Software Development of Web Services

Asher Sharif (240193)
Jonas Puidokas (137282)
Anders Friis Persson (233469)
Haissam Kamel Barghooth (233470)
Bassel Mohammad Beiroumi (194309)
Muhammad Zaid Mehmood (240177)

January 24, 2025

1 Introduction

The project DTU-Pay focuses on Microservices-based scalable platform. DTU pay helps users exchange money between customers and merchants from their mobile phones. We have implemented modern techniques in DTU-pay to deliver the necessary business functions.

The project uses the microservices architecture which enables the ease of updation and maintenance of the system. Each Microservice handles certain responsibilities, such as token management, account management, and data reporting. To make communication possible between these services RabbitMQ is playing an important role in ensuring asynchronous interaction. DTU-Pay is implemented using Maven for managing and building the project and also Quarkus is used for developing efficient endpoints in the project. One of the important function is token management, which can be requested by the customer to perform the transaction, which playes important role in security and privacy. These tokens are used to prevent the transection from tampering and also identification of customer as the token becomes non-usable after the transaction is completed. Marchant microservice can verify these tokens during payments by handling the underlying process by DTU pay.

DTU-Pay allows money transfers from customers directly to merchant accounts. To handle transactions DTU connects with SOAP-based APIs which helps keep the record and enables its report-generation function. The project works on CI/CD piplines using Jenkins and GitLab, by using this setup system operations like code building, Docker image control, and testing can happen automatically. Development of tests plays a critical role in this project because it uses Cucumber and jUnit tests to verify complete system performance and end-to-end functionality. This end-to-end evaluation system uses both Cucumber tests to validate user behavior and jUnit tests to test individual software blocks. this combined approach detects errors both quickly and protects ongoing functionality during the development cycle. This approach makes it simpler to integrate new functions without any major change in the system's stability. Automated testing makes testing work easier and speeds up results plus maintains continuous integration in the project.

The modern design and quality improvement methods together with comprehensive testing help DTU-Pay create high-quality payment solutions. We used Agile methods during development to run mob programming sessions and daily stand-up meetings that protected code quality while spreading team learning as we progressed in building this project.

2 Collaboration

An iterative Agile methodology was used to guide the development of the project effectively. The tasks were systematically defined and managed using GitLab, with **milestones** established to describe different development iterations. The first iteration prioritized the implementation of core functionalities, including the execution of payment processes. Subsequent iterations expanded on this foundation, introducing additional features such as reporting capabilities, code refinement for improved readability and maintainability, and other value-adding elements.

Collaboration within the team was structured and deliberate. Small groups participated in mob programming sessions, which facilitated the sharing of knowledge about course content and fostered a collaborative environment to ensure high software quality.

Daily synchronization meetings, modeled after Scrum Daily Standups, were conducted to maintain alignment within the team. These meetings served as a forum for members to report on completed tasks, plan future activities, and identify any obstacles to progress. **Issue board** was used to coordinate the effort in the team. This iterative and communicative approach ensured that the project adhered to Agile principles, promoting transparency, adaptability, and continuous improvement.

3 Event Storming

To kickstart our project and gain a comprehensive overview of its features and actions, we conducted an Event Storming session. This collaborative exercise, held at the project's inception, aimed to foster a shared understanding of the project's scope among team members.

We began by establishing a color-coded legend, ensuring agreement on the meaning of different colored notes. This step laid the foundation for a clear and consistent visual language throughout the session. We then systematically mapped out the entire process, from customer registration on the left to payment processing on the right, using various colored post-it notes.

Initially, we found it challenging to grasp the nuances of the Event Storming methodology, particularly in distinguishing between commands, policies, aggregates, and other components of our application. However, as we progressed, these concepts became clearer, and we grew more comfortable with the process.

The final result was a comprehensive visualization of our system's end-to-end process, which proved invaluable in guiding our development efforts. For instance, the policies identified during the session directly informed our implementation of methods to handle events from the message queue.

In retrospect, while the Event Storming session presented initial challenges, it ultimately provided us with a powerful tool for aligning our team's vision and technical approach, setting a strong foundation for the project's development phase.

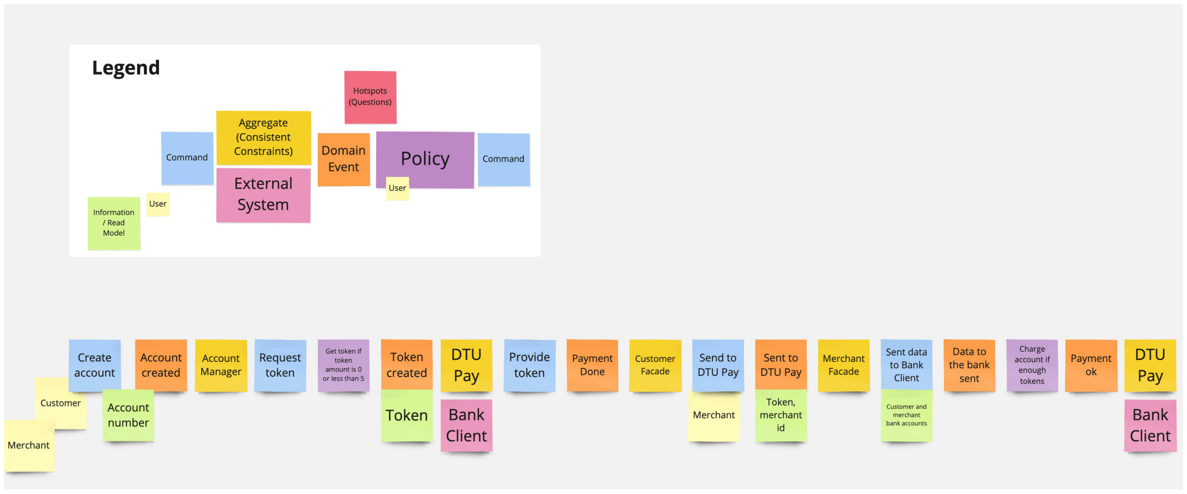

Figure 1: Event Storming

4 Continuous Integration & Delivery

To ensure a seamless and reliable development process, we leveraged Jenkins for CI/CD and integrated it with GitLab to manage code repositories and automate the pipeline. This setup was essential in maintaining the stability of our codebase while supporting the microservices architecture of the project.

Jenkins played a central role in the CI/CD workflow by automating several critical tasks:

- **Code Compilation:** Jenkins automatically compiled the code for every pipeline run, ensuring that changes were free from syntax errors and ready for execution.
- **Docker Image Management:** It built Docker images from the compiled code and deployed them, streamlining the packaging and deployment of services in isolated containers.
- **Testing:** Jenkins executed tests, including:
  - **Cucumber End-to-End Tests:** Validating that the system worked as expected end-to-end.
  - **JUnit Service Tests:** Verifying the functionality and reliability of individual services within the microservices ecosystem.
- **Test Reporting:** Jenkins provided a detailed overview of all executed tests, highlighting successes and failures to facilitate prompt debugging and quality assurance.

GitLab served as the version control and code management platform. The integration between GitLab and Jenkins enabled the following workflow:

1. **Triggered Pipelines:** For every push to the main branch or submission of a pull request in GitLab, Jenkins automatically initiated the CI/CD pipeline.
2. **Code Validation:** The pipeline ran all stages—from code compilation to Docker image deployment and test execution.
3. **Regression Prevention:** By validating the entire codebase against the latest changes, Jenkins ensured that new contributions did not introduce regressions or break existing functionality.

This integrated approach significantly improved the development lifecycle:

- **Early Detection of Issues:** Automated testing identified bugs and inconsistencies early in the development process, reducing the time and effort needed for debugging later.
- **Code Stability:** The validation pipeline ensured that every code change adhered to predefined quality standards and did not disrupt previously stable features.
- **Enhanced Collaboration:** Developers could submit and review code confidently, knowing that automated checks validated the integrity of their contributions.

Using Jenkins and GitLab in combination, we maintained a robust automated pipeline that reinforced the quality and reliability of our application.

5 Architecture

5.1 Description of the Architecture

The architecture of our system is based on a microservices approach, enabling modularity, scalability, and flexibility. Each microservice is designed to encapsulate specific business functionalities while remaining independent and reusable. The system consists of five main microservices: `dtu-pay`, `account-manager`, `payment-service`, `token-manager`, and `reporting-service`. These services collectively form a robust and scalable payment ecosystem by utilizing RESTful APIs for synchronous interactions and message queues, such as RabbitMQ, for asynchronous communication.

5.2 Microservices

- **dtu-pay**: Acts as the central entry point for all users, including customers and merchants. This service is responsible for handling user registration, deregistration, token generation, and payment initiation requests. It provides the primary interface for interaction with the system and coordinates between other microservices.
- **account-manager**: Manages account-related operations such as the creation, retrieval, and deregistration of customer and merchant accounts. It acts as the authoritative source of account data and ensures consistency across the system.
- **payment-service**: Handles the payment process between customers and merchants by validating tokens, retrieving account details, and performing the actual transaction. It ensures secure and reliable transfer of funds within the system.
- **token-manager**: Manages the lifecycle of tokens, including their generation, validation, and expiration. This service ensures that only valid and secure tokens are used for transactions, providing an additional layer of security to the payment process.
- **reporting-service**: Responsible for collecting, processing, and reporting on transaction data. It listens to events such as `PaymentReported` and provides detailed transaction reports for both merchants and customers. The service utilizes message queues like RabbitMQ to handle asynchronous events and deliver the requested transaction information back to the relevant parties.

5.3 REST Interfaces

The system uses RESTful APIs to enable communication between microservices and external clients. The interfaces are designed following REST principles, ensuring consistency, simplicity, and ease of use. The table below provides an overview of the REST endpoints:

Resource URI Path	**HTTP Method**	**Function**
	Merchant Endpoints	
/merchant/register	POST	Registers a new merchant with the provided details. Returns a unique merchant ID.
/merchant/deregister/merchantId	DELETE	Deregisters a merchant identified by the given ID.
/pay	POST	Initiates a payment for a specified merchant, token, amount, and description. Returns payment status and transaction ID.
/merchant/get-transactions/merchantId	GET	Retrieves the transaction history for a specific merchant.
	Customer Endpoints	
/customer/register	POST	Registers a new customer with the provided details. Returns a unique customer ID.
/customer/deregister/customerId	DELETE	Deregisters a customer identified by the given ID.
/customer/getTokens/customerId	GET	Retrieves the list of tokens associated with a customer.
/customers/get-transactions/customerId	GET	Retrieves the transaction history for a specific customer.
/customer/createTokens	POST	Generates tokens for a customer based on the provided request.

Table 1: Overview of REST Interfaces for Merchant and Customer Facades

5.4 Design Decisions

The design of the system prioritizes modularity, clarity, and scalability. RESTful principles are followed rigorously to ensure each resource is represented by intuitive URLs and operations that align with HTTP methods. Nouns are used for resource names, while actions are represented by HTTP verbs (e.g., POST for creation and DELETE for removal). This approach reduces complexity and enhances the maintainability of the system.

Resource URLs are hierarchically structured to reflect the relationships between entities, such as customers, merchants, and their transactions. For example, `/merchants/merchantId/transactions` logically groups a merchant's transaction history under its unique identifier. This structure aids developers in understanding and utilizing the endpoints efficiently.

5.5 Microservices Communication

The microservices communicate through a combination of RESTful APIs and message queues, such as RabbitMQ. REST APIs are employed for synchronous interactions, where an immediate response is necessary, such as retrieving account details or initiating payments. Message queues are used for asynchronous communication, enabling services to publish and consume events decoupled from one another.

The system uses distinct message queues for different event types, ensuring separation of concerns and reducing potential bottlenecks. For instance, customer-related events such as `CustomerRegistrationRequested`

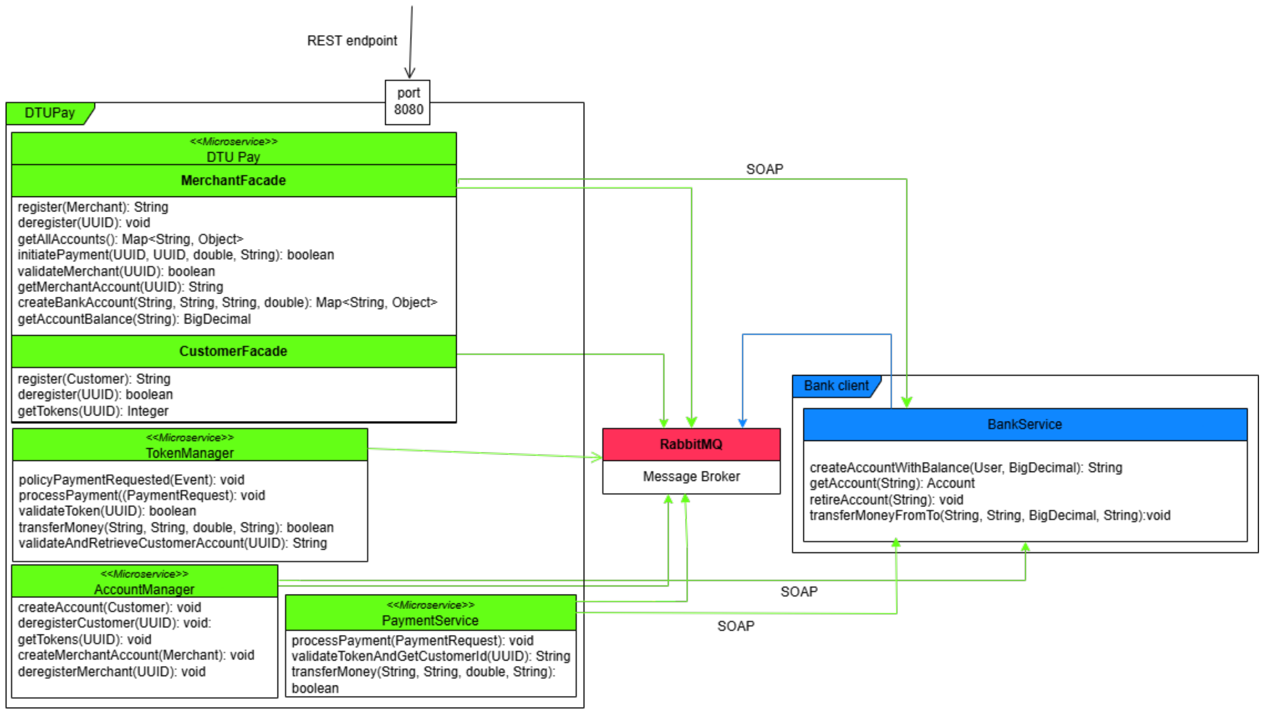
REST endpoint
port 8080
DTUPay
<<Microservice>>
DTU Pay
MerchantFacade
register(Merchant): String
deregister(UUID): void
getAllAccounts(): Map<String, Object>
initiatePayment(UUID, UUID, double, String): boolean
validateMerchant(UUID): boolean
getMerchantAccount(UUID): String
createBankAccount(String, String, String, double): Map<String, Object>
getAccountBalance(String): BigDecimal
CustomerFacade
register(Customer): String
deregister(UUID): boolean
getTokens(UUID): Integer
<<Microservice>>
TokenManager
policyPaymentRequested(Event): void
processPayment((PaymentRequest): void
validateToken(UUID): boolean
transferMoney(String, String, double, String): boolean
validateAndRetrieveCustomerAccount(UUID): String
<<Microservice>>
AccountManager
createAccount(Customer): void
deregisterCustomer(UUID): void:
getTokens(UUID): void
createMerchantAccount(Merchant): void
deregisterMerchant(UUID): void
<<Microservice>>
PaymentService
processPayment(PaymentRequest): void
validateTokenAndGetCustomerId(UUID): String
transferMoney(String, String, double, String): boolean
RabbitMQ
Message Broker
SOAP
Bank client
BankService
createAccountWithBalance(User, BigDecimal): String
getAccount(String): Account
retireAccount(String): void
transferMoneyFromTo(String, String, BigDecimal, String):void
SOAP
SOAP

Figure 2: Project Architecture

and `CustomerDeregistrationRequested` are processed through a dedicated queue. Similarly, token-related events like `TokenValidationRequested` and payment-related events such as `PaymentRequested` have their own queues. Additionally, the `reporting-service` subscribes to events like `PaymentReported` to generate detailed transaction reports for both merchants and customers.

This event-driven architecture improves scalability and reliability by allowing services to handle messages at their own pace. It also simplifies error handling, as failed messages can be retried without affecting other parts of the system. Overall, this communication strategy ensures high availability and fault tolerance.

6 Classes

Following are class diagrams for all the microservices in DTU Pay.

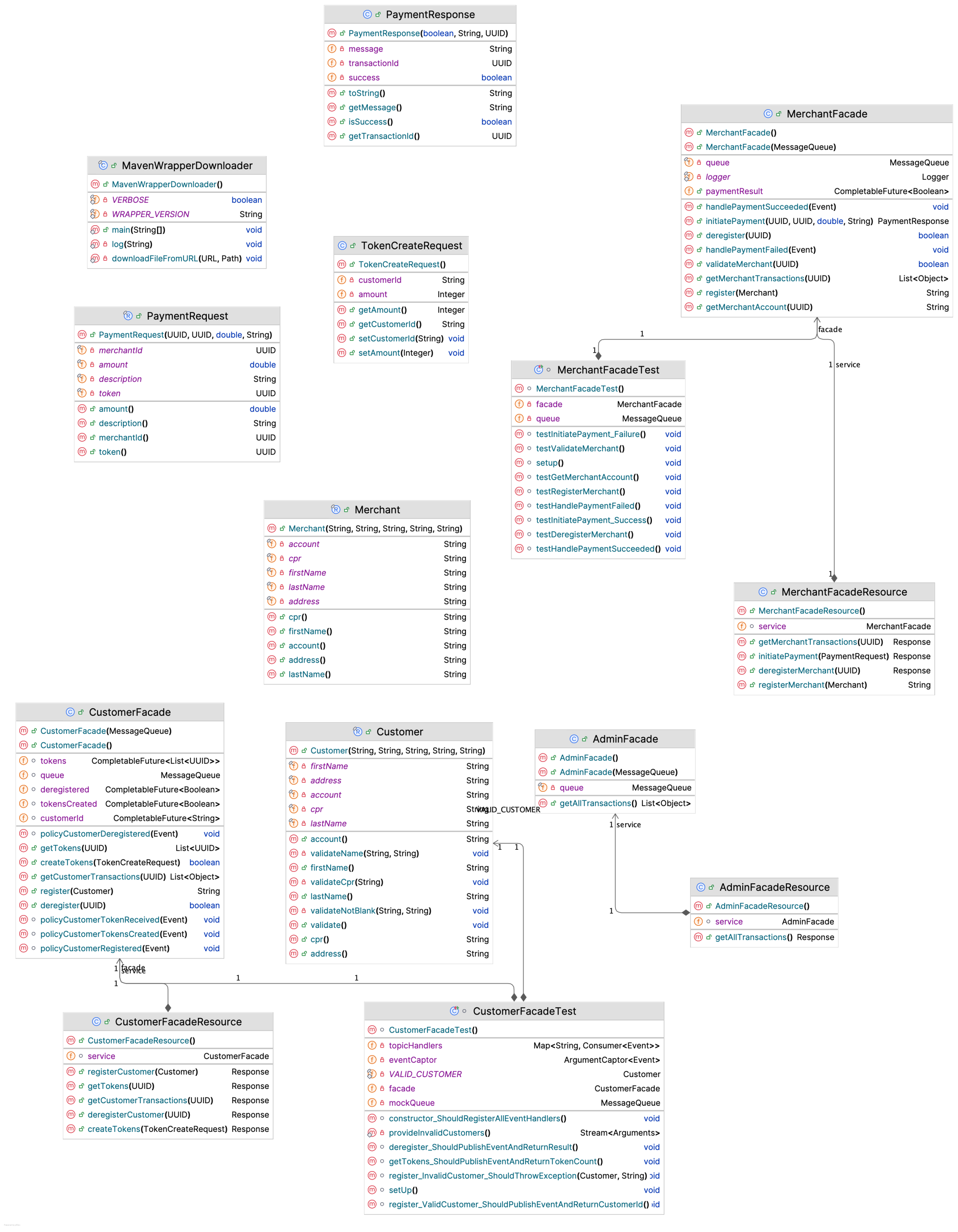
PaymentResponse
PaymentResponse(boolean, String, UUID)
message String
transactionId UUID
success boolean
toString() String
getMessage() String
isSuccess() boolean
getTransactionId() UUID
MerchantFacade
MerchantFacade()
MerchantFacade(MessageQueue)
queue MessageQueue
logger Logger
paymentResult CompletableFuture<Boolean>
handlePaymentSucceeded(Event) void
initiatePayment(UUID, UUID, double, String) PaymentResponse
deregister(UUID) boolean
handlePaymentFailed(Event) void
validateMerchant(UUID) boolean
getMerchantTransactions(UUID) List<Object>
register(Merchant) String
getMerchantAccount(UUID) String
MavenWrapperDownloader
MavenWrapperDownloader()
VERBOSE boolean
WRAPPER_VERSION String
main(String[]) void
log(String) void
downloadFileFromURL(URL, Path) void
TokenCreateRequest
TokenCreateRequest()
customerId String
amount Integer
getAmount() Integer
getCustomerId() String
setCustomerId(String) void
setAmount(Integer) void
PaymentRequest
PaymentRequest(UUID, UUID, double, String)
merchantId UUID
amount double
description String
token UUID
amount() double
description() String
merchantId() UUID
token() UUID
facade
1
1
service
1
MerchantFacadeTest
MerchantFacadeTest()
facade MerchantFacade
queue MessageQueue
testInitiatePayment_Failure() void
testValidateMerchant() void
setup() void
testGetMerchantAccount() void
testRegisterMerchant() void
testHandlePaymentFailed() void
testInitiatePayment_Success() void
testDeregisterMerchant() void
testHandlePaymentSucceeded() void
1
Merchant
Merchant(String, String, String, String, String)
account String
cpr String
firstName String
lastName String
address String
cpr() String
firstName() String
account() String
address() String
lastName() String
MerchantFacadeResource
MerchantFacadeResource()
service MerchantFacade
getMerchantTransactions(UUID) Response
initiatePayment(PaymentRequest) Response
deregisterMerchant(UUID) Response
registerMerchant(Merchant) String
CustomerFacade
CustomerFacade(MessageQueue)
CustomerFacade()
tokens CompletableFuture<List<UUID>>
queue MessageQueue
deregistered CompletableFuture<Boolean>
tokensCreated CompletableFuture<Boolean>
customerId CompletableFuture<String>
policyCustomerDeregistered(Event) void
getTokens(UUID) List<UUID>
createTokens(TokenCreateRequest) boolean
getCustomerTransactions(UUID) List<Object>
register(Customer) String
deregister(UUID) boolean
policyCustomerTokenReceived(Event) void
policyCustomerTokensCreated(Event) void
policyCustomerRegistered(Event) void
Customer
Customer(String, String, String, String, String)
firstName String
address String
account String
cpr String
lastName String
account() String
validateName(String, String) void
firstName() String
validateCpr(String) void
lastName() String
validateNotBlank(String, String) void
validate() void
cpr() String
address() String
VALID_CUSTOMER
AdminFacade
AdminFacade()
AdminFacade(MessageQueue)
queue MessageQueue
getAllTransactions() List<Object>
1
service
1
AdminFacadeResource
AdminFacadeResource()
service AdminFacade
getAllTransactions() Response
1
1
1
facade
service
1
1
1
CustomerFacadeResource
CustomerFacadeResource()
service CustomerFacade
registerCustomer(Customer) Response
getTokens(UUID) Response
getCustomerTransactions(UUID) Response
deregisterCustomer(UUID) Response
createTokens(TokenCreateRequest) Response
CustomerFacadeTest
CustomerFacadeTest()
topicHandlers Map<String, Consumer<Event>>
eventCaptor ArgumentCaptor<Event>
VALID_CUSTOMER Customer
facade CustomerFacade
mockQueue MessageQueue
constructor_ShouldRegisterAllEventHandlers() void
provideInvalidCustomers() Stream<Arguments>
deregister_ShouldPublishEventAndReturnResult() void
getTokens_ShouldPublishEventAndReturnTokenCount() void
register_InvalidCustomer_ShouldThrowException(Customer, String) void
setUp() void
register_ValidCustomer_ShouldPublishEventAndReturnCustomerId() void

Figure 3: DTU Pay

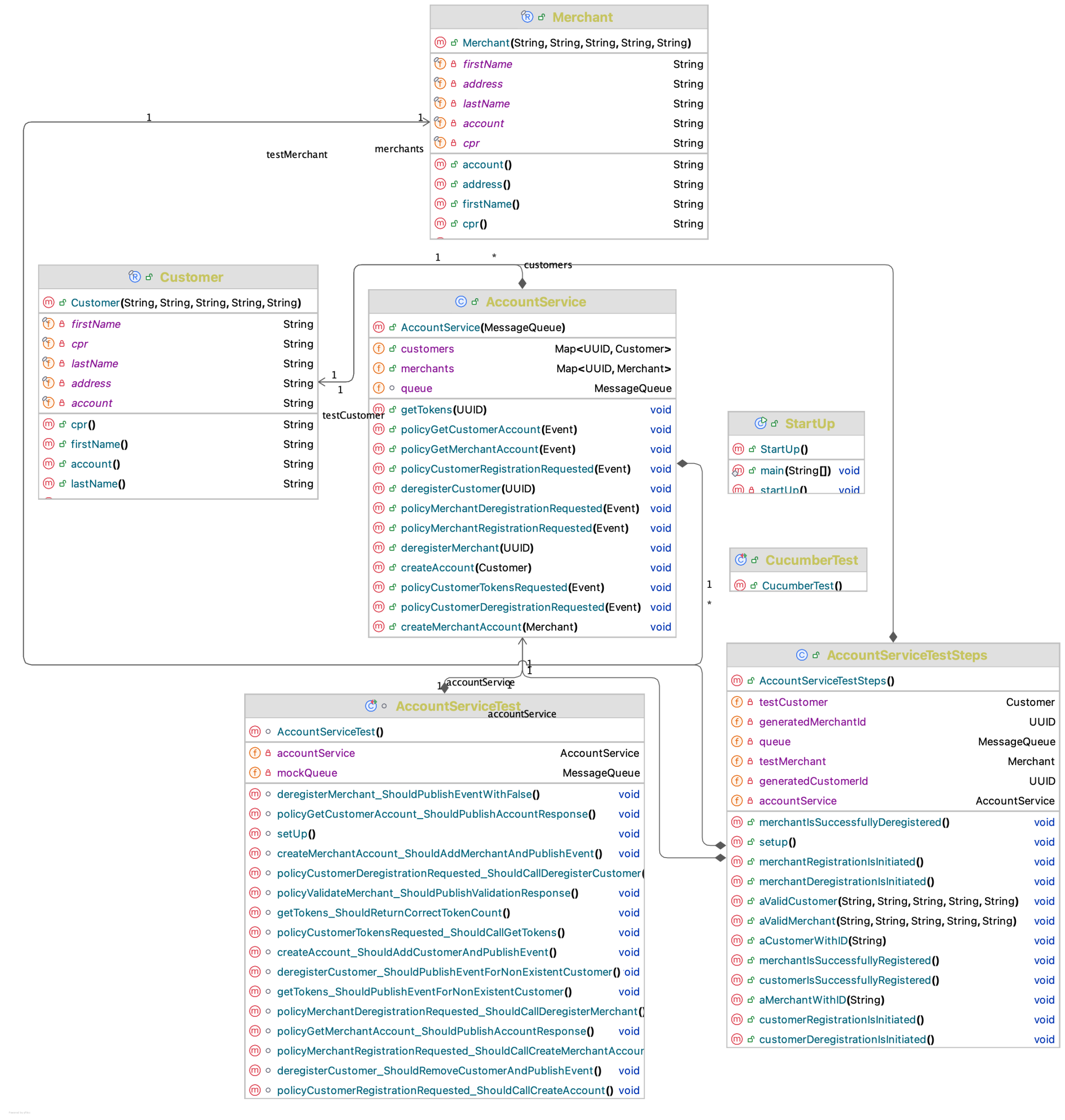
Merchant
Merchant(String, String, String, String, String)
firstName String
address String
lastName String
account String
cpr String
account() String
address() String
firstName() String
cpr() String
Customer
Customer(String, String, String, String, String)
firstName String
cpr String
lastName String
address String
account String
cpr() String
firstName() String
account() String
lastName() String
AccountService
AccountService(MessageQueue)
customers Map<UUID, Customer>
merchants Map<UUID, Merchant>
queue MessageQueue
getTokens(UUID) void
policyGetCustomerAccount(Event) void
policyGetMerchantAccount(Event) void
policyCustomerRegistrationRequested(Event) void
deregisterCustomer(UUID) void
policyMerchantDeregistrationRequested(Event) void
policyMerchantRegistrationRequested(Event) void
deregisterMerchant(UUID) void
createAccount(Customer) void
policyCustomerTokensRequested(Event) void
policyCustomerDeregistrationRequested(Event) void
createMerchantAccount(Merchant) void
StartUp
StartUp()
main(String[]) void
startUp() void
CucumberTest
CucumberTest()
AccountServiceTestSteps
AccountServiceTestSteps()
testCustomer Customer
generatedMerchantId UUID
queue MessageQueue
testMerchant Merchant
generatedCustomerId UUID
accountService AccountService
merchantIsSuccessfullyDeregistered() void
setup() void
merchantRegistrationIsInitiated() void
merchantDeregistrationIsInitiated() void
aValidCustomer(String, String, String, String, String) void
aValidMerchant(String, String, String, String, String) void
aCustomerWithID(String) void
merchantIsSuccessfullyRegistered() void
customerIsSuccessfullyRegistered() void
aMerchantWithID(String) void
customerRegistrationIsInitiated() void
customerDeregistrationIsInitiated() void
AccountServiceTest
AccountServiceTest()
accountService AccountService
mockQueue MessageQueue
deregisterMerchant_ShouldPublishEventWithFalse() void
policyGetCustomerAccount_ShouldPublishAccountResponse() void
setUp() void
createMerchantAccount_ShouldAddMerchantAndPublishEvent() void
policyCustomerDeregistrationRequested_ShouldCallDeregisterCustomer(
policyValidateMerchant_ShouldPublishValidationResponse() void
getTokens_ShouldReturnCorrectTokenCount() void
policyCustomerTokensRequested_ShouldCallGetTokens() void
createAccount_ShouldAddCustomerAndPublishEvent() void
deregisterCustomer_ShouldPublishEventForNonExistentCustomer() oid
getTokens_ShouldPublishEventForNonExistentCustomer() void
policyMerchantDeregistrationRequested_ShouldCallDeregisterMerchant(
policyGetMerchantAccount_ShouldPublishAccountResponse() void
policyMerchantRegistrationRequested_ShouldCallCreateMerchantAccour
deregisterCustomer_ShouldRemoveCustomerAndPublishEvent() void
policyCustomerRegistrationRequested_ShouldCallCreateAccount() void
testMerchant
merchants
customers
testCustomer
accountService
1
*

Figure 4: Account Manager

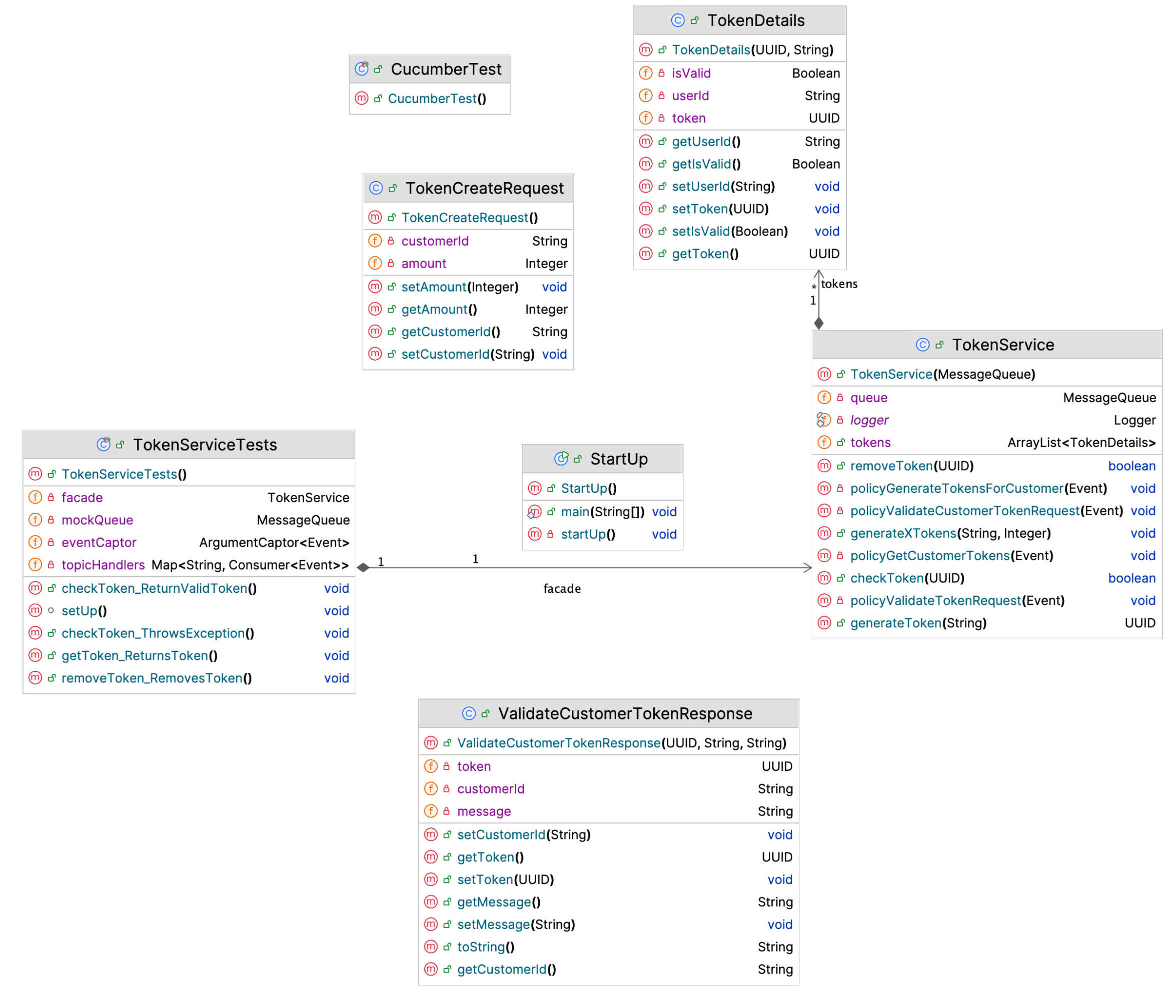

Figure 5: Token Manager

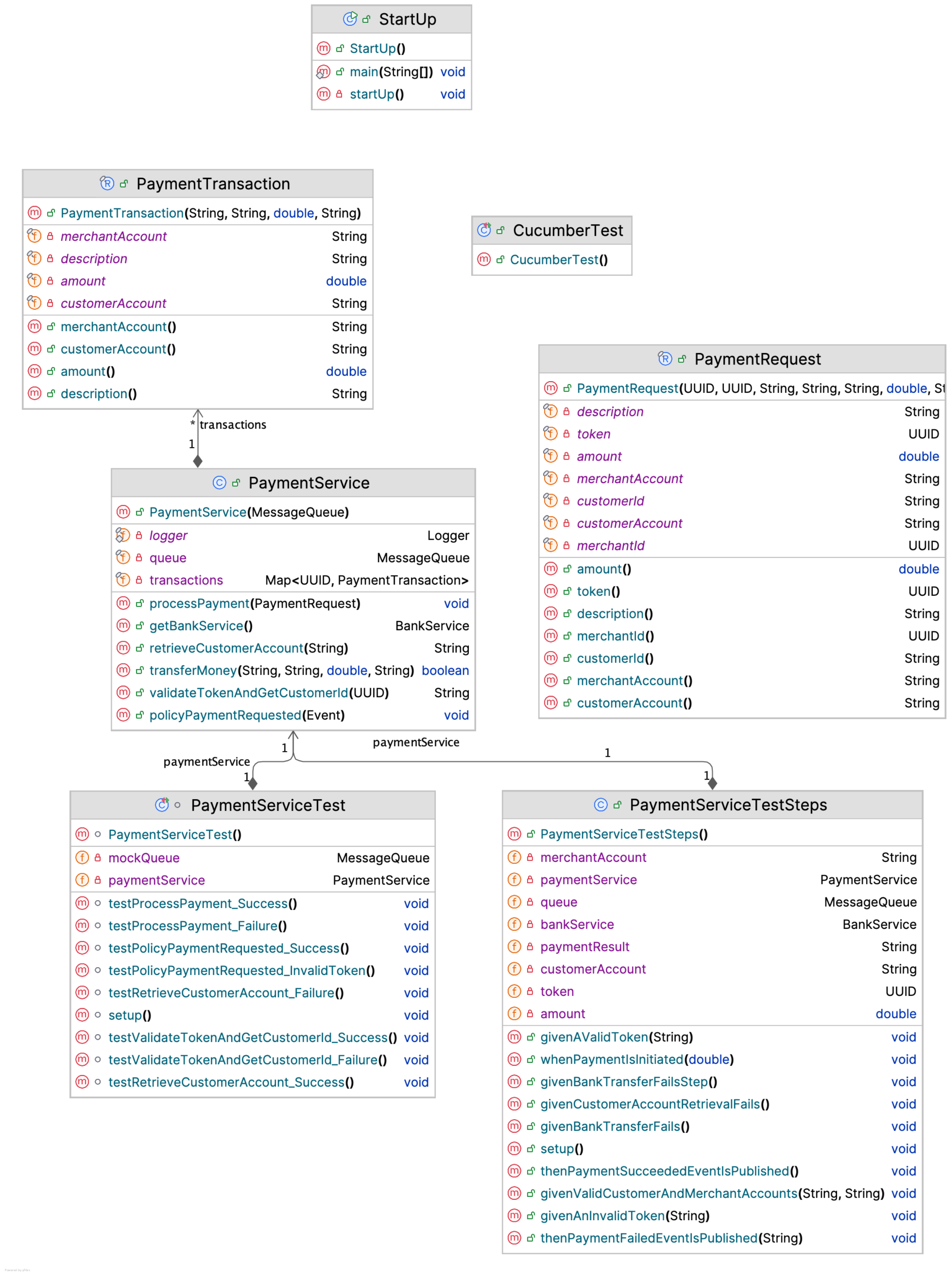
StartUp
StartUp()
main(String[]) void
startUp() void
PaymentTransaction
PaymentTransaction(String, String, double, String)
merchantAccount String
description String
amount double
customerAccount String
merchantAccount() String
customerAccount() String
amount() double
description() String
CucumberTest
CucumberTest()
PaymentRequest
PaymentRequest(UUID, UUID, String, String, String, double, St
description String
token UUID
amount double
merchantAccount String
customerId String
customerAccount String
merchantId UUID
amount() double
token() UUID
description() String
merchantId() UUID
customerId() String
merchantAccount() String
customerAccount() String
* transactions
1
PaymentService
PaymentService(MessageQueue)
logger Logger
queue MessageQueue
transactions Map<UUID, PaymentTransaction>
processPayment(PaymentRequest) void
getBankService() BankService
retrieveCustomerAccount(String) String
transferMoney(String, String, double, String) boolean
validateTokenAndGetCustomerId(UUID) String
policyPaymentRequested(Event) void
paymentService
1
paymentService
1
1
1
PaymentServiceTest
PaymentServiceTest()
mockQueue MessageQueue
paymentService PaymentService
testProcessPayment_Success() void
testProcessPayment_Failure() void
testPolicyPaymentRequested_Success() void
testPolicyPaymentRequested_InvalidToken() void
testRetrieveCustomerAccount_Failure() void
setup() void
testValidateTokenAndGetCustomerId_Success() void
testValidateTokenAndGetCustomerId_Failure() void
testRetrieveCustomerAccount_Success() void
PaymentServiceTestSteps
PaymentServiceTestSteps()
merchantAccount String
paymentService PaymentService
queue MessageQueue
bankService BankService
paymentResult String
customerAccount String
token UUID
amount double
givenAValidToken(String) void
whenPaymentIsInitiated(double) void
givenBankTransferFailsStep() void
givenCustomerAccountRetrievalFails() void
givenBankTransferFails() void
setup() void
thenPaymentSucceededEventIsPublished() void
givenValidCustomerAndMerchantAccounts(String, String) void
givenAnInvalidToken(String) void
thenPaymentFailedEventIsPublished(String) void

Figure 6: Payment Service

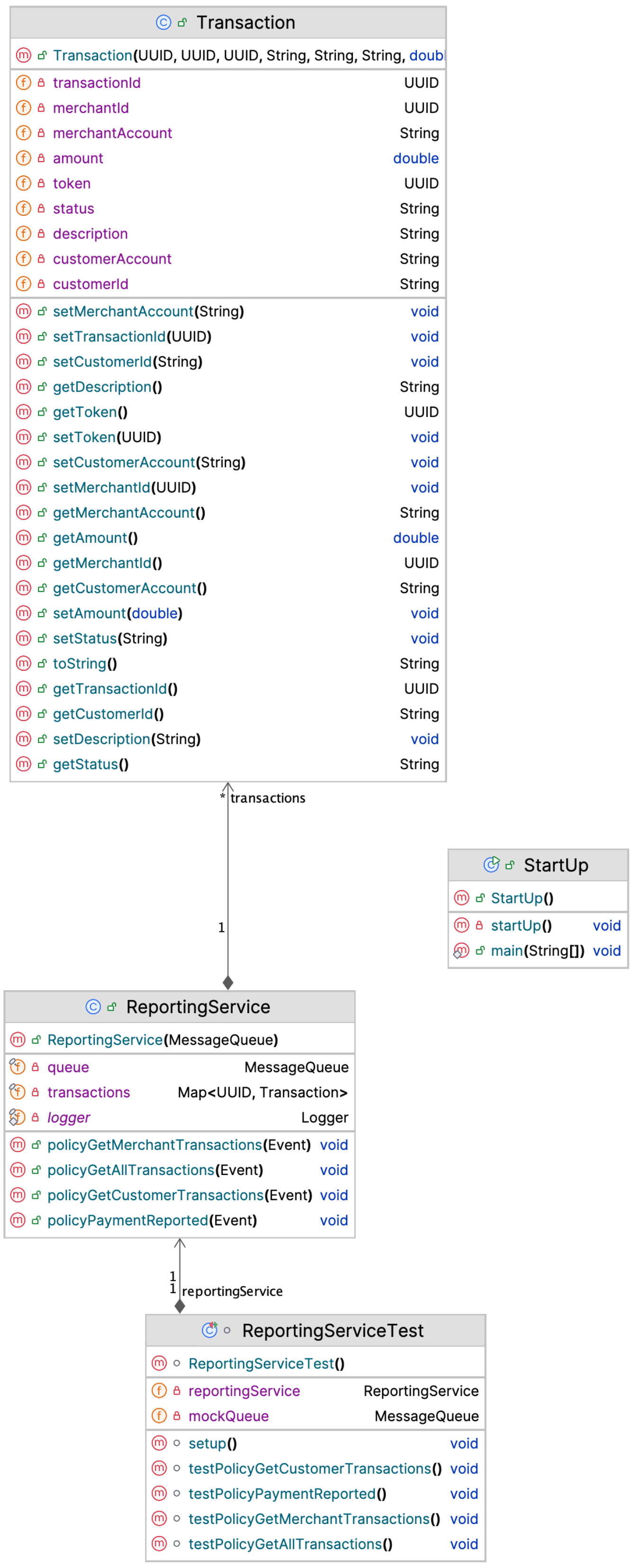
Transaction
Transaction(UUID, UUID, UUID, String, String, String, doubl
transactionId UUID
merchantId UUID
merchantAccount String
amount double
token UUID
status String
description String
customerAccount String
customerId String
setMerchantAccount(String) void
setTransactionId(UUID) void
setCustomerId(String) void
getDescription() String
getToken() UUID
setToken(UUID) void
setCustomerAccount(String) void
setMerchantId(UUID) void
getMerchantAccount() String
getAmount() double
getMerchantId() UUID
getCustomerAccount() String
setAmount(double) void
setStatus(String) void
toString() String
getTransactionId() UUID
getCustomerId() String
setDescription(String) void
getStatus() String
*
transactions
1
StartUp
StartUp()
startUp() void
main(String[]) void
ReportingService
ReportingService(MessageQueue)
queue MessageQueue
transactions Map<UUID, Transaction>
logger Logger
policyGetMerchantTransactions(Event) void
policyGetAllTransactions(Event) void
policyGetCustomerTransactions(Event) void
policyPaymentReported(Event) void
1
1
reportingService
ReportingServiceTest
ReportingServiceTest()
reportingService ReportingService
mockQueue MessageQueue
setup() void
testPolicyGetCustomerTransactions() void
testPolicyPaymentReported() void
testPolicyGetMerchantTransactions() void
testPolicyGetAllTransactions() void

Figure 7: Report Service

7 Major features/scenarios

DTU-Pay uses microservices to create an architecture that can easily updated possible to update its operations staying easy to update. The platform incorporates RabbitMQ for message queue handling while using Quarkus to develop efficient endpoints to support all essential system features. Below is a detailed description of the features implemented by each microservice:

7.1 Account Manager

The **Account Manager** leads all tasks for customer and merchant account management. The system enables complete account handling with RabbitMQ by controlling outfit new customers and merchants plus changing account data for asynchronous message transfer.

Account Creation The DTU Pay app starts the registration process when either a customer or merchant uses it. The application sends a `RegisterAccount` command to the Account Manager via the REST platform. Our system verifies customer authentication and publishes `CustomerRegistered` or `MerchantRegistered` events to RabbitMQ. Other services automatically update their databases. The Token Manager connects new user tokens with their accounts once registration completes successfully.

Account Request Handling The Account Manager handles requests to end customer or merchant accounts when they submit a `DeregisterAccount` message. After processing the Account Request, the Account Manager sends `CustomerDeregistered` or `MerchantDeregistered` messages to RabbitMQ. The microservices network receives notice from the Token Manager to stop issuing tokens or disable payments.

7.2 DTU Pay

As the central hub, **DTU Pay** coordinates the proper activity of each microservice within the system. DTU Pay uses RabbitMQ to connect both external bank APIs and internal services such as the Token Manager and Payment Service at once.

Plugin Functionality DTU Pay operates based on incoming events from the RabbitMQ messaging system. DTU Pay validates the token and processes the payment, then updates transaction records when the Merchant App sends a `PaymentRequest` event.

Transaction Handling When a `PaymentRequest` appears, DTU Pay receives data forwarded by the Merchant App, including token confirmation and payment details. The Payment Service triggers token validation in the Token Manager, which then starts the payment process. DTU Pay sends `PaymentSucceeded` or `PaymentFailed` event results to maintain transaction system accuracy.

7.3 Payment Service

The Payment Service handles the secure and reliable transaction processing between the customer and merchant using RabbitMq for messaging and SOAP APIs communicate with the bank. Its the crucial part of the DTU-pay system as its acts on the payment requests and make proper funds transfer possible.

7.3.1 Processing Payments

The payment service starts when the PaymnetRequest event is published by the DTU Pay to RabbitMQ message queue. This event have all the information like customer token, ID, Marchent ID, and payment amount.

- The payment service takes the validation of the token to token Manager via TokenValidation request. This checks that token is valid and unique.
- The service then talks to the bank's SOAP API to make sure that the customer's bank account exists and has enough funds. The Payment Service transfers funds from customer's account to the merchant's account if validation and fund check are successful.

- The service then calls to the bank's SOAP API, asks it if the customer's bank account exists and there is sufficient funds. The Payment Service assumes the funds transfer is successful if the account validation and fund check are completed, and publishes a PaymentProcessed event to RabbitMQ after completing the transaction. And this is an event that is consumed by the Reporting Service to update transaction logs and generate reports. This event is also used by the Account Manager to get the customer account balance and the current account balance to update the respective account balances (customer account – merchant account).

This structured process guarantees your transaction remains securely processed, validated across each step, and the whole system remains safely and modularly useable.

7.4 Token Manager

Token Manager ensures secure, privacy preserving unique transaction identifiers managing the life cycle of tokens. It issues tokens upon request, validates tokens on payments, and invalidates tokens when they are used.

7.4.1 Token Lifecycle:

When a customer has less than six unused tokens, customers can request new tokens. The Token Manager handles the following steps:

- **Token Generation:** When the system receives a token request it produces between one and five tokens that cannot be predicted and are secure and distinct. These tokens protect privacy because they do not share any customer-identifying details. After producing tokens the service records them in its database and sends a TokenGenerated notification to RabbitMQ for other platforms to receive.
- **Token Validation:** The Merchant App sends the payment request data to the Token Manager including the token. The Token Manager ensures the token meets three important requirements: it needs to be ready for use and linked to the customer making the payment. The Token Manager checks to verify only approved tokens from the right customer are processed.
- **Token Expiration:** When a token completes its payment purpose it becomes unusable. The system tags tokens as unusable once they are used for payments in order to protect security.

By managing the token lifecycle effectively, the Token Manager plays crucial role in the security and privacy of the payment system.

7.5 Reporting Service

The Reporting Service combines transaction records to create detailed reports for both customers and merchants. It shows everything clearly while reporting logs of transactions.

7.5.1 Customer and Merchant Reporting

- **Customer Reports:** Customers can obtain full transaction details by using the Customer API. The reports show everything about each transaction including payment information, merchant codes, and payment tokens. Through this process customers view complete transaction details of their payments.
- **Merchant Reports:** The system helps merchants write payment receipt details into one easy-to-read report. Each report shows how much customers paid but it provides no personal data to protect privacy. By monitoring their received payments merchants can view their profits without learning customer details.

After completing the transaction the reporting service sends updates to the RabbitMQ service for real-time report creation. System provides vital information to users so they can make smart choices but also keeps data protected.

8 User manual

DTU Pay can be started in the following way on Unix machines (MacOS/Linux):

1. Open Terminal
2. Execute the following: `./build_and_run.sh`

This command compiles the source code, builds Docker images and deploys Docker images, so that they run and DTU Pay is ready to receive API requests. To run JUnit and end-to-end Cucumber tests, do the following:

1. Open Terminal
2. Execute the following: `./run-tests.sh`
3. You can review the aggregated test report for all modules in *root_folder/target/surefire-reports/surefire.html*

In Windows, you can use Git Bash or Windows Subsystem for Linux (WSL) to execute the scripts. *include picture of all tests* For a more selective execution of the application, following scripts are provided:

1. `./compile.sh`: to install bank-client and rabbitmq and package all microservices.
2. `./build_images.sh`: builds Docker images of all microservices and stores them on the computer locally.
3. `./deploy.sh`: starts up rabbitmq and all microservices as Docker containers. DTU Pay can then be accessed via `localhost:8080`.
4. `./run_tests.sh`: runs all Cucumber and JUnit tests and produces surefire report in *target/surefire-reports/surefire.html*.
5. `./reset.sh`: a helper script that can be run before running `build_and_run.sh` to stop all Docker containers and remove all Docker images.

9 Installation Manual

The system can be installed and tested from the command line on any Linux computer without requiring Jenkins or access to a virtual machine. All shell scripts are provided with Unix-style line endings and are executable for the user.

Jenkins: http://fm-02.compute.dtu.dk:8282/

- Username: huba
- Password: huba975!

A top-level shell script is included to build, deploy, and run all tests, this can be executed with the command:

```
./build-and-run.sh
```

- The Git repository can be accessed via the URL, and the commit for the hand-in is the latest commit, and the user `huba` has been given access:
  - GitLab (gbar.dtu.dk): https://gitlab.gbar.dtu.dk/s137282/02267-sdws-dtu_pay
- **Detailed installation instructions:**
  1. **Prerequisites:** these versions of the tools work with our project. Detailed descriptions on how to install the individual tools are provided on their websites:
     - Java Development Kit (JDK) 21.0.6: Download from https://www.oracle.com/java/technologies/javase-jdk21-downloads.html
     - Apache Maven 3.9.9: Download from https://maven.apache.org/download.cgi

  - Docker 27.4.0: Download from https://www.docker.com/products/docker-desktop.

2. **Clone the Repository:** Next up you should clone the repository and enter the correct directory.

```
git clone https://gitlab.gbar.dtu.dk/s137282/02267-sdws-dtu_pay.git
cd 02267-sdws-dtu_pay
```

3. **Install Bank Client and RabbitMQ:** You then need to install the bank-client software as well as the RabbitMQ message broker.

   - **Manual Installation using Maven:**

   ```
   mvn clean install -f bank-client/pom.xml
   mvn clean install -f rabbitmq/pom.xml
   ```

   - **Automatic Installation using bash script:** If this option is chosen the next step can be skipped.

   ```
   ./compile.sh
   ```

4. **Build the Services:** next build our microservices using Maven or alternatively use the provided bash script.

   - **Manually build using Maven:**

   ```
   mvn package -f services/pom.xml
   ```

5. **Build the docker images:**

```
./build-images.sh
```

6. **Deploy the docker container for our tests:**

```
./deploy.sh
```

7. **Run All Tests:** our testing can be run by executing the provided shell script to build, deploy, and run tests:

```
./run_tests.sh
```

10 Collaboration

10.1 Report

Topic	Student
Introduction	Asher Sharif (s240193)
Collaboration	Jonas Puidokas (137282)
Event Storming	Jonas Puidokas (137282)
Continuous Integration & Delivery	Jonas Puidokas (137282)
Architecture	Anders Persson (233469)
Class diagrams	Jonas Puidokas (137282)
Major features/scenarios	Haissam Kamel (S233470) & Muhammad Zaid (S240177) & Bassel Beiroumi (194309)
Installation manual	Anders Persson (233469)
User manual	Jonas Puidokas (137282)

10.2 Diagrams

Topic	Student
Event storming	Jonas Puidokas (137282)
Architecture	Anders Persson (233469)
Class diagrams	Jonas Puidokas (137282)
Swagger/OpenAPI files	Asher Sharif (240193)

10.3 Systems

Topic	Student
Jenkins setup	Jonas Puidokas (137282)
Gitlab setup	Jonas Puidokas (137282)
Jenkins & Gitlab integration	Jonas Puidokas (137282)
CI/CD	Jonas Puidokas (137282)

10.4 Code

Topic	Student
Customer facade	Jonas Puidokas (137282)
Account manager	Jonas Puidokas (137282) & Asher Sharif (s240193)
Merchant facade	Asher Sharif (240193)
Payment service	Asher Sharif (240193)
Token manager	Bassel Beiroumi (194309) & Haissam Kamel Barghooth (s233470) & Anders Persson (s233469)
Report generator	Muhammad Zaid Mehmood (240177)
End to end testing client for merchant	Asher Sharif (s240193)