Technical University of Denmark

02267 Software Development of Web Services

Authors

Adrian Zvizdenco - s204683
Simon Hermansen - s204712
Adrian Ursu - s240160
Jonas Kjeldsen - s204713
Jeppe Mikkelsen - s204708
Paul Nelson Becker - 194702

January 24, 2025

1 Introduction

Throughout this report, we will explore the various aspects of our development process towards delivering a fully functional microservice application exposing a REST API allowing `Customers` and `Merchants` to register/deregister, pay, and get logs of payments. This includes detailed discussions of our work methods, design decisions, implementation strategies, and testing methodologies.

We will provide insights into the technologies and frameworks chosen, the ideas behind key architectural decisions, and how we addressed challenges encountered during the agile development lifecycle.

Finally, we will evaluate the end product against the initial requirements and objectives, highlighting its strengths, and identifying potential areas for future improvement.

2 Design

The design process used the methodology of **event storming** to collaboratively model the requirements of the system and align the vision of the team. During the event storming session, the entire team participated in identifying the key events, services, and workflows that form the backbone of the DTUPay system.

2.1 Event Storming

This approach ensured that everyone had a shared understanding of the services required, how they should interact, and the expected behaviour of each component. By visualising the system's flow and pinpointing potential bottlenecks or ambiguities early on, the team could establish a clear, unified vision.

Furthermore, the event storming session served as a foundation for defining the system's architecture, helping us break down complex requirements into manageable, well-defined services. It also fostered collaboration, promoted knowledge sharing, and ensured that the design aligned with the project's goals and constraints.

We decided to follow the usual Event Storming conventions and stuck to the following legend:

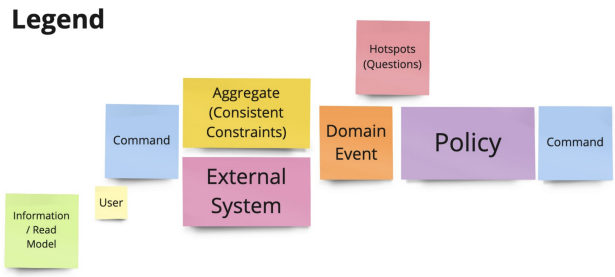
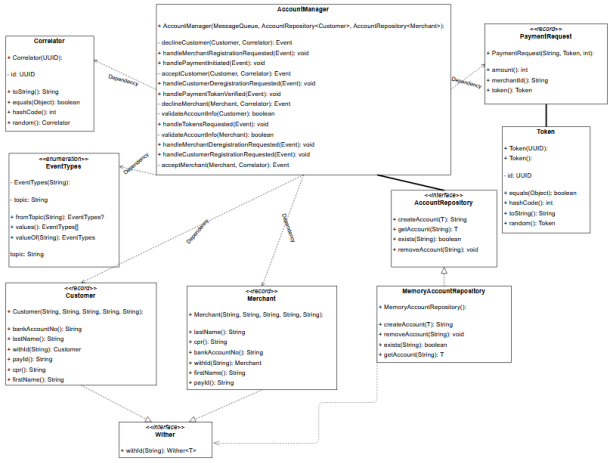


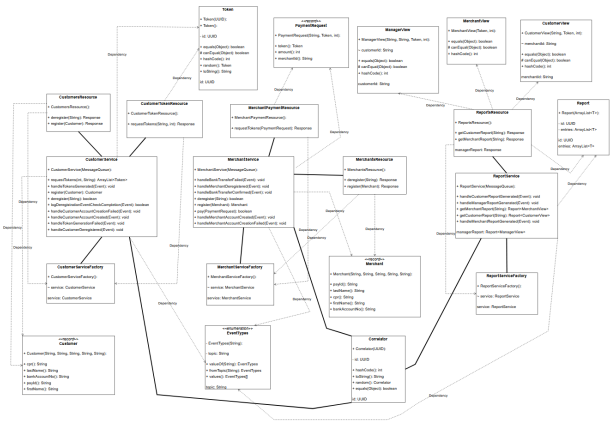
Figure 1: Event Storming colour scheme

2.2 UML Class Diagrams

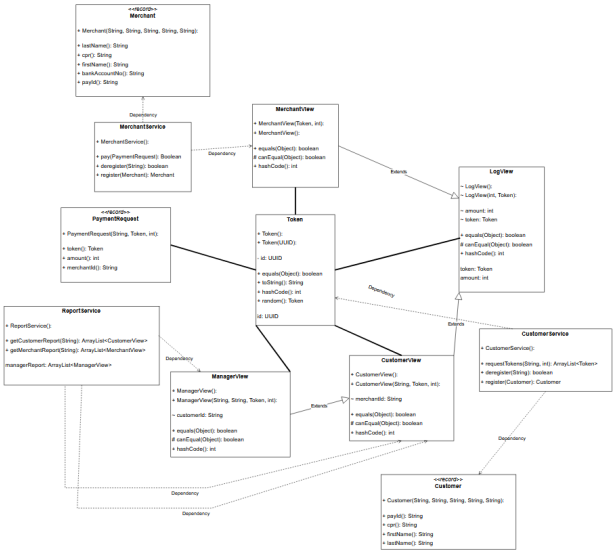
The following UML class diagrams illustrate the classes responsible for implementing the business logic in each of the microservices. These diagrams provide an overview of the key components, their attributes, methods, and relationships. Each microservice is designed to encapsulate specific functionality, ensuring modularity and separation of concerns.



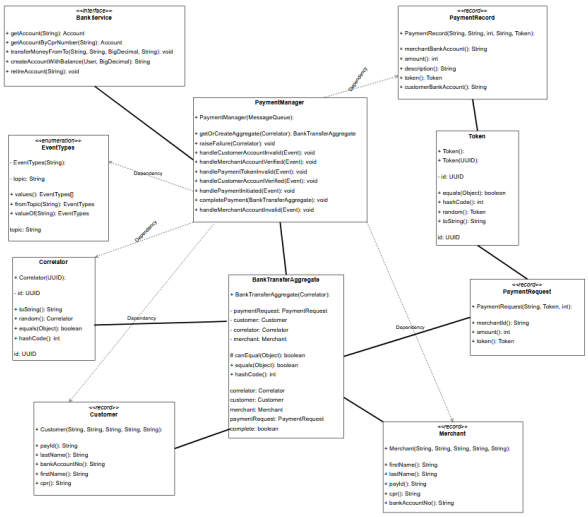
(a) Account Management



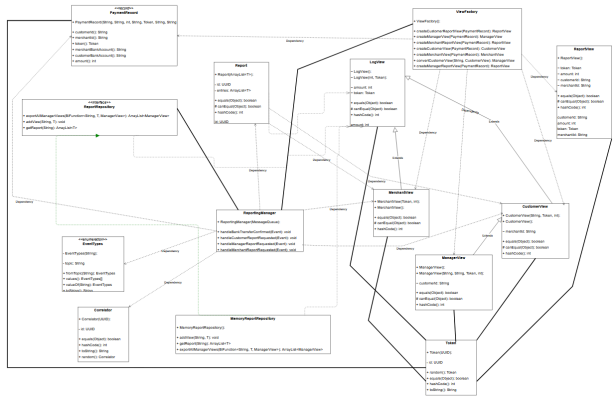
(b) DTU-Pay Facade



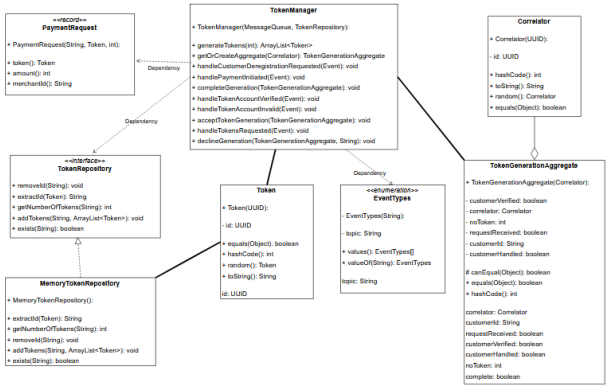
(c) End-to-end



(d) Payment management



(e) Reporting service



(f) Token management

Figure 2: UML class diagrams representation of all microservices

3 Features/Scenarios

In this section, we will describe the implemented features, and include an overview of how each feature was realised, as well as how they complement each other, to achieve the desired functionality of the payment service.

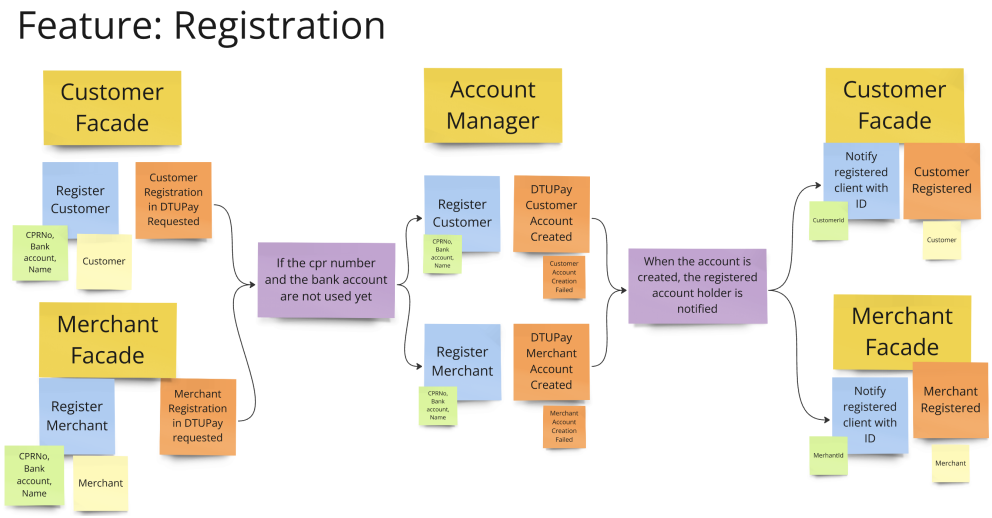


Figure 3: Event Storming Flow for Registration

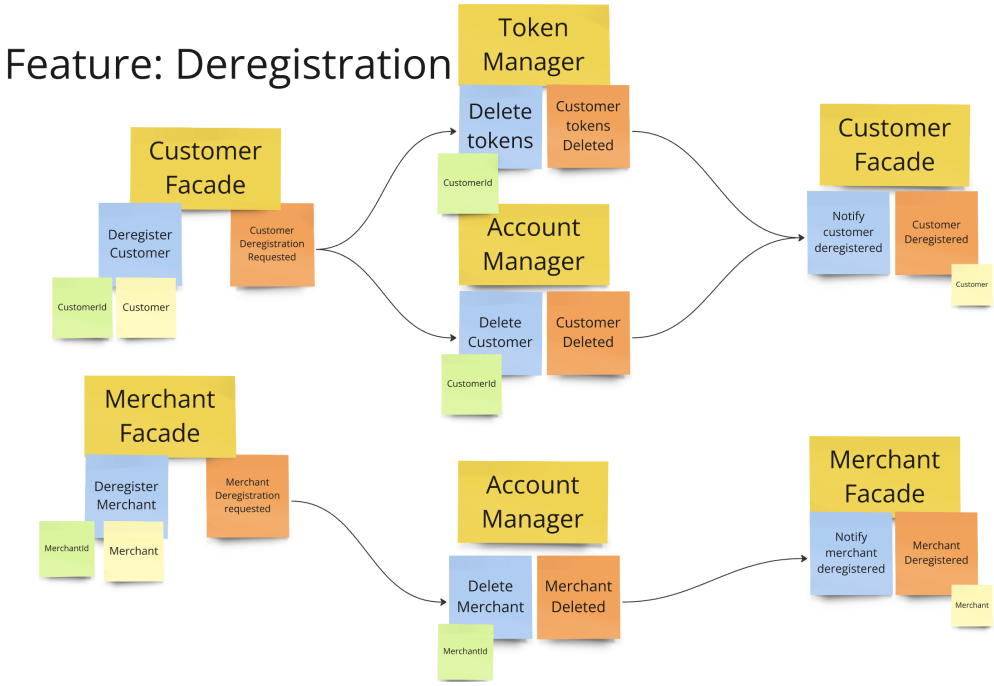


Figure 4: Event Storming Flow for De-registration

3.1 Registration of Customers and Merchants

The goal of this feature is to allow a `Customer` or a `Merchant` to register in the DTUPay application through the REST API interface, in order to gain access to the other key functionalities of their respective roles in the application. The expected outcome of this feature is the introduction of unique `Customer`/`Merchant` IDs gained from the `Account Manager` once their credentials have been verified.

The registration feature is identical for both `Customer` and `Merchant`, so we will describe the `Customer` scenario and implementation, as the same approach applies to `Merchant`.

3.1.1 Scenario Example

The feature is initiated through a `POST` request to the DTUPay facade exposed as a REST API as a new resource would be created on success. The facade then forwards this request to the account management through the **RabbitMQ** channels as an `"CustomerRegistrationRequested"` event. Upon receiving this event, the account manager will assign the received customer object a unique ID, and publish the a `"CustomerAccountCreated"` event containing the ID as a field. This will then be communicated back to the facade, which in turn generates a response to the customer, providing them with a unique ID and status code 201(`CREATED`), as the customer has been successfully stored.

In order for the customer to use DTUPay, they are mandated to provide valid information. For example, a customer is required to provide a bank account number to perform a transaction. The account manager, before storing a generated ID, will validate the account info. If the validation check fails, the service will trigger a `"CustomerAccountCreationFailed"` response event. This in turn will be handled by the facade that will produce an HTTP 400(`BAD REQUEST`) error code as a response, which will be communicated back to the customer's client application since the provided input was invalid and therefore a bad request.

3.2 De-registration of Customers and Merchants

The goal of the de-registration feature is to completely remove the user from the platform through the platform facade interface. The expected outcome of this feature is the deletion of unique `Customer`/`Merchant` IDs obtained from the Account Manager after validation of credentials and potentially a clean-up of active payment tokens created for the customer.

3.2.1 Scenario flow for deregistering a customer

The feature starts with a `DELETE` HTTP request sent to the DTU-pay REST API on the `/customers/{cid}` endpoint, meaning a specific customer account is requested for removal. The facade then forwards this request to the account manager and to the token manager as a `"CustomerDeregistrationRequested"` event. After receiving this event, the account manager will delete the received customer according to their ID, and create a `"CustomerDeleted"` event. The token manager will follow a similar flow, deleting the tokens assigned to the customer, and producing a `"CustomerTokensDeleted"` event. The events produced by the token manager and account manager are both necessary to formulate a REST response from the customer facade, producing a status code of 200(`OK`), and notifying the client app that the customer has been deregistered successfully.

3.2.2 Scenario flow for de-registering a merchant

The flow for deregistering a merchant is almost identical to the customer de-registration, with the exception that we do not require the involvement of the token manager in the process.

3.3 Token Generation

Before a customer is allowed to perform payments, they are required to have at least 1 valid (unused) token. Tokens are a key part of the payment process, as they are used to validate customers in the transaction itself, without exposing the customers' identity, but are also logged in the reporting system. Hence, tokens are a means to protect customer privacy, while maintaining accessibility to the system functionality

3.3.1 Scenario Example

The feature is initiated through `POST` request to the rest API, as we want to update a resource on the web service, that can be retrieved and used by the customer. The `POST` requires a valid customer ID and the amount of tokens they desire to be generated. The DTU-Pay facade receives the request and produces a "GenerateTokenRequest" event, that triggers the token manager service, to produce the specified amount of tokens upon validation of the customer account and the limit of active tokens. These objects are stored in the token manager for the payment process, through an in-memory bi-directional map that enables fast

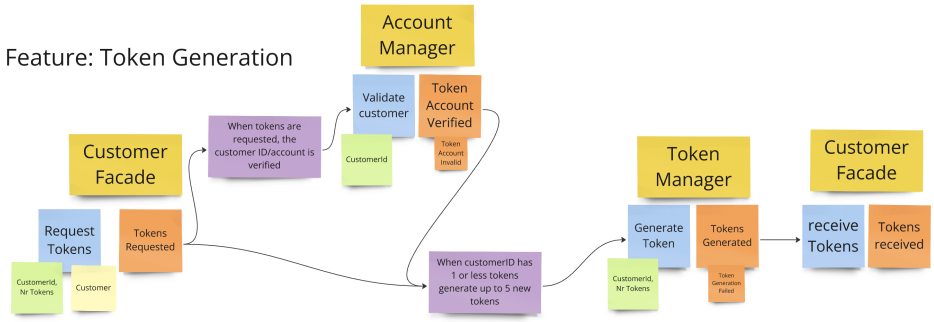


Figure 5: Event Storming for Token Generation

retrieval of tokens associated with a customer and the customer holding a given token. When tokens have been generated, a "TokensGenerated" event is produced, in response to which the service facade will return the requested amount of tokens to the customer client, producing a status code of 200(`OK`) and a serialized payload containing a list of tokens.

3.4 Payment

This is the core functionality of DTUPay. A merchant can initiate a payment using a customers valid payment token. The payment is carried out using the customers and merchants accounts with the bank. A successful payment is also logged such that it occurs in its related reports. The event flow is shown in Figure 6.

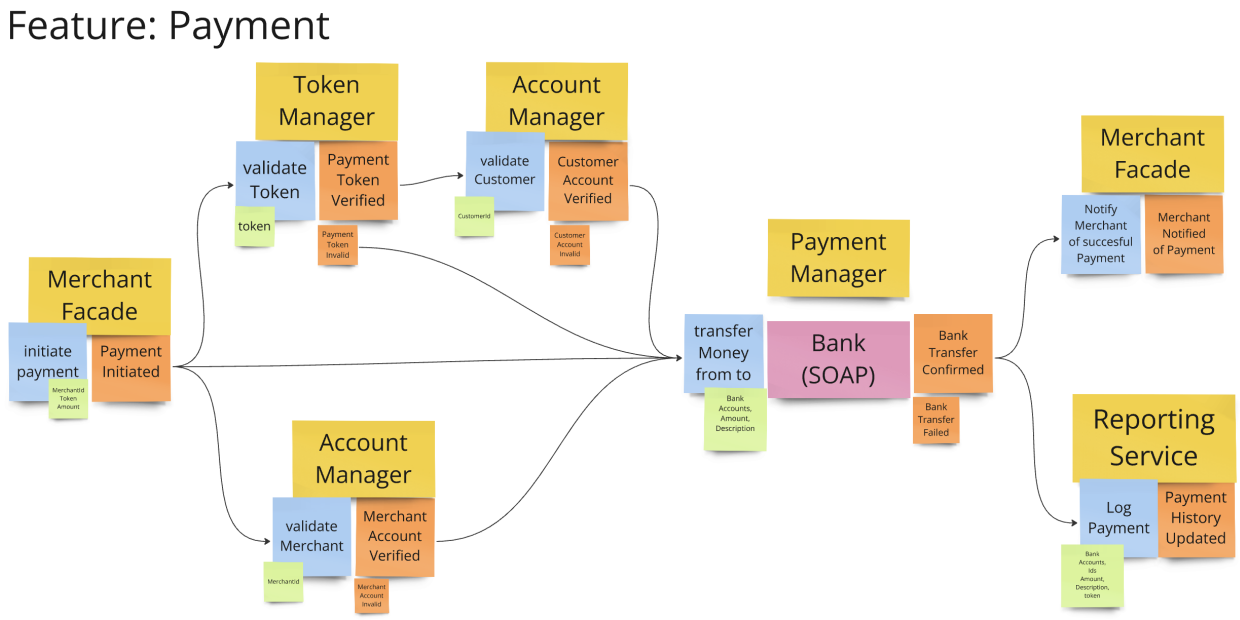


Figure 6: Event storming payment feature

3.4.1 Scenario Example

The feature is initiated through a `POST` request to the REST API and it contains the merchant ID, the customer's payment token and the amount to be transferred, wrapped inside a payment request record. The merchant facade then publishes an initial event for the payment request and assigns a correlator to it. This event is picked up by the token manager as well as the account manager. While the account manager verifies that merchant ID is known to DTUPay and retrieves the information stored about the merchant, the

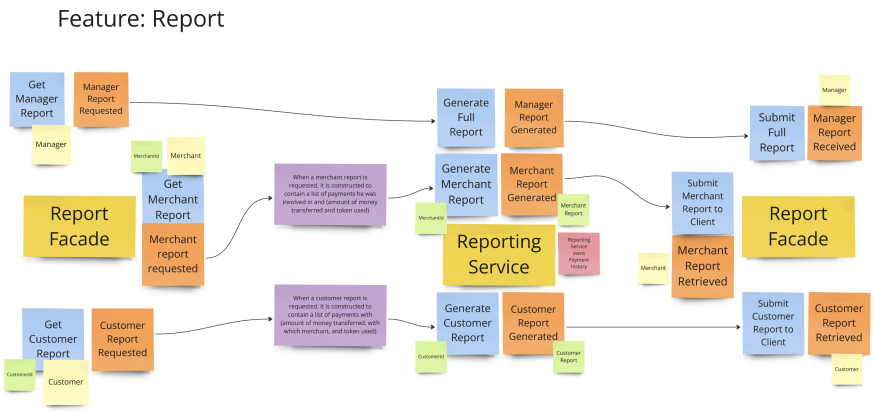


Figure 7: Event Storming result for the payment reporting process

token manager validates the payment token and retrieves the customer ID it belongs to. Now the account manager picks up the customer ID, verifies that it belongs to a registered DTUPay customer and retrieves the associated information. The payment manager awaits successful merchant and customer validation to then perform the payment request to the SOAP-based bank service. Finally it publishes a confirmation event containing a payment record record with all information needed for logging. This is picked up by the report service to log the payment and by the merchant facade to construct the REST API response. The response either has status code 200(`OK`) indicating success or status code 400(`BAD REQUEST`) indicating failure with an error message entity. For the payment flow, error handling for an unknown merchant, an invalid payment token and a failed bank SOAP request are implemented.

3.5 Report

When customer and merchant have been part of a payment process, information about transactions will be logged in the web-service. This information is retrievable by both the merchant and customer, in addition to a manager that have access to all information. The standard info a merchant gets are the amount of funds transferred and the token used. The customer has access to additional information about the registered merchant ID.

3.5.1 Scenario example

The feature is initiated through `GET` requests to the rest API. The `GET` request requires an ID for both customers and merchants but is not required for the manager, as they can be seen as the admin of the system. Three different interface methods have been implemented for the different users, by exposing appropriate resources on the `/reports` REST path prefix. Whenever, a resources is accessed by the HTTP request, the facade will call `getCustomerReport(String customerId)` that will publish a `"CustomerReportRequested"` event designated for the reporting microservice. After receiving this event, the reporting service will retrieve a list of information stored in an object `CustomerView`, containing the merchant id, the token used and amount used, and produce a Report object, that will be published with an `"CustomerReportGenerated"` event. Eventually, the latter will be caught by the reporting facade and generate a HTTP response containing the list of log objects. The response produces a status code of 200(`OK`), as if a user tries to retrieve a report before any payments have been done, they will simply be provided an empty list.

4 Architecture

Each of the developed services was designed with the goal of complying to the principles of hexagonal architecture. This architectural style emphasizes a clear separation between the core business logic, the external interfaces and data persistence adapters, promoting modularity, testability, and ease of maintenance as when one service fail, it would not break an entire monolith.

In this approach, the core domain logic is encapsulated within the application inner layers, while interactions with external systems (such as databases, APIs, or user interfaces) are handled through clearly defined ports and adapters. This ensures that the business logic remains independent of external frameworks or technologies, making it easier to replace or modify integrations without impacting the core functionality. Additionally, the hexagonal architecture aligns with best practices for modern microservice design, enabling better integration, improved test coverage, and clear separation of concerns.

4.1 Microservices

For the communication between microservices in the backend, we employed a **RabbitMQ** containerized message queue to which all units are connected. Doing so enabled us to perform asynchronous communication between services allowing access to multiple users. For the asynchronous request/response interactions, it was essential to employ correlation, such that we had an efficient means to distinguish different requests. As each microservice follows the Single Responsibility Principle, communication between services is required.

4.1.1 account-manager-service

The purpose of this microservice is to manage and store all registered accounts. The key features of the `account-manager-service` include handling the registration and deregistration of all users in the application. This ensures account validation, which is a necessary component of the payment scheme.

4.1.2 token-manager-service

This microservice is responsible for storing and generating unique payment tokens for `Customers`. Its key features include the secure generation of unique tokens, the validation of tokens to ensure they are legitimate and the ability to associate tokens with specific customers and transactions. It employs a bi-directional mapping repository that helps with the fast retrieval of tokens and customer IDs from storage.

4.1.3 payment-manager-service

This microservice is solely dedicated to the payment functionality. It listens for multiple events from the services depicted in the event storming diagram in Figure 6, ensuring accurate validation and context-aware error handling for the provided payment requests. Additionally, it interacts with the bank using the **SOAP** protocol.

4.1.4 facade-service

This microservice provides the functionality of DTU-Pay through a **REST API**, serving as the interface for interacting with the system. It is designed to expose the functionality of the DTU app for both `Customer` and `Merchant` end-users.

The microservice includes the `Customer`, `Merchant`, and `Report` facades, which encapsulate the underlying logic for managing customer- and merchant operations, and reporting functionalities, respectively. These facades ensure the clear separation of concerns, simplify API interactions, and simplifying the extension or modification of individual features without affecting others.

4.1.5 reporting-manager-service

This microservice is responsible for logging successful payments and generating query-specific reports based on the stored data. It facilitates queries for `Customer` and `Merchant` reports based on their account ID. The `Manager` report gives the complete log over successful payments. Finally, we have chosen to implement a simpler version of the reporting mechanism with in-memory storage of payment logs. We have also attempted to implement the *Event Sourcing & CQRS* technique in the `cqrs-reporting-service` module of the repository which failed however, and was left disconnected from the application.

4.2 Resources

For the RestAPI, we implemented 3 different facades, as described in the problem description. For the implemented REST API, the exposed resources and their supported operations can be seen in Table 1.

URI Path = Resource	HTTP Method	Function
/customers	`POST`	Register a customer
/customers/{id}	`DELETE`	Deregister customer
/customers/{id}/tokens	`POST`	Generate customer tokens
/merchants	`POST`	Register a merchant
/merchants/{id}	`DELETE`	Deregister merchant
/merchants/{id}/payments	`POST`	Initiates payment process
/reports/customers/{id}	`GET`	Retrieves the customer report
/reports/merchant/{id}	`GET`	Retrieves the merchant report
/reports/manager	`GET`	Retrieves the manager report

Table 1: Overview of implemented resources, their methods and functions

And interactive example can be found in the root folder of our repository in a Swagger/OpenAPI generated `YAML` file called `openapi.yaml`.

By default, the Web service provides three different root resources, `/customers`, `/merchants` and `/reports`. As described in the problem description, the webservice is meant to connect to different entities and while both `/customers` and `/merchants` provide similar functionalities, it would not make sense to merge them. Based on the business logic the facades should be split, as customers do not need access to `/merchants/id/payments`, and similarly merchants do not need access to `/customers/{id}/tokens`. The resources for reports, also makes a distinction between what kind of user that access it. However due to providing a similar functionality for all users, it can be stored within the same root resource `/reports`.

5 Common Patterns

During implementation of different microservices we encountered similar challenges, which we generally solved in a similar way to be consistent throughout the code.

5.1 Aggregates

Several feature event flows require multiple events to arrive for a specific request correlator in order to proceed with request. For example during payment, the `Payment Manager` awaits the events *Merchant Account Verified*, *Customer Account Verified* and *Bank Transfer Confirmed* to arrive for the same correlator, before performing the bank transfer. To collect the incoming events related to a correlator, we implemented custom aggregate classes. The aggregates are stored in a thread safe map data structure with their correlator as a key. For each new incoming event correlator, an aggregate is created and placed into the map. Once all required events for the correlator have arrived, the aggregate's `isComplete` method succeeds, the aggregate is removed and the event flow continues using the information from the aggregate.

5.2 Repository

Multiple microservices need to store data. To abstract the storage from the business logic, we created repository interfaces defining the storage functionality. This way the type of data storage can be changed later without needing to change the business logic. However, in all cases we have opted for in-memory storage implementations given the time constraints and complexity of the project.

5.3 Event Enum

The **EventTypes** enum is used to define a centralized, strongly-typed list of events in the system. This approach is utilized in the messaging architecture through the **RabbitMQ** communication channel, where producers publish events to a message queue, and the consumers subscribe to and process these events based on their type or more advanced topics. We have not used the topic functionality extensively, but we could notice a potential in splitting success/failure events by leaf-level path splits. Hence, we could reuse topic patterns to subscribe to a set of related events. Using an enum ensures that only valid event types can be used. This minimizes runtime errors due to invalid or wrongly typed event names as strings. This patterns guarantees that the system is easier to maintain, making the codebase more self-documented.

5.4 Exception

Error handling is needed for all features of DTUPay. Internally, errors are communicated using negative versions of the expected events. In the facade, a received rest API request is handled using a business logic method call returning a completable future. If a failure event for the request is received, the future is completed exceptionally using a custom domain exception with an appropriate error message. That exception is then caught in the adapter handling the rest API calls and triggers a failure REST API response (in the 400+ domain) to the request with the received error message as an entity. In the end-to-end test project, this failure HTTP response is again triggering a domain exception appropriate for the client, to be thrown with the error message.

6 DevOps

We have opted to structure our service-based application in a single **Git** repository with separate Java modules for each microservice. Each of the service modules contains a `Dockerfile` assembled to build the service image starting with the base `eclipse-temuring:23-alpine` and a fresh compilation of the service source code with required dependencies. Similarly, we have worked together on a single version control branch (`"main"`).

Having been provided with a virtual machine for building, testing and deploying the containerized application throughout the development, we have set up a self-managed instance of the automation server **Jenkins**. Initially, we configured a single freestyle project for **Continuous Deployment** that triggered the build script that built all service-images, deployed them in a composite network through `docker compose` and ran the end-to-end tests. With time, we noticed it was rather useful to distinguish which part of the system (service-level tests or end-to-end tests) was failing. Verifying the failure reason in the monolithic build required diving into the build logs. Instead of re-arranging the services in separate **Git** repositories or sub-modules, we have opted to declare multiple upstream Maven projects in **Jenkins** for each of the microservice test suites. This approach enabled us to block the downstream (main end-to-end) project from deployment whenever one of the composed services updated to a failed build. Hence, the Andon board containing the 6 projects as in Figure 8 helped us identify the failing service early and optimize the use of resources on inevitably failed builds. Similarly, each of the Jenkins projects presents a separate test result graph on which the development progress can be tracked.

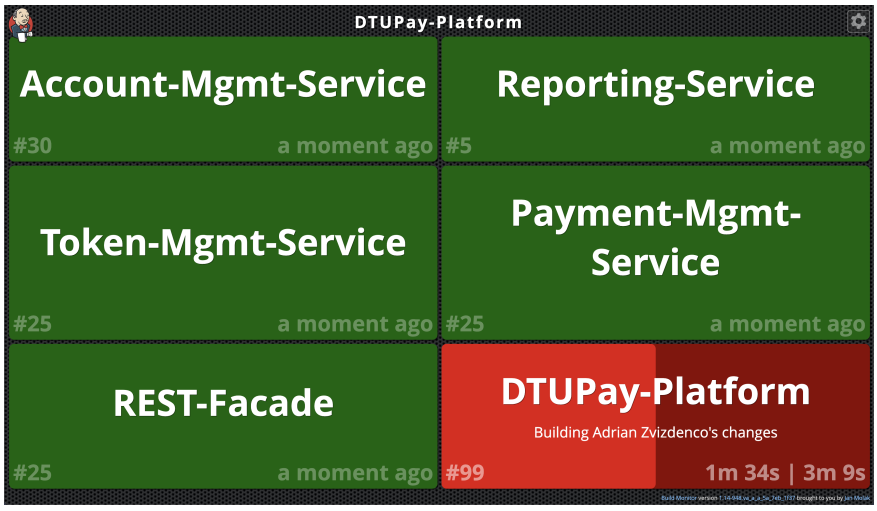


Figure 8: Andon board of service components and main downstream project

As part of the **DevOps** workflow, we have configured a GitHub web-hook that triggers Jenkins builds whenever some update is pushed to the repository. Having configured the microservice project using the *Sparse Checkout* feature, all the service projects would be rebuild even if updates concerned only a certain service's source code. The webhook activates all the upstream builds, however the main project (*DTUPay-Platform*) has been configured to block while upstream builds are being executed, hence waiting for the latest updates in the services. All the **CI/CD** executions generate test reports shown on the platform and

also on GitHub through status check updates; one can notice a checkmark or cross with up to 6 checks along with the commits on the main repository page. The main responsible for the Jenkins instance and project configuration was Adrian Zvizdenco, however the other team members have contributed to it as well.

RabbitMQ Compose Service Note: Even though the `docker-compose` mechanism employs a custom health check for the RabbitMQ container, its booting is completely failing occasionally (as seen in our Jenkins build history). To verify the application testing and deployment, one should try restarting the compose deployment until the RabbitMQ service is marked as healthy.

7 Workflow & Contributions

After the event sourcing was completed we started developing using **MOB** programming to guarantee mutual understanding and agreement on the early design decisions. While doing so we followed a test-driven approach. For each feature, we would start by writing a failing behaviour (end-to-end) test. We would then proceed by writing a failing service test on the first service encountered during the features event flow. Next, we would implement the functionality in the service to make the service level test pass. Once the service level test passed, we would refactor the service and then execute the behaviour test again to confirm the same functionality is kept. We would then iterate over all services required for the features event flow in a similar fashion until the initial end-to-end behaviour test succeeded. Later, once the core functionalities were covered, we opted to work in smaller groups on error handling and other details but still continued to follow a test-driven approach.

Throughout the project, the highest priority was always given to not having any failing test cases in the **CI/CD** pipeline. We normally had an extra screen to show the Andon board and if there was a failing build, we would all focus on fixing it. Our commitment to test-driven development can be observed in our Jenkins test result trend depicted in Figure 9, where the number of successful test cases steadily increased with each commit. Finally, we have tested the code coverage of all Cucumber tests and implicitly Gherkin scenarios, achieving a cumulative 96% of source code lines covered (including the test suites definitions).

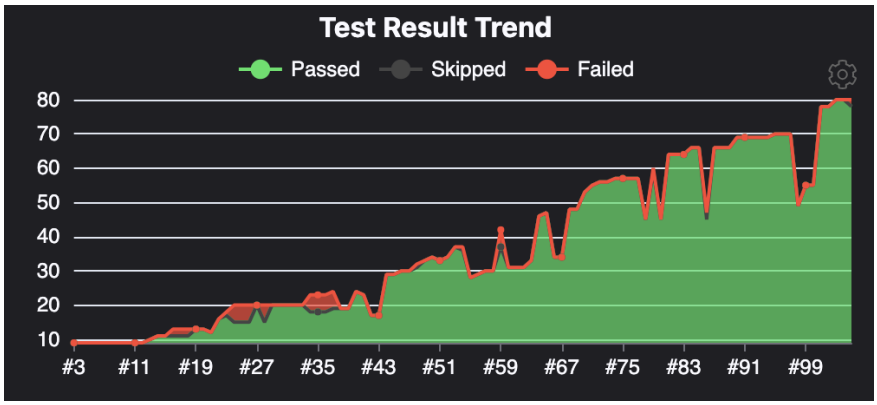


Figure 9: Test result trend on jenkins

NOTE: All team members have contributed to the entire development of the platform: writing feature files, step definitions, implementation and refactoring, and finally containerization. We have all contributed to every aspect of the project, ensuring that while some team members may have had expertise in certain areas, no one was left out of the process. In the repository, you will find annotations indicating who implemented specific parts. However, please interpret these annotations with caution, as methods have been revisited and refined multiple times to accommodate new functionality required by evolving tests, resulting in contributions from multiple authors/drivers.