Project Description: DTU Pay

Web Development and Web Services 02267

Group 4

Sanjeev Acharya s240090

Yvonne Lamisi Bukari s250123

Alexandru-Daniel Florea s250246

Sigurd Valentin Flach de Neergaard s215149

Nichita Railean s243772

Aleksander Kirsebom Salling Sønder s185289

January 2026

Contents

1 Introduction

Author: Aleksander Sønder (Translated from Danish with LLM)

This report describes the implementation of *DTU Pay* a service-oriented payment system extending the *Simple DTU Pay* assignment specification.

Through collaborative **Event Storming** sessions (both in-person and virtual), the team identified domain events, commands, aggregates, and bounded contexts. These artifacts guided the refactoring into microservices and defined clear service responsibilities and boundaries. This domain-driven approach enabled a structured evolution from the simple monolithic program to a fully functional, distributed service-oriented architecture.

Each microservice follows **hexagonal architecture** (Ports and Adapters pattern) to strictly isolate the business/domain logic from infrastructure concerns (asynchronous messaging, in-memory persistence, external integrations).

Inter-service communication is handled asynchronously via **RabbitMQ** (using message queues), providing decoupling, resilience, and traceability (e.g., correlation IDs).

The system is thoroughly tested using **Cucumber** for end-to-end behavior-driven scenarios (covering successful payments, rejections due to insufficient funds, invalid/used tokens, unknown parties, etc.) as well as unit tests.

The report details the architecture, justifies major design decisions (e.g., asynchronous messaging over synchronous calls, hexagonal isolation, path-based routing for access points), presents Event Storming artifacts and UML class/sequence diagrams, and includes selected Cucumber scenario excerpts to demonstrate how core features are realized within the architecture.

1.1 How to run the program

Clone the project from public Github:
https://github.com/02267group4/simple-dtu-pay
Read InstallationGuide.pdf (./InstallationGuide.pdf) in the root of the project for the complete step-by-step instructions.

1.1.1 Swagger and OpenAPI

All services are using Quarkus default dev-mode endpoints pattern:

- Swagger UI: `http://localhost:PORT/q/swagger-ui/`
- OpenAPI document (JSON): `http://localhost:PORT/q/openapi`
- OpenAPI document (YAML): `http://localhost:PORT/q/openapi?format=yaml`

Customer Service	Port: 8081 Swagger UI: `http://localhost:8081/q/swagger-ui/` OpenAPI: `http://localhost:8081/q/openapi` YAML: `http://localhost:8081/q/openapi?format=yaml` Location: `customer_service/src/main/resources/application.properties`
Merchant Service	Port: 8082

	Swagger UI: `http://localhost:8082/q/swagger-ui/` OpenAPI: `http://localhost:8082/q/openapi` YAML: `http://localhost:8082/q/openapi?format=yaml` Location: `merchant_service/src/main/resources/application.properties`
Payment Service	Port: 8083 Swagger UI: `http://localhost:8083/q/swagger-ui/` OpenAPI: `http://localhost:8083/q/openapi` YAML: `http://localhost:8083/q/openapi?format=yaml` Location: `payment_service/src/main/resources/application.properties`
Token Service	Port: 8084 Swagger UI: `http://localhost:8084/q/swagger-ui/` OpenAPI: `http://localhost:8084/q/openapi` YAML: `http://localhost:8084/q/openapi?format=yaml` Location: `token_service/src/main/resources/application.properties`
Manager Service	Port: 8085 Swagger UI: `http://localhost:8085/q/swagger-ui/` OpenAPI: `http://localhost:8085/q/openapi` YAML: `http://localhost:8085/q/openapi?format=yaml` Location: `manager_service/src/main/resources/application.properties`

2 List of team members

See title page

3 Contributions

Author: Aleksander Soender

The group has succeeded in developing a larger, fully functional service-oriented application based on the assignment "Simple DTU Pay". The group met in person on campus during the first three days, where the structure for our collaboration was established. The team also discussed which main tasks the project contained. After that, the work switched to virtual collaboration. Below is an individual description of how each person's work on the project has contributed to achieving the course learning objectives for each team member.

3.1 Teamwork

The project group consists of members with diverse backgrounds and educational directions, which has presented both challenges and strengths in our collaboration. We have experimented with several collaboration formats: Mob programming was used more in the early start, but pair programming proved to be the most effective way to tackle complex problems together. Smaller tasks and standalone problem-solving have mainly been handled individually.

Joint decisions, Event Storming, and overarching issues have consistently been addressed during group meetings, ensuring shared understanding and direction. Version control with Git has been used intensively and worked very well: Refactoring to hexagonal architecture (with clean separation of core domain from infrastructure concerns), separated POM structure, and feature-based development. All significant changes have been developed on individual branches and merged into main after joint review.

We have also incorporated message queues using RabbitMQ to handle asynchronous communication and event-driven flows, integrated as adapters in the hexagonal architecture. This has improved decoupling and supported reliable inter-component messaging. Every group member has independently developed and deployed small programs via Jenkins, which provided a solid, hands-on understanding of CI/CD principles. This has strongly contributed to keeping the main application functional and stable throughout the entire project.

3.2 Individual Contributions

3.2.1 Sanjeev Acharya, s240090

Author: Sanjeev Acharya

LO 1: Work in a team and to build a larger service-oriented application
Worked with team members by working on a dedicated feature branch and following existing test conventions established by teammates. Integrated test contributions with the shared test infrastructure.

LO 2: Create services based on their description using agile methods
Applied agile iterative development by following a systematic test workflow: create test run test validate fix issues move to next test. Started with simple test scenarios and progressively added more complex cases.

LO 3: Use existing services according to their description; compose new services from existing ones
Created integration tests for the Manager Service that validate its report functionality, which retrieves and aggregates payment data from across the system.

LO 4: Develop, test, and document a larger service-oriented application in a team using agile practices
Created Cucumber BDD test scenarios for the manager service covering: basic report retrieval, payment status validation (COMPLETED, FAILED, PENDING), data integrity checks, and concurrent access patterns. Documented the system architecture through UML diagrams.

LO 5: Build, deploy, and run a service-oriented application
Ran the Docker Compose environment with all microservices to execute tests. Configured test execution using Maven with proper environment variables.

LO 6: Discuss coordination and the security of Web services
Tested the async request-reply coordination pattern using RabbitMQ with correlation IDs. Validated security aspects including API key authentication and token-based payment authorization with rejection of reused tokens.

LO 7: Discuss service-oriented architectures

Created UML diagrams documenting the microservices architecture, illustrating service boundaries, communication patterns, and data flow between services.

3.2.2 Yvonne Lamisi Bukari, s250123

Author: Yvonne Lamisi Bukari

LO 1: Work in a team and to build a larger service-oriented application
Yvonne was an active member of the team and participated consistently in virtual meetings throughout the project. I contributed during group discussions and event storming and participated in the mob programming on service boundaries, integration challenges, and the overall architecture of the system. My involvement helped align the understanding of message flow between merchant-service and payment-service, supporting the teams overall coordination.

LO 2: Create services based on their description using agile methods
The merchant reporting feature was developed based on acceptance criteria and discussions from the teams event-storming sessions. I contributed to refactoring the merchant service, implemented the merchant-side logic for generating a merchant payment report, which involved exposing the merchantsid reports endpoint and connecting it to an asynchronous messaging workflow, and translated the intended behaviour into concrete code by adding correlation-ID handling, JSON request structures, and asynchronous response handling. This work required multiple incremental iterations and testing cycles, following the teams agile workflow.

LO 3: Use existing services according to their description; compose new services from existing ones
My work centered on integrating the merchant-service with the existing payment-service through RabbitMQ. I used the payment-services documented behaviour to design the message formats for report requests and responses, ensuring both services could communicate reliably. In addition, I configured the outgoing and incoming messaging channels, created the necessary exchanges and routing keys, and connected them to the existing domain logic. This implementation allowed the merchant-service to compose functionality from the payment-service without creating direct synchronous coupling.

LO 4: Develop, test, and document a larger service-oriented application in a team using agile practices
While implementing the merchant reporting flow, I spent a lot of time testing how the merchant-service and payment-service communicated in practice. I ran both services in Quarkus dev mode and used RabbitMQ logs to confirm that messages were sent and received correctly. Along the way, I debugged issues such as incorrect routing keys, missing queues, and port conflicts during service startup. Our team also relied on BDD acceptance tests written in Cucumber to verify end-to-end behaviour, and my updates to the reporting feature helped keep these tests consistent with how the services now interacted.

LO 5: Build, deploy, and run a service-oriented application
I ran the merchant- and payment-services locally using Quarkus, configured RabbitMQ manually, and resolved runtime issues such as port conflicts, misconfigured queues, and connection failures, after which the entire end-to-end workflow was tested using curl and application logs to verify that messages were correctly published and consumed by the system and that the messaging

infrastructure behaved as expected.

LO 6: Discuss coordination and the security of Web services
The merchant reporting feature required careful coordination between distributed services. I contributed to understanding and implementing correlation IDs to match asynchronous requests and responses, ensuring reliable communication between the merchant-service and payment-service. I also participated in team discussions about message integrity, error handling, and timeouts, as well as all important aspects of secure and reliable service coordination.

LO 7: Discuss service-oriented architectures
Through this project, I understand key SOA principles such as decoupling services through messaging, designing domain-driven request/response structures, and the hexagonal architecture within the merchant-service, explain how RabbitMQ supports asynchronous communication and why exchanges, queues, and routing keys were separated for merchant and manager reporting.

3.2.3 Alexandru-Daniel Florea, s250246

Author: Alexandru-Daniel Florea

LO 1: Work in a team and to build a larger service-oriented application
Alex Participated in group meetings both virtual and at DTU, joined the event storming sessions, and contributed to the discussed topics like architecture decisions, service boundaries and implementation strategy. Collaborated with the team members, developed functionalities on different branches throughout the project.

LO 2: Create services based on their description using agile methods
As a group we created the project's base structure allowing us to individually implement services. Alex developed the customer service and implemented the customer report feature, following the project requirements and the more concrete design from the event storming boards.

LO 3: Use existing services according to their description; compose new services from existing ones
The customer report feature it was necessary to integrate the customer-service with the payment-service, through RabbitMQ. Alex implementing publishers and consumers on both services, created events and messaging components, allowing the customer-service to request payment data asynchronously.

LO 4: Develop, test, and document a larger service-oriented application in a team using agile practices
The group created tests for the general and main functionalities of the project. Alex wrote Cucumber tests (features and steps) for the customer report, verifying the behavior of the functionality. He also contributed to API documentation by creating Swagger specifications for the project's REST endpoints, ensuring the services were properly documented for team members and future developers.

LO 5: Build, deploy, and run a service-oriented application
Alex worked with the project's build and deployment infrastructure. Running the full system, using Maven for building and Docker Compose for orchestration. Troubleshooted configuration issues related to service dependencies and environment variables. Also contributed to Jenkins configuration, helping ensure automated builds and tests ran reliably.

LO 6: Discuss coordination and the security of Web services
The customer report feature implements the async request-reply coordination pattern using RabbitMQ with correlation IDs. Alex ensured that each report request receives a unique identifier, with the customer-service storing a CompletableFuture and waiting for the payment-service to respond with matching correlation ID. This pattern provides reliable coordination between distributed services while handling timeout scenarios gracefully.

LO 7: Discuss service-oriented architectures
Through customer report feature, Alex gained practical knowledge of SOA principles, service decoupling, asynchronous messaging, hexagonal architecture for separating domain logic from infrastructure concerns, and the use of DTOs for cross-service communication. Understands how RabbitMQ exchanges, queues, and routing keys enable loosely coupled service interactions.

3.2.4 Sigurd Valentin Flach de Neergaard, s215149

Author: Sigurd Valentin Flach de Neergaard

LO 1: Work in a team and to build a larger service-oriented application
Sigurd participated in both physical and virtual meetings and took part in mob programming and pair programming sessions throughout the project. He contributed to discussions around domain understanding and service boundaries, including workshops such as event storming. Sigurd was involved in refactoring parts of the system towards a hexagonal (ports-and-adapters) architecture, supporting clearer separation of concerns and easier collaboration within the team. In addition, Sigurd coordinated the integration of the teams work by managing most merges and ensuring that contributions from different team members worked correctly together in the shared codebase.

LO 2: Create services based on their description using agile methods
The group implemented services based on event storming results and acceptance criteria defined as Cucumber scenarios. Sigurds main contribution was working on the Token service, where he translated service descriptions and acceptance criteria into concrete endpoints and domain logic. He also contributed to the initial setup of payment-related tests.

LO 3: Use existing services according to their description; compose new services from existing ones
Sigurd worked with integrating the Token service with other services through well-defined contracts, including REST endpoints and message payloads. He helped resolve integration issues related to identifiers, message flow, and service communication, ensuring that services interacted correctly according to their documented behavior.

LO 4: Develop, test, and document a larger service-oriented application in a team using agile practices
The team used BDD acceptance tests with Cucumber to verify end-to-end behavior. Sigurd focused primarily on implementing and maintaining tests related to the Token service, ensuring correct token creation, validation, and failure scenarios. He also contributed to initial unit and Cucumber tests for the payment flow to validate core logic and expected error handling.

LO 5: Build, deploy, and run a service-oriented application
Sigurd contributed to containerizing services using Docker and running the full system using

Docker Compose. This included configuring service dependencies and environment variables, as well as troubleshooting issues related to container networking to ensure that the system could be started and tested reliably.

LO 6: Discuss coordination and the security of Web services
The project used asynchronous messaging and correlation identifiers to coordinate distributed workflows between services. Sigurd participated in discussions around these mechanisms. Security aspects were only briefly discussed within the group and not fully explored in the implementation.

LO 7: Discuss service-oriented architectures
Sigurd contributed to architectural discussions and can explain key decisions such as service decomposition, the use of messaging for decoupling, and the application of hexagonal architecture to keep domain logic independent of infrastructure concerns.

3.2.5 Nichita Railean, s243772

Author: Nichita Railean

LO 1: Work in a team and to build a larger service-oriented application
Nichita was responsible for a lot of the refactoring work in this project. He moved the payment module to hexagonal architecture early on, which set the pattern for other services. Later he extracted the manager functionality into its own microservice and made sure it worked with the rest of the system. Nichita also did a lot of work merging branches and fixing conflicts when different team members' code needed to come together. This included multiple merge commits where he had to carefully keep both new features and existing functionality working.

LO 2: Create services based on their description using agile methods
Nichita implemented the manager report feature from scratch based on the acceptance criteria. This included creating the REST endpoint, the RabbitMQ messaging flow, and the DTOs needed to pass payment data between services. He also helped fix and refactor the merchant report feature to follow the same pattern. The work was done in small commits and tested incrementally.

LO 3: Use existing services according to their description; compose new services from existing ones
The manager service Nichita built composes with the payment service through RabbitMQ. When a manager requests a report, the manager service sends a request message with a correlation ID, and the payment service responds with the payment data. Nichita implemented both the publisher and consumer sides of this flow. He also fixed the integration between token service and payment service after refactoring.

LO 4: Develop, test, and document a larger service-oriented application in a team using agile practices
Nichita wrote the Cucumber feature file for manager reports and the corresponding step definitions. After refactoring the codebase, he spent a lot of time fixing broken tests and making sure all the end-to-end flows still passed. He also updated the SimpleDTUPay client to work with the new service structure and fixed DTO mappings that broke during the refactoring.

LO 5: Build, deploy, and run a service-oriented application
Nichita set up and maintained most of the DevOps infrastructure. He created the initial Jenkinsfile that builds all services, starts them with Docker Compose, runs the Cucumber tests,

and cleans up. He wrote the multi-stage Dockerfile that builds any service using build arguments. He configured docker-compose.yml with health checks so RabbitMQ starts before the other services. Nichita also added Maven wrapper files where they were missing and fixed various configuration issues like port mappings and environment variables.

LO 6: Discuss coordination and the security of Web services
Nichita implemented the async request/reply pattern with correlation IDs for the manager report flow. Each request gets a unique ID, the service stores a CompletableFuture, sends the request to RabbitMQ, and waits up to 30 seconds for a reply with the same ID. This handles coordination safely and avoids blocking forever if something goes wrong. Security was not a focus in this project but Nichita can discuss how tokens provide some anonymity for customers during payments.

3.2.6 Aleksander Sønder, s185289

Author: Aleksander Sønder

LO 1: Work in a team and to build a larger service-oriented application
Aleksander participated in all physical and virtual meetings and actively contributed to frequent mob programming and pair programming sessions. He helped facilitate online workshops (e.g., event storming) to align on domain events and service boundaries. Aleksander also drove parts of a larger refactoring effort where we migrated core logic towards a hexagonal (ports-and-adapters) architecture, improving separation of concerns and enabling clearer collaboration across the team.

LO 2: Create services based on their description using agile methods
The group implemented services from (re-visiting) the event storming session and acceptance criteria expressed as Cucumber scenarios. Aleksander contributed by translating service descriptions into concrete endpoints.

LO 3: Use existing services according to their description; compose new services from existing ones
Aleksander helped ensure services used each other through stable contracts (URLs, payload formats, routing keys) and resolved integration issues when assumptions about identifiers and container networking caused runtime failures.

LO 4: Develop, test, and document a larger service-oriented application in a team using agile practices
The team used BDD acceptance tests (Cucumber) for end-to-end behavior, and Aleksander additionally implemented unit tests for the payment functionality to validate core business logic in isolation (e.g., state transitions and failure cases). This made change breaks easier to catch early. Also contributed with Cucumber tests for fail payment scenarios.

LO 5: Build, deploy, and run a service-oriented application
Aleksander worked with containerizing services (Dockerfiles) and orchestrating the system with Docker Compose, including infrastructure dependencies (RabbitMQ) and port mappings. Also helped troubleshoot runtime configuration issues (container networking and environment variables) to ensure the system could be built, started, and tested end-to-end reliably.

LO 6: Discuss coordination and the security of Web services
The team coordinated distributed workflows using asynchronous messaging and correlation

identifiers (request IDs) for processes. This is probably where the project is halting a bit as we haven't discussed security much.

LO 7: Discuss service-oriented architectures
Aleksander contributed to and can explain architectural decisions such as decomposing responsibilities into services, using messaging to decouple workflows, and applying hexagonal architecture to keep domain logic independent from infrastructure.

4 Result of the Event Storming Process

The Event Storming process was conducted collaboratively online using the Miro platform. This allowed real-time sticky-note collaboration, clustering, and refinement of domain events, commands, aggregates, and bounded contexts.

The team revisited the Miro boards several times during the process and changed things which is also key in agile development.

Participants: Aleksander Sønder, Yvonne Lamisi Bukari, Sigurd Valentin Flach de Neergaard.

The figures below show the final state of each major bounded context / core process after consolidation and cleanup. Colors follow this convention:

- Orange = Domain Events
- Blue = Commands
- Green = Aggregates
- Purple = Policies / Read Models
- Yellow = External Systems / Actors

4.1 Registering customer, merchant

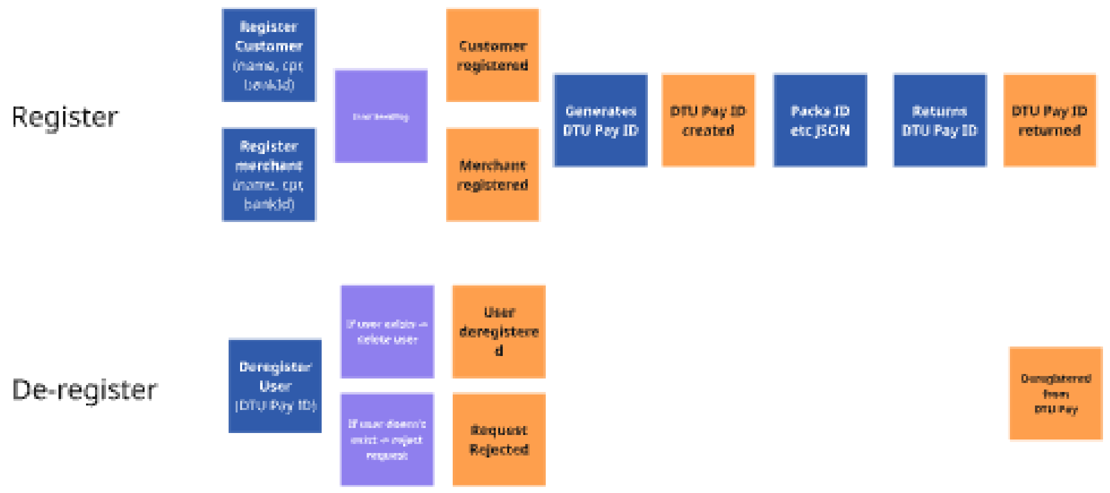

Register
Register Customer (name, cpr, bankId)
Register merchant (name, cpr, bankId)
Customer registered
Merchant registered
Generates DTU Pay ID
DTU Pay ID created
Packs ID etc JSON
Returns DTU Pay ID
DTU Pay ID returned
De-register
Deregister User (DTU Pay ID)
If user exists -> delete user
If user doesn't exist -> reject request
User deregistered
Request Rejected
Deregistered from DTU Pay

Figure 1: Event Storming Registering customer and merchant (account creation)

4.2 Processing payment

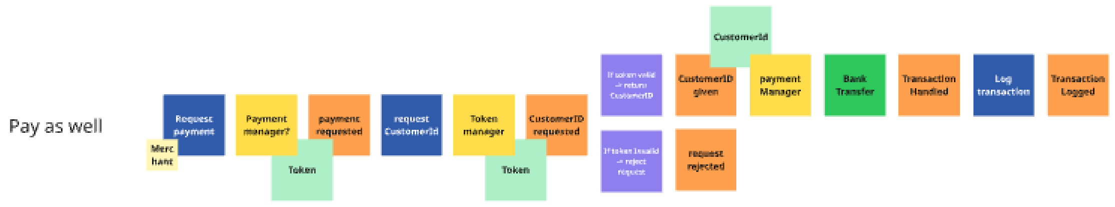


Figure 2: Event Storming Core payment flow (happy path + rejection cases)

4.3 Managing tokens

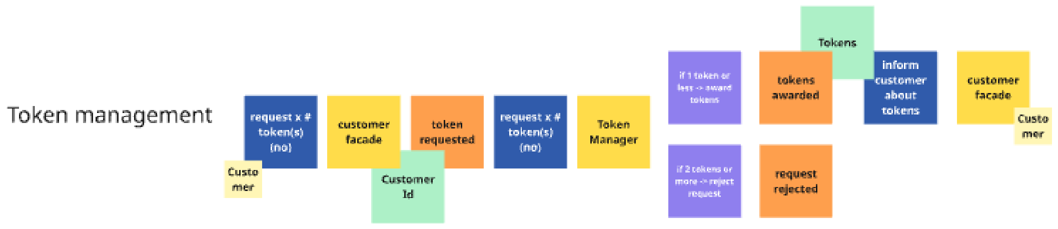

Token management
request x # token(s) (no)
Customer
customer facade
token requested
Customer Id
request x # token(s) (no)
Token Manager
if 1 token or less -> award tokens
if 2 tokens or more -> reject request
Tokens
tokens awarded
request rejected
inform customer about tokens
customer facade
Customer

Figure 3: Event Storming Token lifecycle (request, issuance, consumption, expiration)

4.4 Manager reporting

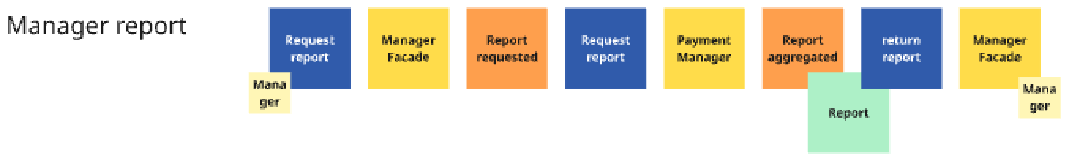


Figure 4: Event Storming Manager / DTU Pay internal reporting (global overview)

4.5 Merchant reporting

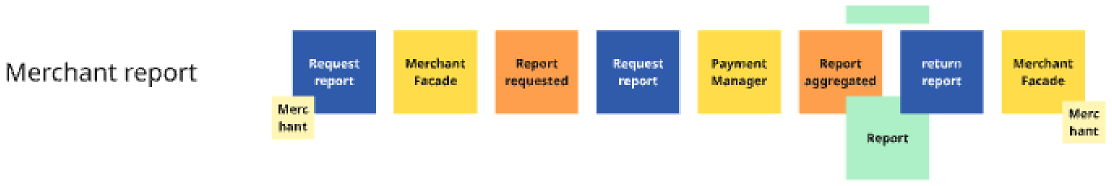


Figure 5: Event Storming Merchant-specific reporting (payments received, anonymized)

4.6 Customer reporting

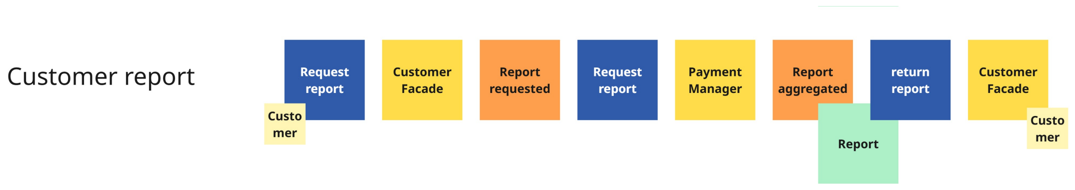


Figure 6: Event Storming Customer-specific reporting (personal payment history)

5 Application Architecture

@Author: Aleksander Soender s185289

Initial app -> layered app

5.1 From layered to hexagonal architecture

Converting to hexagonial

Why?

Testability. When modules are tied to other modules it's actually very difficult to test specific module in isolation. With hexagonial module design

We split the system into small services: customer, merchant, token, payment, and manager. Each service has a clear boundary and its own REST API.

Most synchronous flows use REST, for example register and delete. Asynchronous flows use RabbitMQ and a request/reply pattern with correlation ids. This is used for token issuing, token validation results, and manager/merchant reports.

The payment service owns payment state and talks to the bank service. Customer and merchant services only store basic profile data and IDs.

5.2 Service Overview

The DTU Pay system is built as a set of microservices, each running in its own Docker container. The services communicate through REST APIs for simple operations and RabbitMQ for async flows. Below is a table showing each service, its port, and what it does:

Service	**Port**	**Purpose**
customer_service	8081	Customer registration, token requests, customer reports
merchant_service	8082	Merchant registration, merchant reports
payment_service	8083	Payment processing, bank transfers, central report hub
token_service	8084	Token creation, validation, and consumption
manager_service	8085	Manager-level payment reports
RabbitMQ	5672	Message broker for async communication

Table 1: Services and their ports

5.3 Hexagonal Architecture in Each Service

Each service follows the hexagonal (ports and adapters) pattern. The folder structure looks like this:

`domain/model/` – Domain entities like Customer, Token, Payment

`domain/service/` – Business logic (e.g. TokenService, PaymentService)

`application/port/out/` – Repository interfaces (output ports)

`adapter/in/rest/` – REST controllers (driving adapters)

`adapter/in/messaging/` – RabbitMQ consumers (driving adapters)

`adapter/out/persistence/` – In-memory repositories

`adapter/out/messaging/` – RabbitMQ publishers (driven adapters)

This setup keeps the domain logic clean and testable. The domain layer does not depend on infrastructure code. Instead, it uses port interfaces that adapters implement.

5.4 Communication Patterns

The system uses two main communication patterns:

5.4.1 REST for simple operations

Registration and deletion of customers and merchants use direct REST calls. For example, `POST /customers` creates a new customer and returns immediately.

5.4.2 RabbitMQ for async flows

More complex flows use async messaging with a request/reply pattern. All services connect to a single RabbitMQ topic exchange called `dtu.pay`. Each message type has a routing key like `token.issue.request` or `payment.requested`.

When a service needs something from another service:

1. It generates a correlation ID
2. Stores a CompletableFuture in a local store
3. Publishes the request with the correlation ID
4. The other service processes it and publishes a reply with the same ID
5. The first service matches the ID and completes the future

This pattern is used for:

Token issuance: customer_service → token_service → customer_service

Token validation during payment: payment_service → token_service → payment_service

All report types: manager/merchant/customer → payment_service → requester

5.5 Infrastructure

The system runs with Docker Compose. Each service has its own Dockerfile and the `docker-compose.yml` sets up the network, environment variables, and health checks. RabbitMQ starts first and the other services wait for it to be healthy before starting.

Jenkins is used for CI/CD. The Jenkinsfile builds all services, starts them with Docker Compose, waits for them to be ready, runs the Cucumber tests, and then tears everything down.

6 UML Class Diagram

Author: Sanjeev Acharya

The system is split into five microservices. Each service follows the hexagonal architecture pattern with similar package structure. Below we describe the main classes in each service.

6.1 Customer Service

`CustomerResource` - REST controller for customer registration and token requests

`RequestResource` - REST endpoint for polling async request results

`CustomerService` - Domain service with business logic for managing customers

`Customer` - Domain entity representing a customer

`CustomerRepositoryPort` - Interface for customer persistence

`InMemoryCustomerRepository` - In-memory implementation of CustomerRepositoryPort

`RequestStore` - Tracks async request state for polling

`RabbitMQTokenIssueRequestPublisher` - Publishes token issue requests

`RabbitMQTokenListRequestPublisher` - Publishes token list requests

`RabbitMQTokenIssueResultConsumer` - Receives token issue results

`RabbitMQTokenListResultConsumer` - Receives token list results

`MessagingStartup` - Initializes message consumers on startup

6.2 Merchant Service

`MerchantResource` - REST controller for merchant registration and reports

`MerchantService` - Domain service for merchant business logic

`Merchant` - Domain entity with id, name, CPR, and bank account

`MerchantRepositoryPort` - Interface for merchant persistence

`MerchantRepository` - In-memory implementation of MerchantRepositoryPort

`MerchantReportStore` - Stores pending report requests with CompletableFuture

`RabbitMQMerchantReportPublisher` - Publishes report requests to payment service

`RabbitMQMerchantReportResponseConsumer` - Receives report responses

`PaymentDTO` - Data transfer object for payment info

`MessagingStartup` - Initializes message consumers on startup

6.3 Token Service

`TokenResource` - REST endpoint for direct token creation (outdated, see 7.3)

`TokenService` - Domain service that issues, lists, and validates tokens

`Token` - Domain entity representing a single-use token

`TokenInfo` - Value object with customer and bank account info

`TokenRequest` - DTO for REST token creation

`TokenRepositoryPort` - Interface for token storage

`InMemoryTokenRepository` - In-memory implementation

`RabbitMQTokenIssueRequestConsumer` - Listens for token issue requests

`RabbitMQTokenListRequestConsumer` - Listens for token list requests

`RabbitMQPaymentRequestedConsumer` - Validates tokens during payments

`RabbitMQTokenIssueResultPublisher` - Sends token issue results

`RabbitMQTokenListResultPublisher` - Sends token list results

`RabbitMQTokenValidationResultPublisher` - Sends validation results

`MessagingStartup` - Initializes message consumers on startup

6.4 Payment Service

`PaymentResource` - REST controller for payment requests

`PaymentService` - Domain service with payment processing logic

`Payment` - Domain entity tracking payment state

`PaymentRequest` - DTO with payment details

`PaymentRepositoryPort` - Interface for payment persistence

`InMemoryPaymentRepository` - In-memory implementation

`BankProducer` - CDI producer that creates BankService SOAP client

`RabbitMQPaymentRequestedPublisher` - Publishes payment requests to token service

`RabbitMQTokenValidationResultConsumer` - Receives token validation results

`RabbitMQManagerReportRequestConsumer` - Handles manager report requests

`RabbitMQMerchantReportRequestConsumer` - Handles merchant report requests

`RabbitMQManagerReportResponsePublisher` - Sends manager report responses

`RabbitMQMerchantReportResponsePublisher` - Sends merchant report responses

`MessagingStartup` - Initializes message consumers on startup

6.5 Manager Service

`ManagerResource` - REST endpoint for manager report requests

`ManagerReportStore` - Stores pending requests with CompletableFuture

`RabbitMQManagerReportPublisher` - Sends report requests to payment service

`RabbitMQManagerReportResponseConsumer` - Receives report responses

`PaymentDTO` - Data transfer object for payment info

`ManagerReportEvents` - Event DTOs for request/response

`MessagingStartup` - Initializes message consumers on startup

6.6 Common Patterns

All services follow the same hexagonal structure:

`adapter/in/rest/` - Driving adapters (REST controllers)

`adapter/in/messaging/` - Driving adapters (RabbitMQ consumers)

`adapter/out/persistence/` - Driven adapters (in-memory repositories)

`adapter/out/messaging/` - Driven adapters (RabbitMQ publishers)

`adapter/out/request/` - Request stores for async correlation

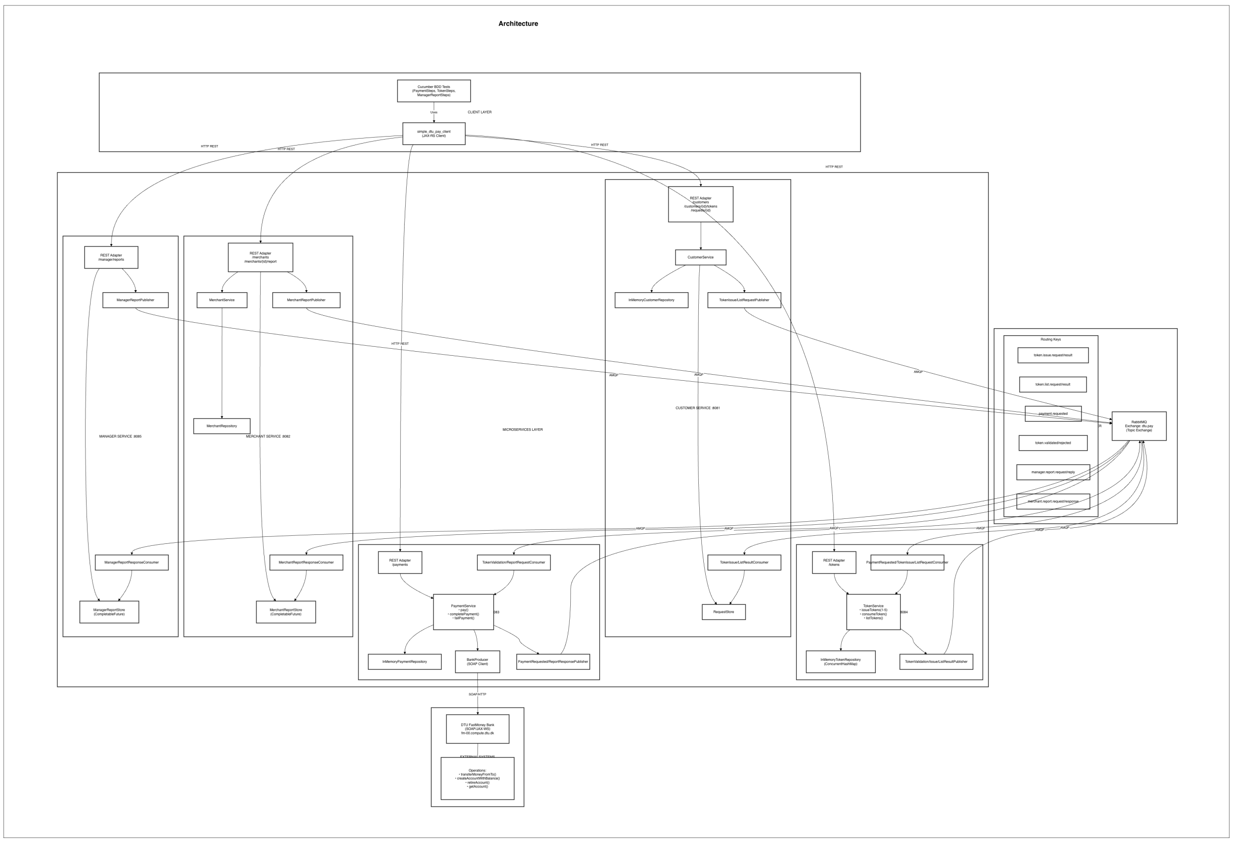
`application/port/out/` - Output ports (repository interfaces)

`domain/model/` - Domain entities and value objects `domain/service/` - Domain services with business logic

`MessagingStartup` - Lifecycle initialization for consumers

6.7 UML Diagrams

Overall Architecture



Architecture
Cucumber BDD Tests (PaymentSteps, TokenSteps, ManagerReportSteps)
Uses
CLIENT LAYER
simple_dtu_pay_client (JAX-RS Client)
HTTP REST
HTTP REST
HTTP REST
HTTP REST
REST Adapter /manager/reports
ManagerReportPublisher
MANAGER SERVICE :8085
ManagerReportResponseConsumer
ManagerReportStore (CompletableFuture)
REST Adapter /merchants /merchants/{id}/report
MerchantService
MerchantReportPublisher
MerchantRepository
MERCHANT SERVICE :8082
MerchantReportResponseConsumer
MerchantReportStore (CompletableFuture)
MICROSERVICES LAYER
HTTP REST
REST Adapter /customers /customers/{id}/tokens /requests/{id}
CustomerService
InMemoryCustomerRepository
TokenIssue/ListRequestPublisher
CUSTOMER SERVICE :8081
TokenIssue/ListResultConsumer
RequestStore
AMQP
Routing Keys
token.issue.request/result
token.list.request/result
payment.requested
token.validated/rejected
manager.report.request/reply
merchant.report.request/response
RabbitMQ Exchange: dtu.pay (Topic Exchange)
REST Adapter /payments
TokenValidation/ReportRequestConsumer
PaymentService • pay() • completePayment() • failPayment()
InMemoryPaymentRepository
BankProducer (SOAP Client)
PaymentRequested/ReportResponsePublisher
REST Adapter /tokens
PaymentRequested/TokenIssue/ListRequestConsumer
TokenService • issueTokens(1-5) • consumeToken() • listTokens()
8084
InMemoryTokenRepository (ConcurrentHashMap)
TokenValidation/Issue/ListResultPublisher
SOAP/HTTP
DTU FastMoney Bank (SOAP/JAX-WS) fm-00.compute.dtu.dk
EXTERNAL SYSTEMS
Operations: • transferMoneyFromTo() • createAccountWithBalance() • retireAccount() • getAccount()

Figure 7: UML Overall Architecture

Message Flow The message flow illustrates the complete lifecycle of a payment transaction across the microservices. A customer first registers through the Customer Service, then requests tokens asynchronouslythe request is published to RabbitMQ, processed by the Token Service, and the result is returned via a separate message that the customer can poll for. When making a payment, the client submits a request to the Payment Service, which creates a pending payment and publishes a token validation request. The Token Service consumes the token (making it single-use), then publishes either a validation success or rejection. Upon receiving a valid token, the Payment Service executes a bank transfer via SOAP and marks the payment as completed. For reporting, the Manager and Merchant services publish report requests to RabbitMQ, which the Payment Service consumes, processes, and replies to with the requested payment data. This event-driven approach ensures loose coupling between services while maintaining reliable request-response coordination through correlation IDs.

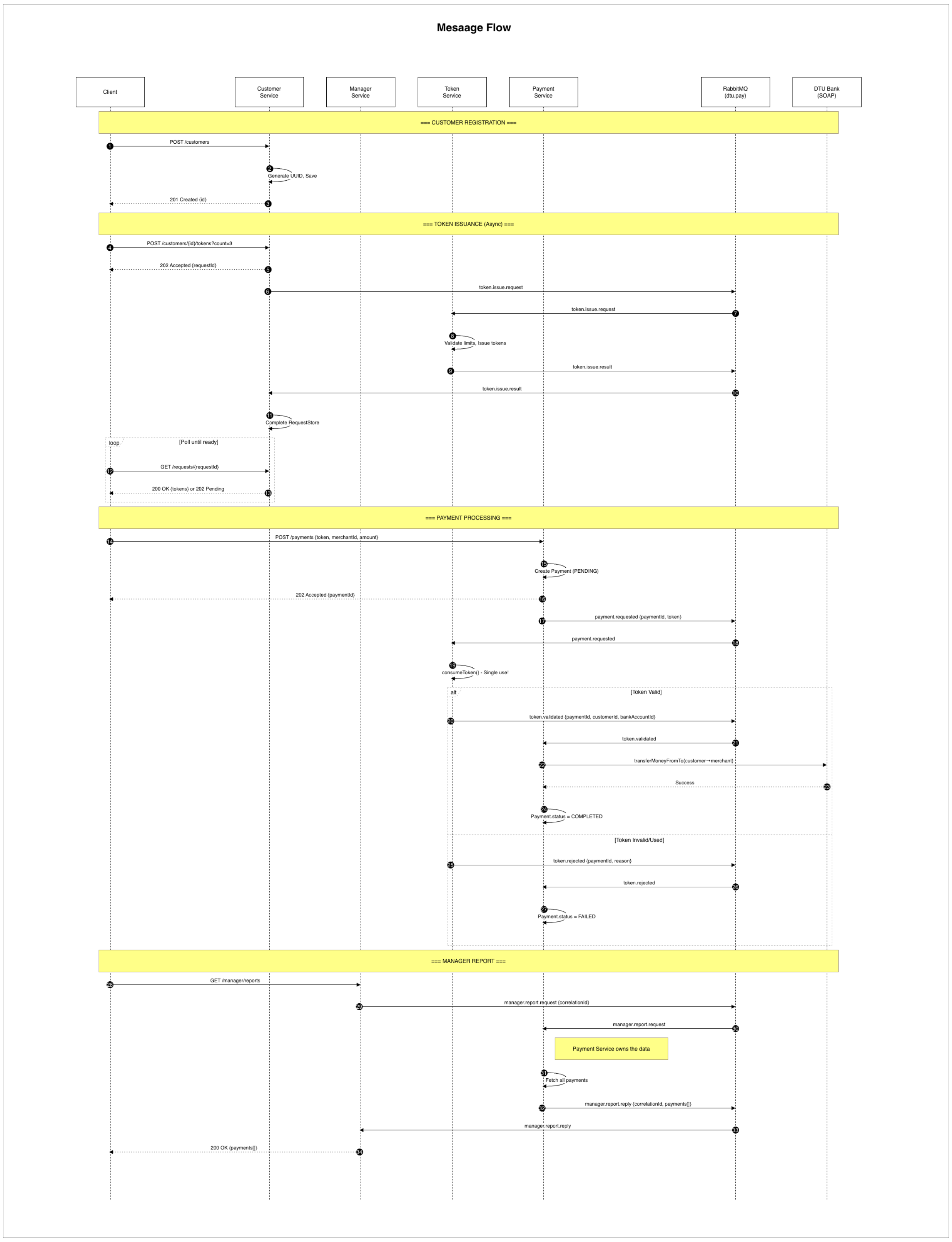

Mesaage Flow
Client
Customer Service
Manager Service
Token Service
Payment Service
RabbitMQ (dtu.pay)
DTU Bank (SOAP)
=== CUSTOMER REGISTRATION ===
1 POST /customers
2 Generate UUID, Save
3 201 Created {id}
=== TOKEN ISSUANCE (Async) ===
4 POST /customers/{id}/tokens?count=3
5 202 Accepted {requestId}
6 token.issue.request
7 token.issue.request
8 Validate limits, Issue tokens
9 token.issue.result
10 token.issue.result
11 Complete RequestStore
loop
[Poll until ready]
12 GET /requests/{requestId}
13 200 OK {tokens} or 202 Pending
=== PAYMENT PROCESSING ===
14 POST /payments {token, merchantId, amount}
15 Create Payment (PENDING)
16 202 Accepted {paymentId}
17 payment.requested {paymentId, token}
18 payment.requested
19 consumeToken() - Single use!
alt
[Token Valid]
20 token.validated {paymentId, customerId, bankAccountId}
21 token.validated
22 transferMoneyFromTo(customer→merchant)
23 Success
24 Payment.status = COMPLETED
[Token Invalid/Used]
25 token.rejected {paymentId, reason}
26 token.rejected
27 Payment.status = FAILED
=== MANAGER REPORT ===
28 GET /manager/reports
29 manager.report.request {correlationId}
30 manager.report.request
Payment Service owns the data
31 Fetch all payments
32 manager.report.reply {correlationId, payments[]}
33 manager.report.reply
34 200 OK {payments[]}

Figure 8: UML Message Flow

Hexagonal Architecture Each microservice is structured with driving adapters (REST controllers and message consumers) that receive external requests and invoke the domain service. The domain service contains business logic and interacts with driven adapters (repositories, message publishers, and the bank client) through output ports. This separation isolates the core domain from infrastructure concerns, allowing adapters to be replaced without modifying business logic.

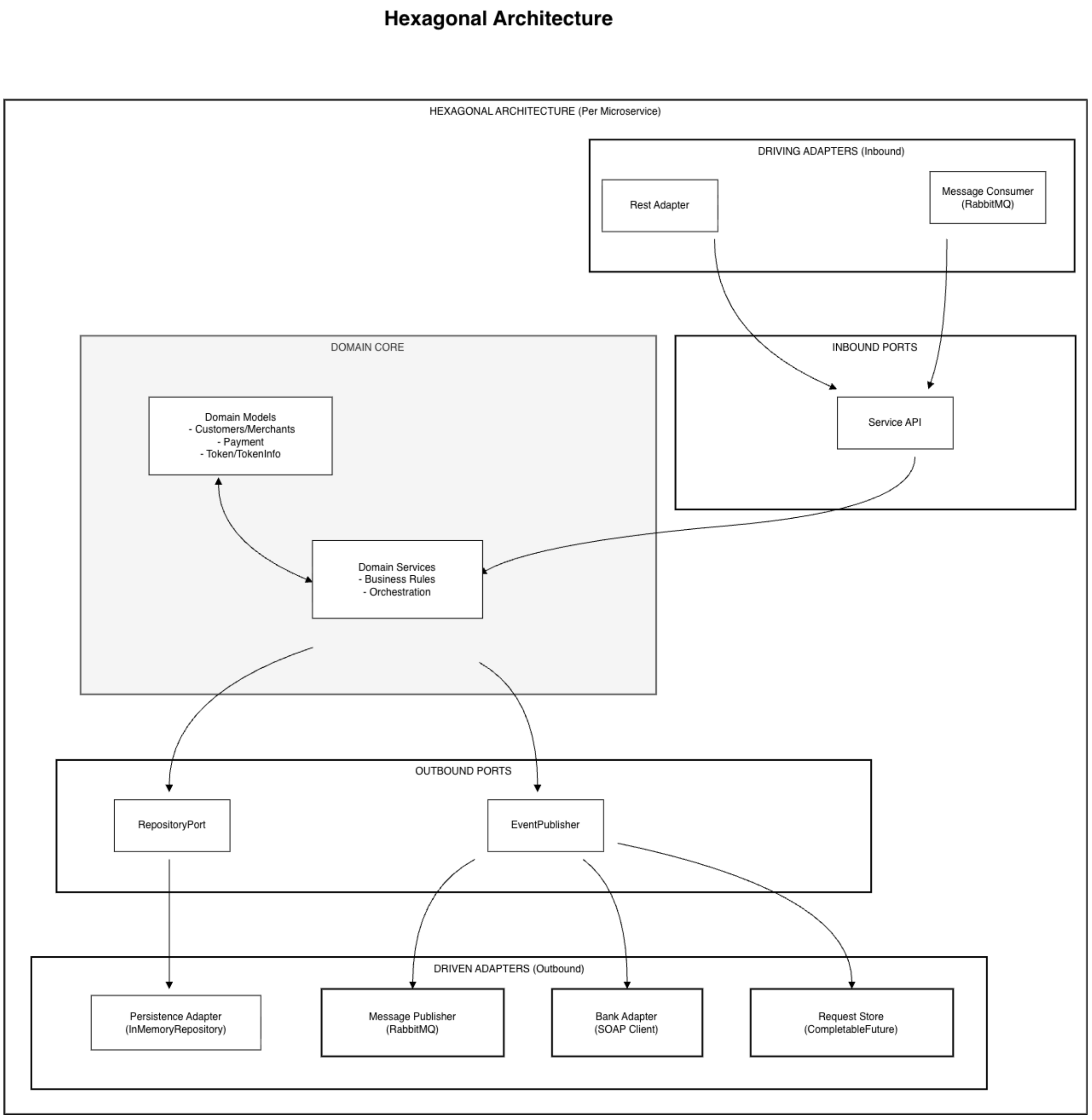

Hexagonal Architecture
HEXAGONAL ARCHITECTURE (Per Microservice)
DRIVING ADAPTERS (Inbound)
Rest Adapter
Message Consumer (RabbitMQ)
INBOUND PORTS
Service API
DOMAIN CORE
Domain Models - Customers/Merchants - Payment - Token/TokenInfo
Domain Services - Business Rules - Orchestration
OUTBOUND PORTS
RepositoryPort
EventPublisher
DRIVEN ADAPTERS (Outbound)
Persistence Adapter (InMemoryRepository)
Message Publisher (RabbitMQ)
Bank Adapter (SOAP Client)
Request Store (CompletableFuture)

Figure 9: UML Hexagonal Architecture

Data Flow Customers and merchants register with their CPR numbers and bank account IDs. Customers request tokens, which are stored with references to their customer ID and bank account. When a merchant initiates a payment, the token is consumed (deleted) to retrieve the customer's bank details, and a payment record is created linking the customer, merchant, token, and amount. The Payment Service then transfers funds between bank accounts via SOAP and updates the payment status to completed or failed. Reports aggregate payment records filtered by merchant or system-wide for managers.

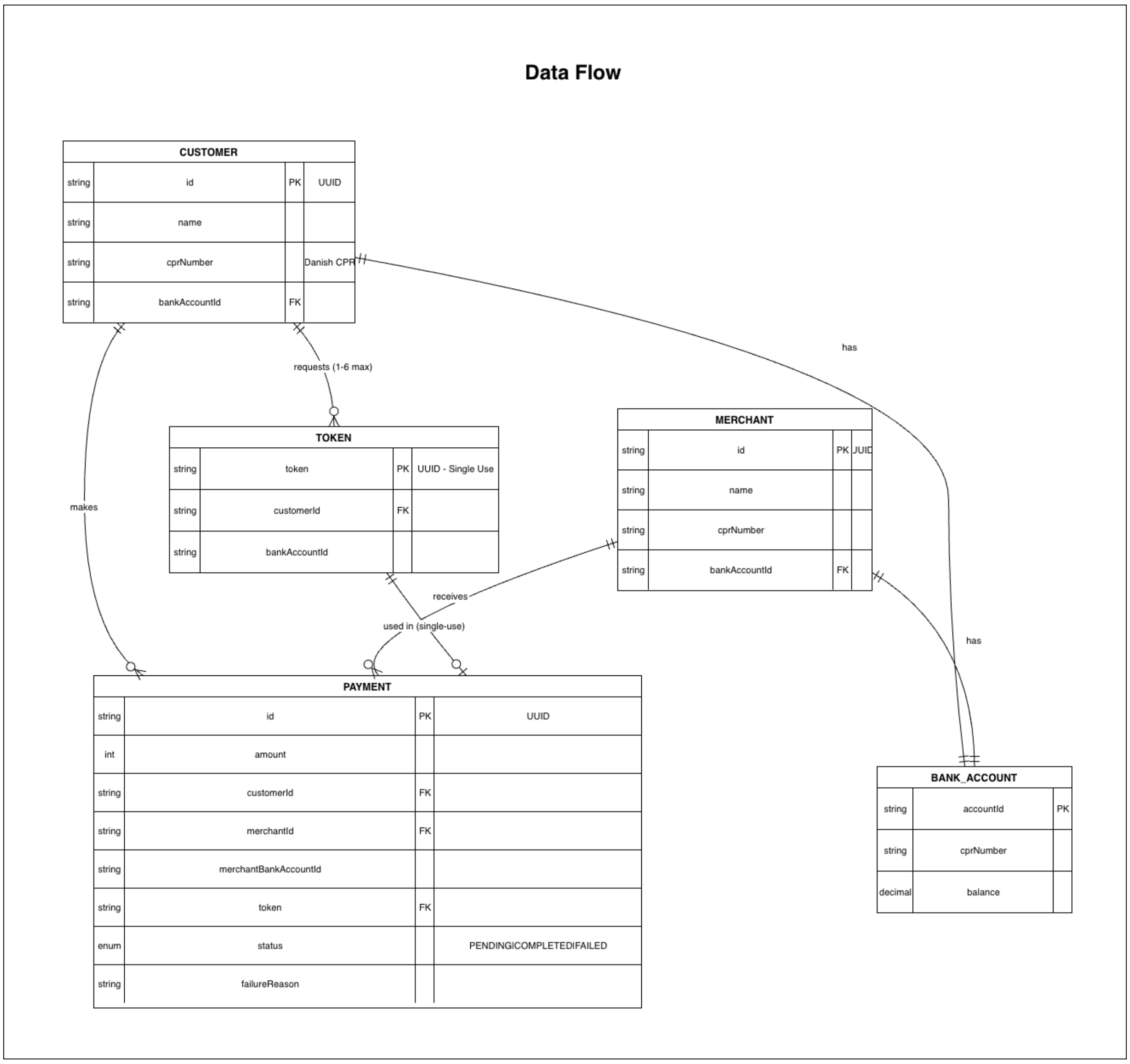


Figure 10: UML Data Flow

7 Main features

Minimim description of six features by minimum one group member each.

7.1 Register and Deregister

Registration allows customers and merchants to sign up for DTU Pay. Each user provides their name, CPR number, and bank account ID. The system stores this information and returns a unique DTU Pay ID that is used for all future operations.

7.1.1 Implementation

Both customer and merchant registration work the same way. The client sends a POST request to the service with the user details. The service creates a new entity, assigns a UUID as the internal ID, and stores it in an in-memory repository. The response includes the generated ID.

For customers, the endpoint is `POST /customers`. For merchants, it is `POST /merchants`. Both are synchronous REST calls that return immediately with the created entity.

The `CustomerService` and `MerchantService` classes contain the business logic. They validate the input and delegate to the repository port for storage. The in-memory repositories implement the port interfaces and store entities in a simple HashMap.

7.1.2 Deregister

Users can be removed from the system using DELETE requests. `DELETE /customers/{id}` removes a customer and `DELETE /merchants/{id}` removes a merchant. This cleans up the stored data for that user.

7.1.3 Tests

Registration is tested as part of the payment and token scenarios. Every test starts by registering a customer and merchant with specific details. If registration failed, the subsequent steps would not work. The cleanup after each test removes the registered users and their bank accounts.

7.2 Payment

Author: Aleksander Sønder (Translated from Danish with LLM)

The requirements for what *payment* in the system must be able to do are given. In short, DTU Pay must be able to handle the transfer of money from a *customer* to a *merchant*. DTU Pay does not create bank accounts but only registers reference IDs to existing accounts in the bank. Since a customer (in principle) can have multiple bank accounts, it is necessary to store a unique account ID for the account that will be used in DTU Pay.

During registration, DTU Pay must **not** validate whether the bank account exists or whether the customer has sufficient funds. The reason is that registration can take place long before the first payment, and the account or balance may change in the meantime. The bank itself is responsible for the business rules during a transfer: it ensures that both accounts exist and that the payer has enough money. If the rules are not met, the bank's money transfer service throws an exception. This design ensures that validation of balance and accounts takes place at the time of payment and not during registration.

7.2.1 Implementation

When a payment is initiated via the `payment-service`, `202 Accepted` is returned. The reason for this is that the payment is processed asynchronously to avoid blocking the client and to separate responsibilities between services (payment, token validation, and bank transfer). Upon receiving the payment request, a `Payment` entity is created with status `PENDING` and stored in the repository. Then the payment-service publishes an event on RabbitMQ (routing key `payment.requested`) containing a reference to the payment ID and the customer's token.

The token-service consumes this event and validates the token. As a result, the token-service publishes either `token.validated` or `token.rejected`. The payment-service has a consumer listening to these events. If the token is validated, the payment-service completes the payment by calling the bank's `transferMoneyFromTo` with the customer's bank account as sender and the merchant's bank account as receiver. If the bank transfer succeeds, the payment status is updated to `COMPLETED`. If the bank rejects the transfer (e.g., account does not exist or insufficient funds), the payment is updated to `FAILED` and the error reason is stored. If the token is rejected, the payment is also marked as `FAILED`.

This solution was chosen because it matches the requirements that the bank is the sole authority for transfer rules, while DTU Pay can handle workflows across services without tight coupling. RabbitMQ is used to create loose coupling between services and to make token validation and payment separate areas of responsibility. Furthermore, a ports-and-adapters / hexagonal structure makes it easier to test the domain logic in isolation from infrastructure (HTTP and messaging).

7.2.2 Tests

The test strategy is divided into two levels:

Unit tests (payment-service):

Unit tests have been written for the payment domain logic, with focus on correct status handling and error scenarios. These tests verify that a payment is created as `PENDING` and later updated to `COMPLETED` or `FAILED` depending on the outcome of the bank transfer and token validation. Unit tests provide fast feedback and make it easier to isolate errors to specific methods without dependency on Docker, RabbitMQ, or external services.

End-to-end tests (Cucumber):

End-to-end tests have been written as Gherkin scenarios for the payment flow. The scenarios cover both success and failure situations, for example:

Listing 1: Cucumber scenario: Payment rejected due to insufficient funds

```
Scenario: Payment is rejected when the customer has insufficient
   funds
    Given a customer with name ‘‘Alice‘‘, CPR ‘‘111111-1111‘‘, and
       balance 50
    And a merchant with name ‘‘Bob‘‘, CPR ‘‘222222-2222‘‘, and balance
       1000
    And the customer has a valid token
    When the merchant initiates a payment for 100 kr by the customer
       using the token
    Then the payment is rejected
    And the balance of the customer at the bank is 50 kr
    And the balance of the merchant at the bank is 1000 kr
```

These tests verify that the entire system works across services (customer-, token-, payment-, and merchant-service) as well as the integration with the bank. End-to-end tests are the most important for validating contracts between services (endpoints, payloads, and routing keys), but they are slower and more dependent on infrastructure than unit tests. The combination of unit tests and end-to-end tests therefore provides both fast error detection during development and high confidence that the overall solution behaves correctly.

7.3 Token Management

Author: Sigurd de Neergaard
Tokens are a central concept in DTU Pay, as they act as the customers consent for a payment while preserving privacy and preventing reuse. The token management design therefore focuses on enforcing strict business rules around token limits and single-use consumption, while keeping tokens opaque and non-guessable. Communication with the token service is intentionally indirect, occurring through customer- and payment-facing flows and asynchronous messaging, to maintain encapsulation and decouple services. Testing is correspondingly focused on externally observable behavior through acceptance tests, complemented by unit tests for core logic, ensuring both correctness of the token rules and reliable integration with the payment flow.

7.3.1 Implementation

The Token Management service is responsible for issuing, storing, and validating tokens for customers. The service is implemented using a layered and service-oriented structure inspired by hexagonal architecture. While the core ideas of separating domain logic from infrastructure are followed, a fully strict hexagonal architecture was not feasible due to project constraints, such as the requirement to use in-memory persistence rather than an external database.

The domain layer contains the token-related business logic, including rules for issuing, listing, and consuming tokens. This logic is implemented in the `TokenService`, which operates on domain models such as `TokenInfo` and token identifiers. Persistence is abstracted behind a repository port, allowing the domain logic to remain independent of the concrete storage implementation. For this project, an in-memory repository is used.

All functional interaction with the token service occurs either through asynchronous messaging or via endpoints exposed by the customer and payment services. RabbitMQ is used for asynchronous communication, where the token service consumes requests related to token issuance, listing, and validation, and publishes corresponding result events. This allows other services, in particular the payment service, to validate tokens without direct synchronous dependencies.

A REST endpoint for directly creating tokens exists as a leftover from an earlier iteration of the architecture and is primarily used for testing purposes. Equivalent functionality is implemented through the customer-facing flow, and the endpoint was kept to avoid introducing new errors late in the project. The intended usage of the token service is therefore through customer- and payment-driven interactions rather than direct access.

7.3.2 Tests

The token functionality is tested at multiple levels using a combination of BDD-style acceptance tests, integration tests, and unit tests. The primary focus is on BDD acceptance tests written in Cucumber, which describe the expected external behavior of the system and enforce the business rules related to token issuance, listing, and consumption.

The token-related acceptance tests cover scenarios where a customer requests too many tokens at once, requests tokens in a valid sequence, and attempts to request additional tokens while already holding active tokens. These scenarios verify that token requests are correctly accepted or rejected according to the defined constraints, and that the list of active tokens returned to the customer accurately reflects the current system state.

Token usage is also tested indirectly as part of the payment acceptance tests. These scenarios verify that a payment succeeds when a valid token is supplied and is rejected when no token is provided or the same token is provided a second time. This confirms that token validation is correctly integrated into the payment flow and that communication between the token and payment services works as intended.

At the implementation level, unit tests are used to validate core components in isolation. The `TokenService` is tested using mocked repository ports to verify business rules such as token limits, single-use consumption, and error handling. The in-memory token repository is tested to ensure correct storage, lookup, and consumption behavior, including enforcement of single-use tokens. In addition, REST-level integration tests validate the remaining token creation endpoint to ensure correct request handling and error responses.

The direct token creation endpoint is a leftover from an early monolithic version of the system used during the first days of the course. In the current architecture, token creation is intended to happen through the customer-facing flow and via asynchronous messaging. The endpoint is retained only to support test setup, specifically in scenarios where a valid token must exist (e.g., the *Given a customer has a valid token* step), and was not removed late in the project to avoid introducing unintended regressions.

7.4 Manager Report

The manager report gives a full view of all payments in the system. This is meant for DTU Pay administrators who need to see the complete payment history. Unlike customer and merchant reports, the manager report includes all details: customer ID, merchant ID, token used, and amount.

7.4.1 Implementation

The manager service is a separate microservice that only handles report requests. When a manager calls `GET /manager/reports`, the following happens:

1. The ManagerResource generates a correlation ID
2. It stores a CompletableFuture in the ManagerReportStore

3. It publishes a request message to RabbitMQ with the correlation ID
4. The payment service receives the request and looks up all payments
5. The payment service publishes a response with the same correlation ID
6. The manager service receives the response and completes the future
7. The REST endpoint returns the list of payments

The timeout is set to 30 seconds. If no response comes back in that time, the endpoint returns a 504 Gateway Timeout error.

7.4.2 Tests

Listing 2: Cucumber scenario: Manager report

```
Scenario: Manager retrieves a report of payments
  Given a customer with name ``Alice``, CPR ``123956-0001``, and balance
     1000
  And a merchant with name ``ShopOne``, CPR ``123956-0002``, and balance
     1000
  And the customer has a valid token
  When the merchant initiates a payment for 200 kr by the customer
     using the token
  And the manager requests the report
  Then the report contains the payment of 200 kr
```

This test creates a payment first, then verifies the manager can see it in the report. The step definitions call the manager service endpoint and check that the returned list contains a payment with the expected amount.

7.5 Customer Report

Customers can request a report of their own payments. This report shows what they paid, to which merchant, and which token was used. The customer cannot see other customers' payments.

7.5.1 Implementation

The customer report works like the manager report but with filtering. When a customer calls `GET /customers/{id}/report`, the customer service sends a request to the payment service with the customer ID. The payment service filters all payments to only include those where the customer was the payer.

The async flow uses the same pattern:

1. Generate correlation ID and store a future
2. Publish request to RabbitMQ with customer ID
3. Payment service filters and responds
4. Customer service completes the future and returns data

The response includes the merchant ID and the token used for each payment. This lets customers see their payment history and verify which tokens they used.

7.5.2 Tests

Listing 3: Cucumber scenario: Customer report

```
Scenario: Customer retrieves a report of their payments
  Given a customer with name ``Alice``, CPR ``123956-0011``, and balance
     1000
  And a merchant with name ``ShopOne``, CPR ``123956-0012``, and balance
     1000
  And the customer has a valid token
  When the merchant initiates a payment for 150 kr by the customer
     using the token
  And the customer requests their report
  Then the customer report contains the payment of 150 kr
  And the customer report contains the merchant
  And the customer report contains the token used
```

The test verifies that the customer can see their own payment details including the merchant and token information.

7.6 Merchant Report

Merchants can see a report of payments they received. For privacy reasons, the merchant report does not include customer IDs. This protects customer anonymity - merchants only see that a payment happened, the amount, and the token used.

7.6.1 Implementation

The merchant report follows the same async pattern as the other reports. The endpoint is `GET /merchants/{id}/reports`. The merchant service publishes a request to the payment service, which filters payments by merchant ID and strips out customer information before responding.

The key privacy feature is that the response DTO only includes:

Payment amount

Token used

Timestamp

The customer ID is deliberately omitted. This means merchants can verify they received payments and match them to tokens, but cannot link payments to specific customers.

7.6.2 Tests

The merchant report functionality is tested by creating a payment and then verifying the merchant can see it in their report without customer details. The test checks that the amount and other non-identifying information is present while customer identity is hidden.

8 Reflections

Author: Nichita Railean

8.1 What went well

The project helped us learn a lot about building distributed systems. Starting with Event Storming was useful because it gave everyone a shared understanding of what the system should do before we wrote any code. The hexagonal architecture made it easier to test services in isolation and swap out implementations when needed.

Using RabbitMQ for async messaging turned out to be a good choice. It forced us to think carefully about how services communicate and handle failures. The request/reply pattern with correlation IDs worked well for operations like generating reports where we needed to wait for data from another service.

Docker Compose made it easy to run the whole system locally. Once we had the Dockerfiles and compose file working, anyone on the team could start all services with one command. Jenkins helped catch integration issues early since it ran the full test suite on every push.

8.2 What was hard

The async messaging took time to get right. Debugging issues where messages were not being received or were going to the wrong queue was frustrating. We had to add a lot of logging to figure out what was happening.

Merging code from different team members was sometimes painful. When multiple people changed the same service, we had to carefully resolve conflicts and make sure nothing broke. The Cucumber tests helped catch regressions but running them took a while.

Keeping all the services in sync was also tricky. When we changed a DTO in one service, we had to update it everywhere else too. A shared library for common types might have helped here.

8.3 What we would do differently

If we started again, we would set up the microservice structure earlier. We spent time refactoring from a monolith to separate services, which could have been avoided with better upfront planning.

We would also add more unit tests. Most of our testing was end-to-end with Cucumber, which is good for verifying behavior but slow for feedback during development. More unit tests would have made refactoring faster and safer.

Finally, we would document the messaging contracts more formally. We had to look at the code to understand what messages each service expected. An API specification or schema would have made integration easier.

8.4 Learning outcomes

The project covered all the course learning objectives. We built a working service-oriented application as a team. We used agile methods like Event Storming and iterative development. We composed services using both REST and messaging. We tested the system with BDD acceptance tests. We deployed with Docker and Jenkins. We discussed coordination patterns like correlation

IDs. And we can now explain why architectures like hexagonal and microservices are useful for building maintainable systems.