists of requirements that our product must satisfy.

- Customers and merchants must be registered with DTUPay before they can use the service.
- Both the customer and merchant must have a mobile device.
- DTUPay is responsible for the presentation layer. We must develop the backend used by this presentation layer.
- For a customer to be able to pay the merchant, he must present a unique token to the merchant. This token will be consumed and used to pay the merchant.
- A token must be unique and it should practically be impossible to guess an unused token.
- If a customer has spent all of their tokens and it is the first time they request tokens or if they only have 1 token left, they can request between 1 to 5 new tokens. If a customer does not fulfill these criteria, the request will be denied.
- Overall a customer can have at most 6 unused tokens in their possession at any time.
- A token can only be used for one payment and its use is recorded with the transactions in DTUPay and is proof that the payment was authorized.

3.1 Functionality

- A customer must be able to pay at a merchant. This includes transferring money from the bank account of the customer to the bank account of the merchant.
- A customer must be able to obtain new unique tokens.
- Customers and merchants must be able to register themselves with DTU-Pay.
- DTUPay must be able to request transaction reports, summarizing all of the transactions that have been transferred in the entire system.
- A customer must be able to request transaction reports, summarizing all of the transactions they have been part of in a given time period. Each element of the report must contain the amount of money, which merchant and the token that was used. This report will be used to generate a monthly report to send to the customer.
- A merchant must be able to request transaction reports summarizing all of the transactions they have been part of in a given time period. Each element of the list must include the amount of money paid and the token used for the transaction. The customers must be anonymous so their name

must not be included. This report will be used to generate a monthly report to send to the merchant.

- It must be possible to refund a payment to a customer.

4 Analysis

As a preliminary exercise we constructed a domain model to explore and understand the domain and furthermore find some natural splits to get some ideas about potential coherent services. In the diagram there are dashed boxes which are not a part of a standard domain diagram. These are drawn to denote the part of the domain that became an actual service in the implemented system.

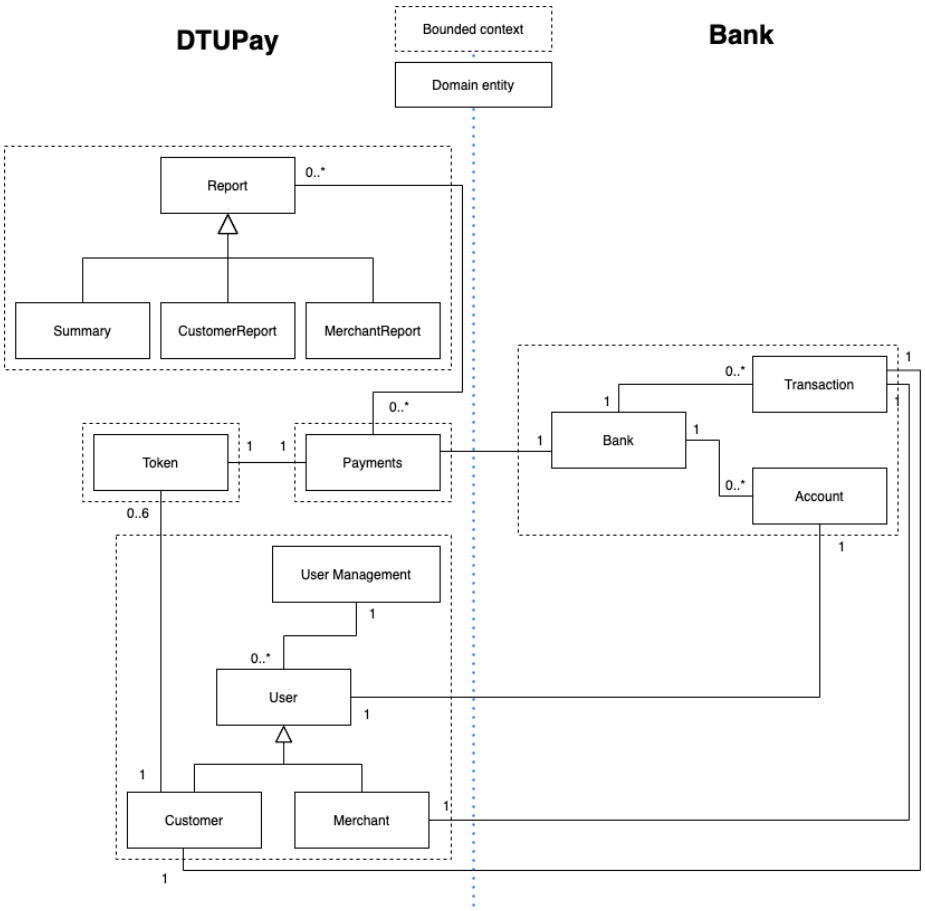

Figure 1: Relational domain diagram extended with service encapsulation indicated by dashed lines.

As part of the analysis we also have to decide which entities communicate with one another to fulfill a requirement in the chosen system architecture. In other words, the different services were delegated responsibilities and therefore the communication between services was visualized to ensure that dependencies for specific requirements could be achieved. Figure 3 depicts the communication between the services. As an example, there must be cooperation between the account service and payment service to perform a payment process. Even though the payment service actually has access to the external bank on it own it is not its responsibility to handle verification of the accounts used in a payment.

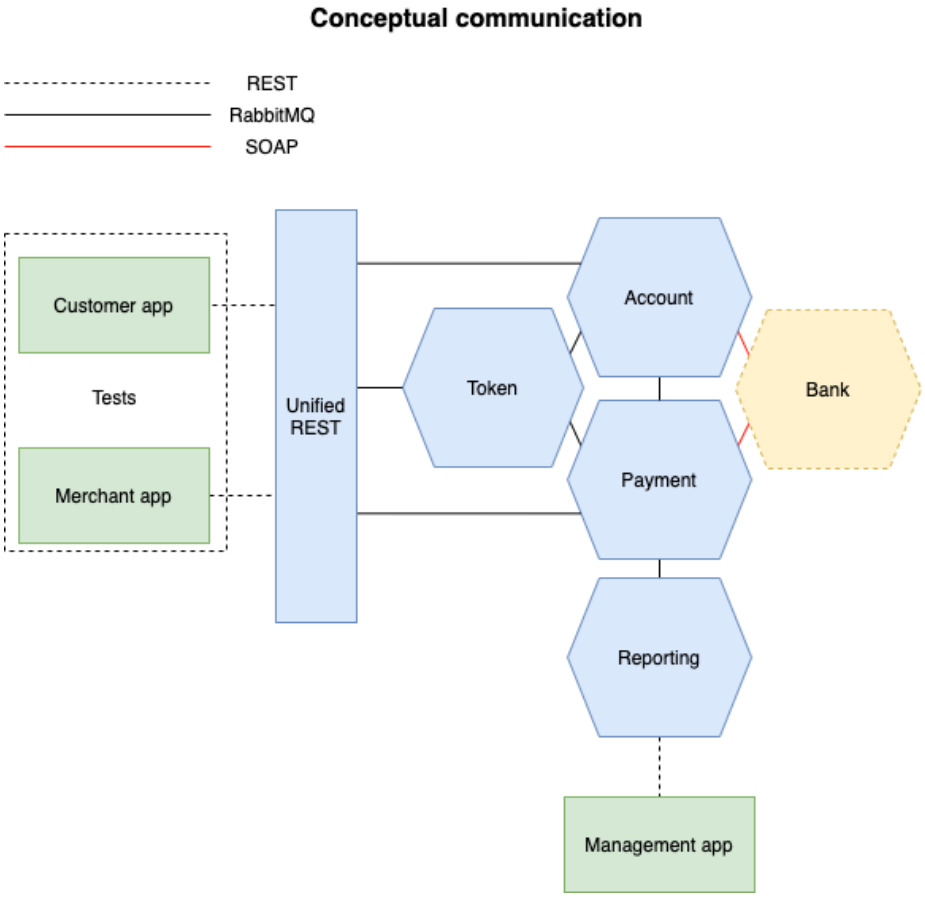

Figure 2: The actual interactions between the various services in the system. Elements depicted in the light blue color are the microservices and their represented communication channels are labeled according to the top left corner. Green elements represent customer, merchant and manager applications.

Both Figure 1 and Figure 3 represents final iterations of the analysis process. Before these, other domain and communication designs were considered but through discussions during daily stand-up meetings and Q&A sessions we arrived at the depicted design. Initially, we had a design where separate customer and merchant services were also included. An initial implementation stage used this design before it was discarded in favor of a the later iteration where these services were merged into the unified REST service.

5 Design

Given the lists of requirements that were defined in Section 3 we start off by designing the overall system architecture. The requirements served as a good starting point for discussions related to the architecture of the system, and some of the elements are the basis of the implemented Cucumber[5] scenarios.

Since we practiced an agile development workflow, this design phase was revisited multiple times while developing the system. As a team we strived to convert the business requirements into a set of Cucumber scenarios and convert them to software tests before implementing any functionality.

While the system could be designed as a monolithic system, we want to split it up into smaller services. Splitting it up into smaller services aided us in managing the development flow, splitting up the work into smaller sub-teams while ensuring that one component did not break the functionality of other components. In the subsequent sections we will break down and design the system as a set of interacting microservices.

5.1 System Architecture

The system is comprised of several microservices that each run in their own separate environment in a Docker[2] container running on a virtual machine we have been given access to by DTU Compute.

Microservices that need to communicate with other microservices within our control will do so by asynchronous messaging using RabbitMQ as the message broker.

This event driven approach has the benefit of non-blocking behavior, where other services are not left in a blocked state waiting for the response of another service, and thus hindering the overall performance of the system. This architecture also make the system very flexible because of very low coupling between components, since none of the services are directly communicating with each other. In other words, the microservices don't know that each other exist in practice.

Microservices that need to communicate with external services will do so using a set of adapters. The mobile applications developed for DTUPay make use of a REST interface, which accepts requests sent to the back end application and sends back responses. As the application makes use of an external bank service, we need to include an adapter for the interface supplied by this supplier. The external bank service supplies a SOAP API, so requests sent to the bank and the responses received need to make use of a SOAP adapter.

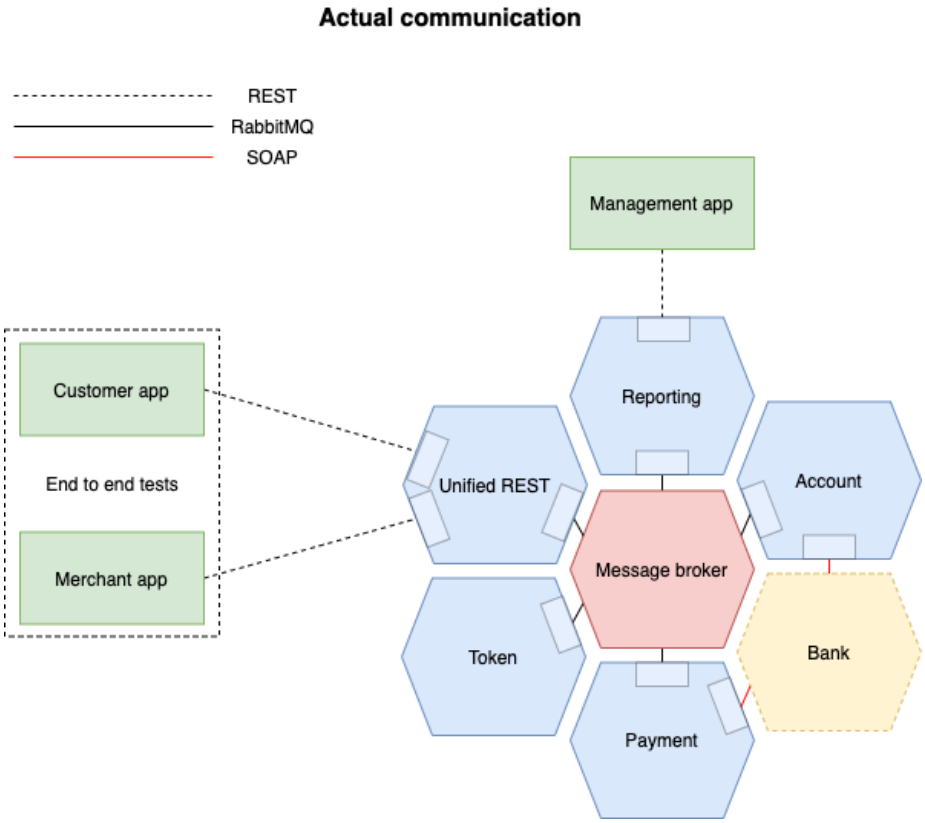

Figure 3: The actual interactions between the various services in the system

6 Implementation

In this section the concrete implementation of the features and functionality designed during the design phase is described.

6.1 The Common Template

Due to the basic architecture of each of the microservices being similar, we decided that a common template made a good starting point for each of the services to be developed from. The template is structured as an example service and attempts to ensure that the services that we develop follow a common project structure. This includes items such as common dependencies and configuration options.

A copy of the template was made for each of the services that we developed, and most of the source lines have since then been replaced.

6.2 Payment Service

The payment service has the responsibility of facilitating the payment and refund process for the customers and merchants. It does so by interacting with the FastMoney bank through SOAP such that it can make use of its money transfer functionality. This also means that the payment service is responsible for sending out messages whether a given payment has been successful or not, such that the customer or merchant will be notified accordingly. In addition to that, the report service will log all of the payments and refunds that go through the system, whether they are successful or failed. This is done for reporting purposes for the customer, merchant and manager.

6.3 Account Service

The purpose of the account service is to keep track of the users that have registered with DTUPay. It is the entity that is responsible for handling account management. As seen in Figure 2 the service communicates through message queues with the payment, token and unified REST services, either to supply information to the services or consume requests to register new users. For example, when a payment is initiated by a merchant, the account service will be notified that it will have to check whether the customer and merchant of the payment are legitimate users and registered within DTUPay.

6.4 Token Service

Token service provides the functionality to request tokens, validate tokens, retire tokens and deliver tokens to the customer. The token service is a standalone service which stores the tokens itself, mapping the tokens to customer identifiers provided by the other services. The most important task of the token service is to validate a given token during the payment process, deciding whether the payment should be successful or not.

6.5 Report Service

The report service intercepts both successful and failed payments in the message queue and logs them in a repository to generate reports for both the customers and merchants. This is done in terms of storing the transaction information between the two parties. The merchant report has been anonymized by removing information regarding the payment amount in the transaction as well as customer identifier. Additionally, it is also possible for a manager to retrieve a complete overview over the payments going through the system.

6.6 Utilities

The utils project contains elements that are shared between all the microservice projects. This includes common exception types, models and data transfer objects. This makes the code-base more streamlined and makes sure that duplicate code does not exist across the services. It is especially helpful regarding the data transfer objects as these have to be similar for both the producer and consumer of the object.

6.7 Deployment

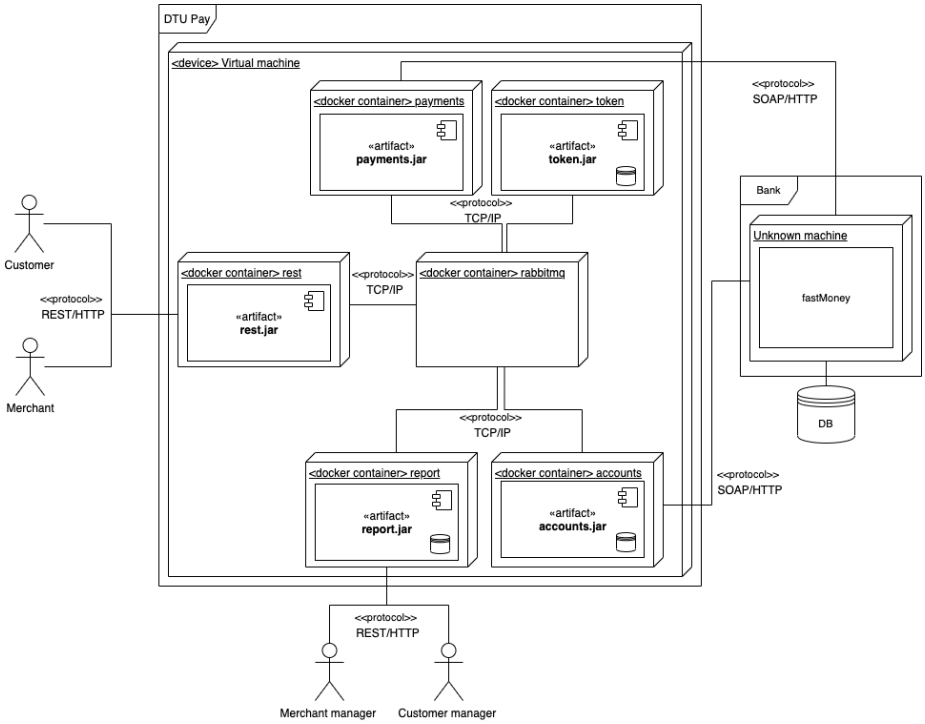

Figure 4: Deployment of the services in the DTUPay system

Figure 4 shows on an instance level how the system has been deployed to the virtual machine. Some minor details have been left out to keep the diagram

simple without too much noise. For instance, as mentioned before, all of the microservices depend on the utils project for common exceptions and data transfer objects. How this dependency works with the payment-service as an example, can be seen in Figure 5.

The system that this report covers and has been developed solely by Group 11 is the system in the DTUPay context, except for the RabbitMQ container which is a docker image.

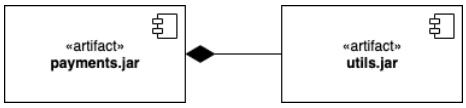

Figure 5: Example relationship between payments and the utils project. Every other project in the system also depend on utils in the same way.

As described in Section 5.1 all the internal communication is through a message broker this communication is using the the application level protocol AMQP to exchange messages, and the communication to the consumers if the service will use HTTP.

In the diagram there is placed a database icon in some of the containers this is to show that's these containers hold data and in a real world setting would have been using there own database to persist data, but in our data just kept data in memory.

6.8 Deployment to the server

The configuration of the containers can be found in a docker compose-file in the root of the project. This file setup how the containers should be deployed. The Jenkins build server will execute this with docker at the time of deployment. It has been set up such that whenever a push operation happens in the repository, the Jenkins server will start a new build of the system and run all tests.

Each of the containers contain an artifact that runs when the container is starting up. Because all the services communicate with RabbitMQ it is crucial that this services is up and running before the other services will try to connect. This way the docker compose-file describes a health check on the RabbitMQ management port and only when the RabbitMQ container answers on this port the other services will run.

6.9 REST interface

The REST interface serves to provide an API which other external application can use to use the service that we provide. Table 2 provides an overview of the interfaces exposed to customers, merchants and managers. The requests sent to the REST API and the responses receives from the API are *JavaScript Object Notation* objects encoded as UTF-8 strings.

URI Path = Resource	HTTP Method	Function
`/account`	POST	Register a user
`/account`	GET	Retrieves all users
`/account/{id}`	GET	Retrieve a user by ID
`/account/{id}`	DELETE	Retires a user by ID
`/account/by-cpr/{cpr}`	GET	Retrieve a user by CPR
`/account/by-cpr/{cpr}`	DELETE	Retires a user by CPR
`/payment`	POST	Perform a transaction
`/payment/refund`	POST	Performs a refund transaction
`/token`	POST	Requests tokens
`/token/{id}`	GET	Retrieves tokens of a user by ID
`/token/{id}`	DELETE	Retires tokens of a user by ID
`/report/customer/{id}`	GET	Requests transactions of a user **Note:** port 8083
`/report/merchant/{id}`	GET	Requests transactions of a merchant **Note:** port 8083
`/management`	GET	Request summary and transactions. **Note:** port 8083

Table 2: REST interface overview

No matter which URI is called, it results in internal event-driven communication between the services using RabbitMQ. In our implementation all the services are communicating using a single message queue of the topic `dtupay.*`.

The different messages put in the queue relative to each URI are described in great detail in the provided User's Guide so the types of events will not be listed here. Please consult the User's Guide.

When designing RESTful webservices, it is important to consider the notion of Richardson's maturity model[3]. When we designed these webservices, we made sure to have ROA (*Resource-Oriented architecture*) in mind as well in cooperating with the maturity model. Considering the maturity model, the implemented REST interfaces comply with level one since we have implemented different REST resources and are not using the same endpoint for all communication, like you typically would in an RPC situation. This is the idea behind ROA.

Considering the second level of the maturity model, our system complies somewhat. We are in all of our REST endpoints using the correct HTTP verbs. For example, when retrieving data GET is always used. POST is used when the request will create data in the system and DELETE when data will be deleted. On the other hand, the HTTP responses that the system is sending back could be improved. For example, when an account is registered in the system, a 200 OK message is returned and not a 203 CREATED. The system however differentiates correctly between 40X and 50X messages for client side and server side errors respectively.

For the final level of the maturity model, the system is not compliant. We chose not to prioritize this level as it was not essential for developing a functioning

system. If the system were to be developed and improved further, it would definitely be a good idea to implement this level as well. For example, it could be used in the report service when retrieving either a customer or merchant report and the request could then send associated URLs with a month added and subtracted from the date given in the request. Another example could be to send relevant URLs when registering an account, for example one that will retire the newly created account.

7 Testing and Continuous Integration

Software testing is key to delivering good software products. During this project we practiced a *test-driven development* approach, where tests are written before the application functionality is implemented. The scenarios that describe the wants and needs of the customer are written in the Gherkin[4] language, which is close to natural language and therefore easy to read and understand by people that are not software developers. The Cucumber[5] testing framework aided us in converting the scenarios into JUnit[6] tests, and running them with variable parameters that are defined in the scenario files. To our best ability we strived to isolate every scenario to verify a specific functionality, but to sometimes multiple functions and logic was tested in a single scenario if it could not be avoided.

The project was tested on mainly two different levels:

- End-to-end
- Service level

The end to end tests strives to provide a verification of the system's functionality as a whole. As an example it was used to verify that interaction as a customer through the REST API is working as intended, which essentially is black-box testing where the input and corresponding output determines the evaluation of some use case.

When testing the services, the tests for a single service is run in isolation from all other services. This lets us observe the internal logic of each service and verify the correctness of the different functionalities it provides. This is done using Jenkins as a continuous integration pipeline, where the testing and deployment is automated.

The Jenkins-file can be found in the root for the project and describes our pipeline to be run on the Jenkins service.

The pipeline is constructed so it contains the following steps

- Compile and build the Maven projects into jar files.
- Build docker images and run them all with docker-compose.
- Service level tests
- End-to-end tests

Jenkins have been setup to automatically deploy to the server every time a the git repositories receives a new commit via a web-hook. The Jenkins pipeline will ensure that only a working version of the system is deployed by running both the end to end and service level tests.

8 Conclusion

The final software solution that was developed for the fictional company DTU-Pay fulfills the list of requirements listed in Section 3. By practicing an agile test-driven development workflow throughout the development process we ensured that the individual components are delivered with a series of tests that are used to verify that the functionality that we want is present while aiding future developers in refactoring existing implementations and adding new functionality without breaking the currently deployed version.

At the beginning of the project, most of the team members had little to no experience with continuous delivery, continuous integration and container technologies. It proved to be both interesting and easy to get started with using such technologies and we feel that they have much more to offer beyond the functionality that we made use of throughout the project. We really wanted to split the master Git repository up into one repository for each service, but this had to be left as a future exercise.

As noted in Section 6 we do not serve a Level 3 REST API to clients who use our API. This functionality was omitted primarily due to not being a requirement and we felt that the time we had was more valuable if spent on other aspects of the system.

8.1 Discussion

During the development of the application we faced various challenges. This section outlines some of the challenges we faced, and are primarily left as lessons-learnt for future projects.

Getting the microservices to communicate with each other proved to be more difficult than initially anticipated. Developing several microservices that communicate requires well defined interfaces so that the developer knows what to expect and how to use the API. Usually these specifications are available to the developer, but as the services we needed to communicate with did not yet exist, some refactoring was necessary to make them work together. Some of the refactoring could have been avoided here, if more time was spent in the design phase to define a set of expectations for the API specifications.

As the project grew in size and complexity it became increasingly difficult to refactor parts of one microservice without affecting other services due to changes in the structure of *data transfer objects*. Given the time for additional refactoring iterations it is possible to restructure the DTOs in a way such that this would be less of an issue.

References

[1] Discord. (2021) Website. Seen 19-01-2021. [Online]. URL: https://www.discord.com/

[2] Docker. (2021) Website. Seen 19-01-2021. [Online]. URL: https://www.docker.com/

[3] Martin Fowler. (2010) Richardson's maturity model. Seen 19-01-2021. [Online]. URL: https://martinfowler.com/articles/richardsonMaturityModel.html

[4] SmartBear. (2021) Gherkin syntax. Seen 19-01-2021. [Online]. URL: https://cucumber.io/docs/gherkin/

[5] SmartBear. (2021) Website. Seen 19-01-2021. [Online]. URL: https://www.cucumber.io/

[6] The JUnit Team. (2021) Junit 5. Seen 19-01-2021. [Online]. URL: https://junit.org/junit5/

[7] Zoom. (2021) Website. Seen 19-01-2021. [Online]. URL: https://www.zoom.us/