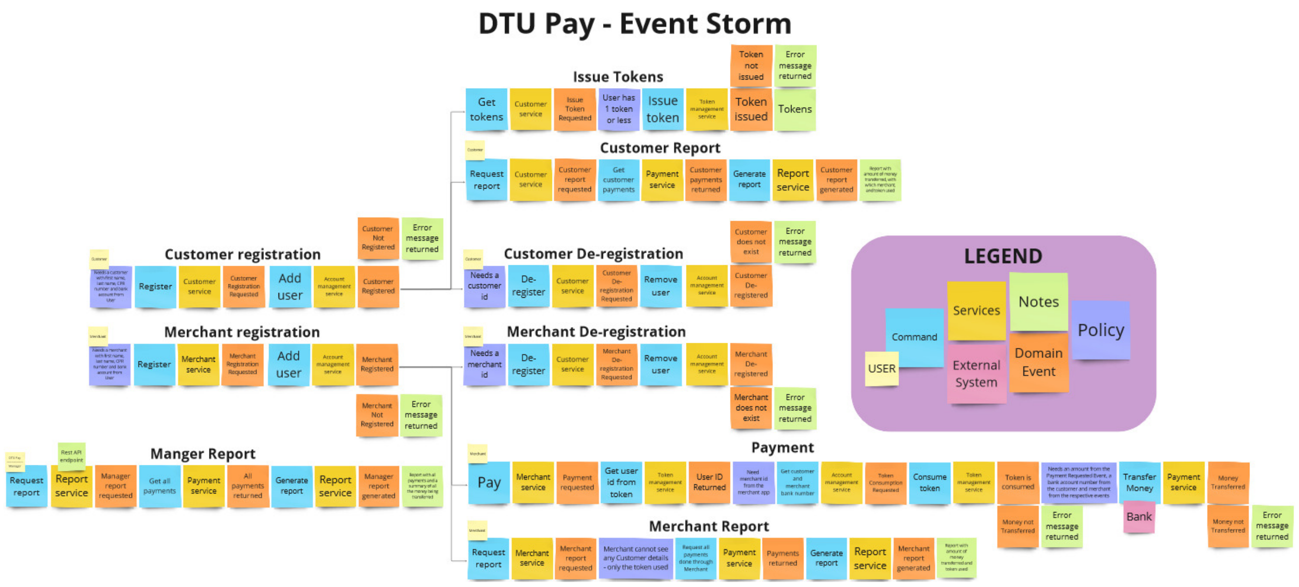
02267 Software Development of Web Services: DTU Pay Project Description

February 19, 2024

Contents

1 Event Storming

Here we present the final result of our event storming process. The diagram below proved to be very helpful throughout the entire development phase as we keep coming back to it for guidance on how to proceed.



2 Architecture

2.1 Microservices

Overall, our monolith contains 5 microservices internally, which will be presented in the following. The internal microservices communicate via message queues, while communication with the web service interface is REST and the communication with the Bank is using SOAP. The drawing below provides an overview of the architecture.

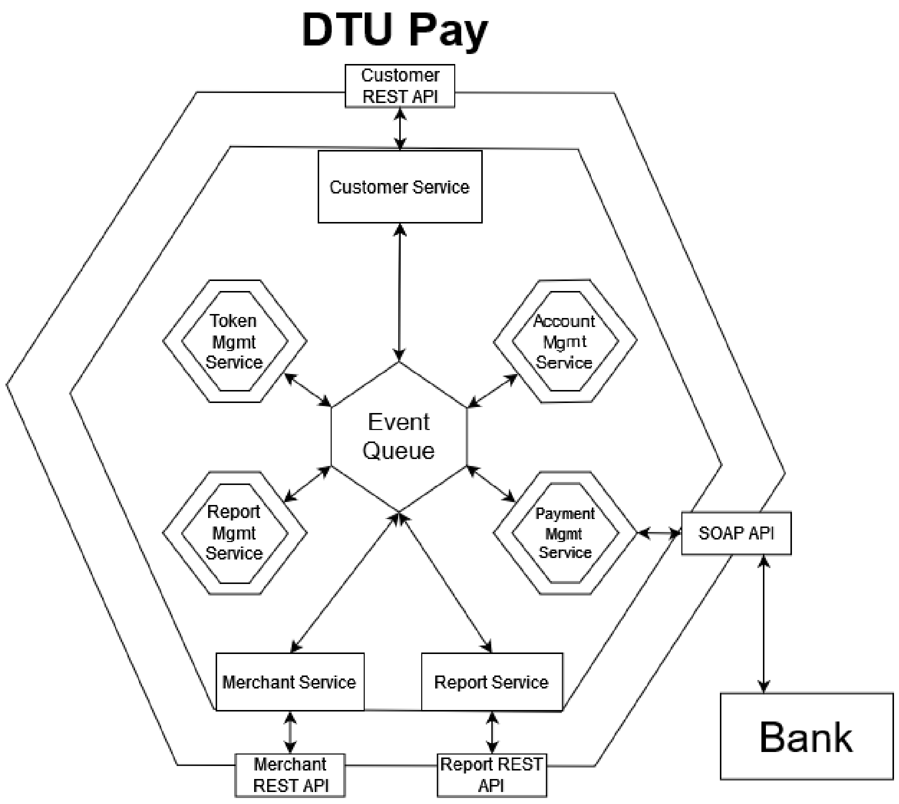


Figure 1: Architecture

2.1.1 Facade Microservice

The facade class diagrams show the classes that serve the external REST interface being provided through the resources CustomerResource, MerchantResource, and ManagerReportResource. Through the CustomerResource, the CustomerService is provided, and likewise, the MerchantResource provides the MerchantService. Finally, the ManagerReportResource provides the ManagerReportService. Moreover, the facade is familiar with the concept of a User as an abstract class, which the Merchant and Customer extend. The facade must also know the concept of a Token, as this is something a Customer can request. Finally, the facade knows the abstract class Report which CustomerReport, MerchantReport, and ManagerReport extend.

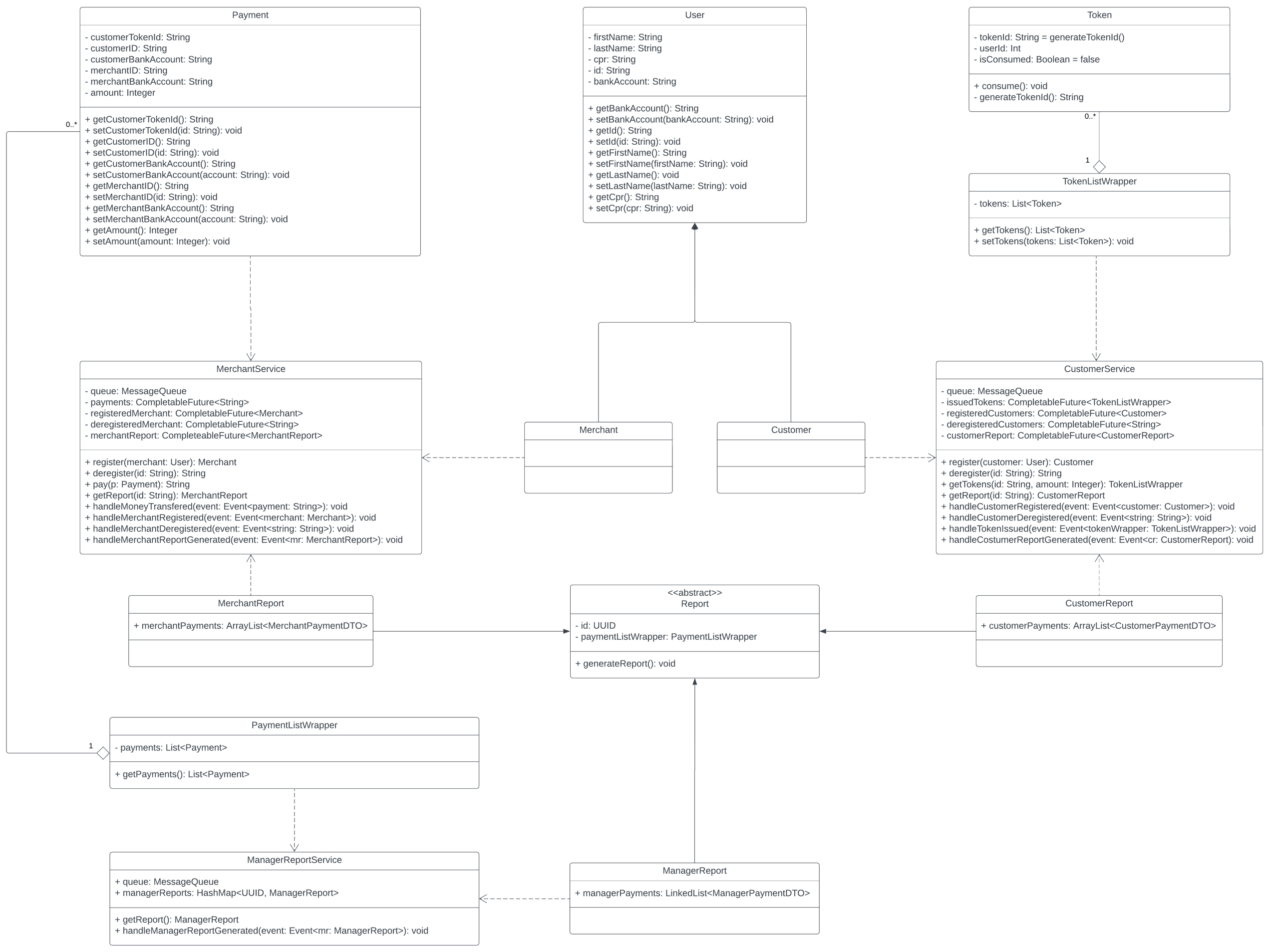

Payment
- customerTokenId: String
- customerID: String
- customerBankAccount: String
- merchantID: String
- merchantBankAccount: String
- amount: Integer
+ getCustomerTokenId(): String
+ setCustomerTokenId(id: String): void
+ getCustomerID(): String
+ setCustomerID(id: String): void
+ getCustomerBankAccount(): String
+ setCustomerBankAccount(account: String): void
+ getMerchantID(): String
+ setMerchantID(id: String): void
+ getMerchantBankAccount(): String
+ setMerchantBankAccount(account: String): void
+ getAmount(): Integer
+ setAmount(amount: Integer): void
User
- firstName: String
- lastName: String
- cpr: String
- id: String
- bankAccount: String
+ getBankAccount(): String
+ setBankAccount(bankAccount: String): void
+ getId(): String
+ setId(id: String): void
+ getFirstName(): String
+ setFirstName(firstName: String): void
+ getLastName(): void
+ setLastName(lastName: String): void
+ getCpr(): String
+ setCpr(cpr: String): void
Token
- tokenId: String = generateTokenId()
- userId: Int
- isConsumed: Boolean = false
+ consume(): void
- generateTokenId(): String
0..*
1
TokenListWrapper
- tokens: List<Token>
+ getTokens(): List<Token>
+ setTokens(tokens: List<Token>): void
0..*
MerchantService
- queue: MessageQueue
- payments: CompletableFuture<String>
- registeredMerchant: CompletableFuture<Merchant>
- deregisteredMerchant: CompletableFuture<String>
- merchantReport: CompleteableFuture<MerchantReport>
+ register(merchant: User): Merchant
+ deregister(id: String): String
+ pay(p: Payment): String
+ getReport(id: String): MerchantReport
+ handleMoneyTransfered(event: Event<payment: String>): void
+ handleMerchantRegistered(event: Event<merchant: Merchant>): void
+ handleMerchantDeregistered(event: Event<string: String>): void
+ handleMerchantReportGenerated(event: Event<mr: MerchantReport>): void
Merchant
Customer
CustomerService
- queue: MessageQueue
- issuedTokens: CompletableFuture<TokenListWrapper>
- registeredCustomers: CompletableFuture<Customer>
- deregisteredCustomers: CompletableFuture<String>
- customerReport: CompletableFuture<CustomerReport>
+ register(customer: User): Customer
+ deregister(id: String): String
+ getTokens(id: String, amount: Integer): TokenListWrapper
+ getReport(id: String): CustomerReport
+ handleCustomerRegistered(event: Event<customer: Customer>): void
+ handleCustomerDeregistered(event: Event<string: String>): void
+ handleTokenIssued(event: Event<tokenWrapper: TokenListWrapper>): void
+ handleCostumerReportGenerated(event: Event<cr: CustomerReport): void
MerchantReport
+ merchantPayments: ArrayList<MerchantPaymentDTO>
<<abstract>>
Report
- id: UUID
- paymentListWrapper: PaymentListWrapper
+ generateReport(): void
CustomerReport
+ customerPayments: ArrayList<CustomerPaymentDTO>
1
PaymentListWrapper
- payments: List<Payment>
+ getPayments(): List<Payment>
ManagerReportService
+ queue: MessageQueue
+ managerReports: HashMap<UUID, ManagerReport>
+ getReport(): ManagerReport
+ handleManagerReportGenerated(event: Event<mr: ManagerReport>): void
ManagerReport
+ managerPayments: LinkedList<ManagerPaymentDTO>

Figure 2: Facade

2.1.2 Report Microservice

The report class diagram shows how we have three different report classes. One for customers, one for merchants, and one for the bank manager. All of them inherit their functionality from the more general, abstract report class. These are then stored in our service class. As with all the other internal service classes communication is handled via message queues.

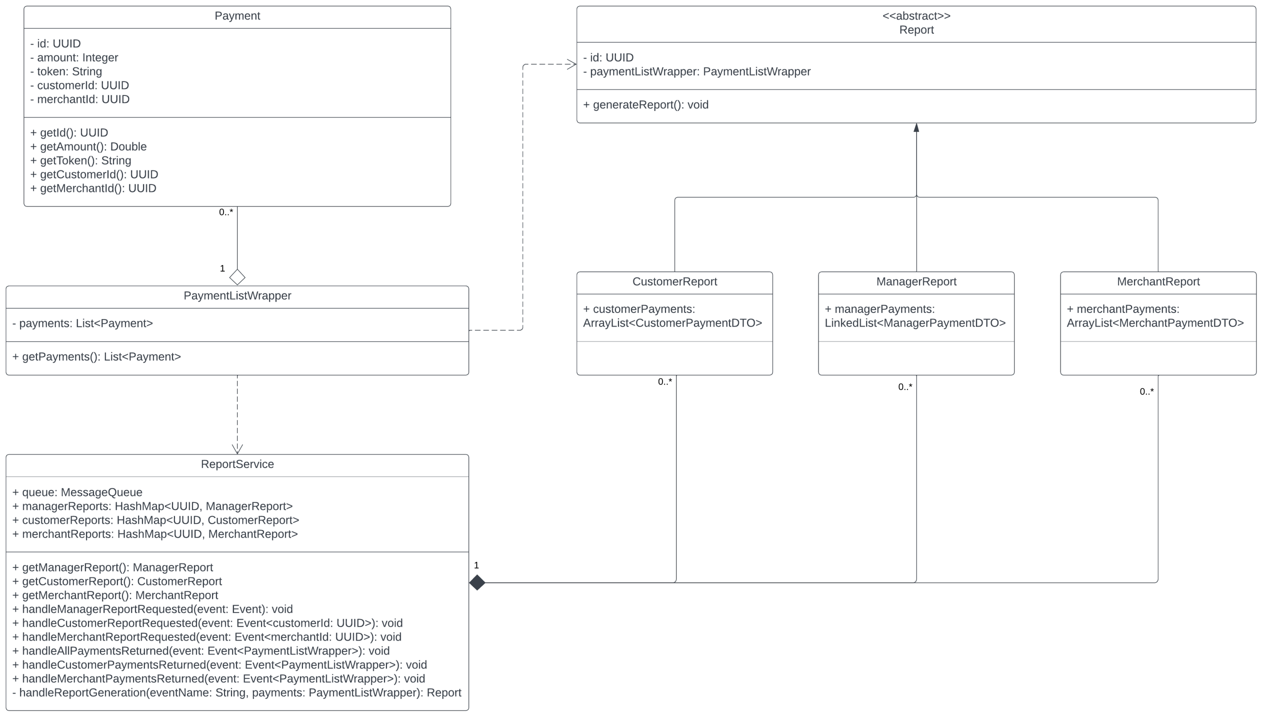

Payment
- id: UUID
- amount: Integer
- token: String
- customerId: UUID
- merchantId: UUID
+ getId(): UUID
+ getAmount(): Double
+ getToken(): String
+ getCustomerId(): UUID
+ getMerchantId(): UUID
0..*
1
PaymentListWrapper
- payments: List<Payment>
+ getPayments(): List<Payment>
<<abstract>>
Report
- id: UUID
- paymentListWrapper: PaymentListWrapper
+ generateReport(): void
CustomerReport
+ customerPayments: ArrayList<CustomerPaymentDTO>
ManagerReport
+ managerPayments: LinkedList<ManagerPaymentDTO>
MerchantReport
+ merchantPayments: ArrayList<MerchantPaymentDTO>
0..*
0..*
0..*
ReportService
+ queue: MessageQueue
+ managerReports: HashMap<UUID, ManagerReport>
+ customerReports: HashMap<UUID, CustomerReport>
+ merchantReports: HashMap<UUID, MerchantReport>
+ getManagerReport(): ManagerReport
+ getCustomerReport(): CustomerReport
+ getMerchantReport(): MerchantReport
+ handleManagerReportRequested(event: Event): void
+ handleCustomerReportRequested(event: Event<customerId: UUID>): void
+ handleMerchantReportRequested(event: Event<merchantId: UUID>): void
+ handleAllPaymentsReturned(event: Event<PaymentListWrapper>): void
+ handleCustomerPaymentsReturned(event: Event<PaymentListWrapper>): void
+ handleMerchantPaymentsReturned(event: Event<PaymentListWrapper>): void
- handleReportGeneration(eventName: String, payments: PaymentListWrapper): Report
1

Figure 3: Report Management Service

2.1.3 Token Microservice

It is the token microservice's responsibility to handle everything that is related to tokens. The TokenService class has a TokenManagement instance in which all token instances are stored in a map. This enables the token management to have methods like generating and consuming tokens, but also retrieving the ID of the user (customer) to which the token belongs.

The token class itself is quite simple, it has three fields, the tokenId, the userId, and the flag isConsumed which tells whether it has been consumed or not. When a token is initialized, the generateTokenId method is invoked, which creates the id as a string representation of a UUID. This implementation ensures that all token IDs are unique and that they cannot be tampered with. To consume a token the public consume method has to be called.

As the only thing the token service needs to know about users are their IDs, it is likewise the only field in the service's user class.

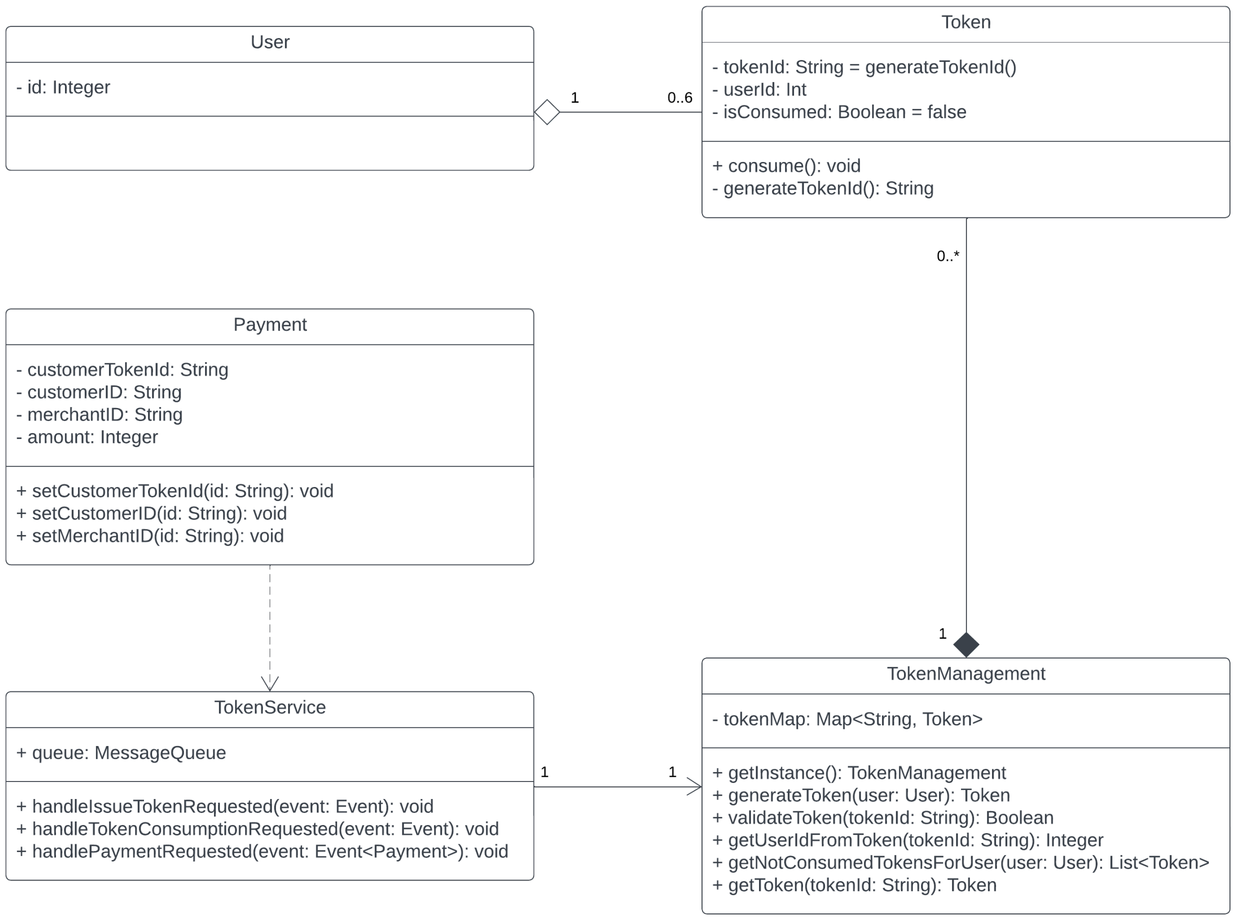
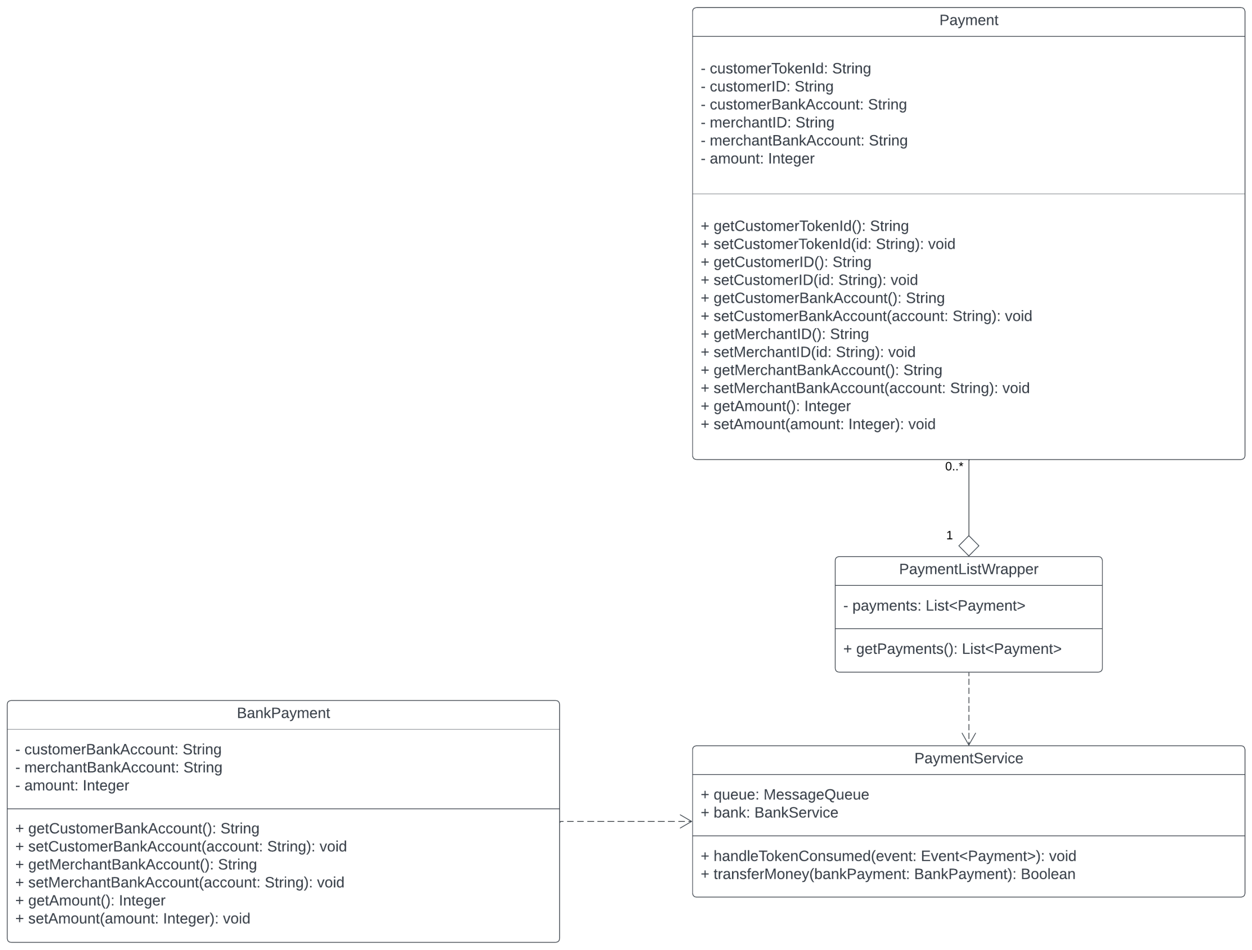

User
- id: Integer
1
0..6
Token
- tokenId: String = generateTokenId()
- userId: Int
- isConsumed: Boolean = false
+ consume(): void
- generateTokenId(): String
0..*
Payment
- customerTokenId: String
- customerID: String
- merchantID: String
- amount: Integer
+ setCustomerTokenId(id: String): void
+ setCustomerID(id: String): void
+ setMerchantID(id: String): void
1
TokenService
+ queue: MessageQueue
+ handleIssueTokenRequested(event: Event): void
+ handleTokenConsumptionRequested(event: Event): void
+ handlePaymentRequested(event: Event<Payment>): void
1
1
TokenManagement
- tokenMap: Map<String, Token>
+ getInstance(): TokenManagement
+ generateToken(user: User): Token
+ validateToken(tokenId: String): Boolean
+ getUserIdFromToken(tokenId: String): Integer
+ getNotConsumedTokensForUser(user: User): List<Token>
+ getToken(tokenId: String): Token

Figure 4: Token Management Service

2.1.4 Payment Microservice

The payment class diagram shows that the account management service also must be familiar with the concept of the abstract User class, which the Merchant and Customer classes extend. The payment microservice is responsible for the SOAP communication with the Bank. In addition, this service is also responsible for storing all payments made and fetching and forwarding the correct ones to the report service when needed. payments are stored in a list of Payment objects which implies that payment service must know a Payment class.



Payment
- customerTokenId: String
- customerID: String
- customerBankAccount: String
- merchantID: String
- merchantBankAccount: String
- amount: Integer
+ getCustomerTokenId(): String
+ setCustomerTokenId(id: String): void
+ getCustomerID(): String
+ setCustomerID(id: String): void
+ getCustomerBankAccount(): String
+ setCustomerBankAccount(account: String): void
+ getMerchantID(): String
+ setMerchantID(id: String): void
+ getMerchantBankAccount(): String
+ setMerchantBankAccount(account: String): void
+ getAmount(): Integer
+ setAmount(amount: Integer): void
0..*
1
PaymentListWrapper
- payments: List<Payment>
+ getPayments(): List<Payment>
BankPayment
- customerBankAccount: String
- merchantBankAccount: String
- amount: Integer
+ getCustomerBankAccount(): String
+ setCustomerBankAccount(account: String): void
+ getMerchantBankAccount(): String
+ setMerchantBankAccount(account: String): void
+ getAmount(): Integer
+ setAmount(amount: Integer): void
PaymentService
+ queue: MessageQueue
+ bank: BankService
+ handleTokenConsumed(event: Event<Payment>): void
+ transferMoney(bankPayment: BankPayment): Boolean

Figure 5: Payment Management Service

2.1.5 Account Management Microservice

The account management class diagram shows that the account management service also must be familiar with the concept of the abstract User class, which Merchant and Customer extends. As this service is responsible for managing users registered to DTU Pay, the representation of a User must contain more information, than for instance, the token management service's representation of a User. Likewise, the account management service does not need to know about any token-related information, as this is the responsibility of the token management service. Finally, the account management service must know what a Payment is, as that holds essential information to fetch the relevant user and their bank accounts.

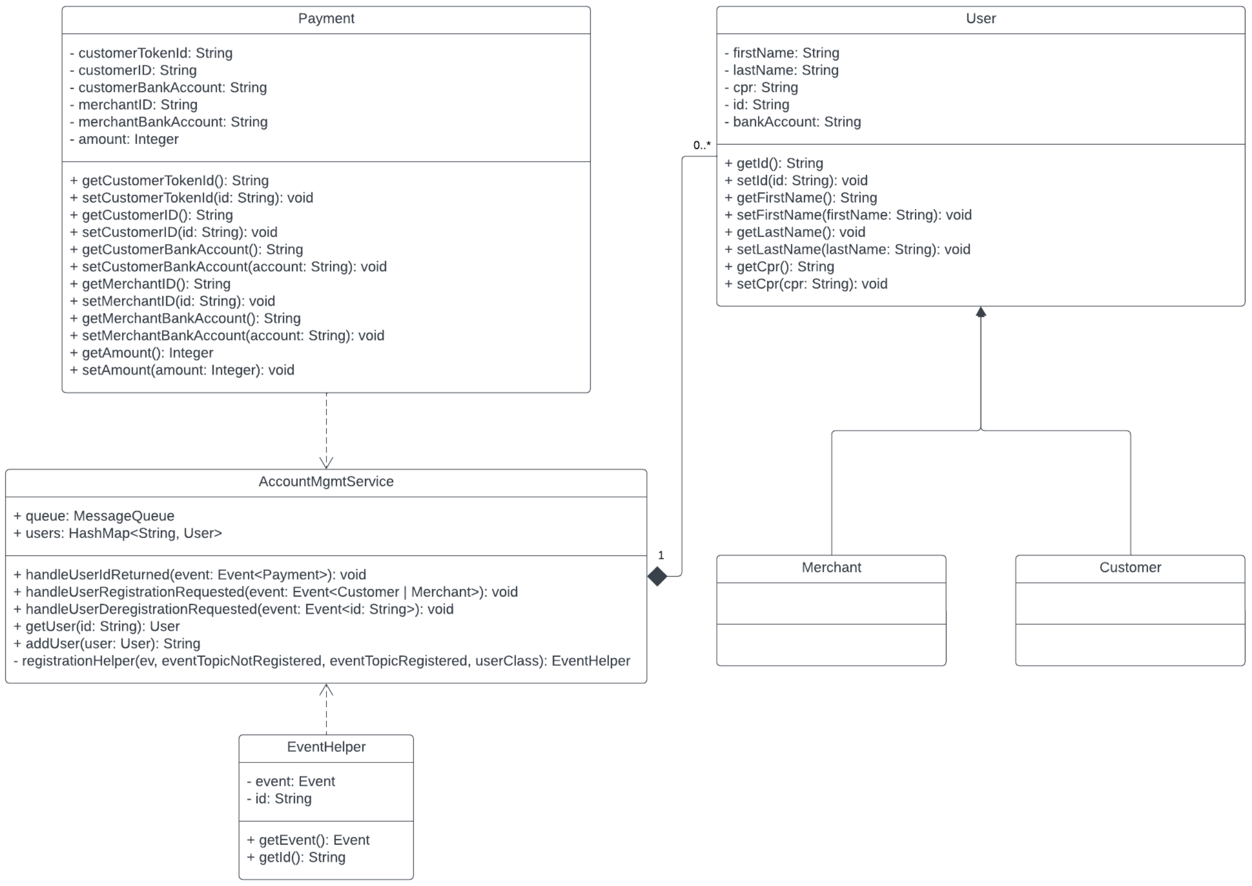

Payment
- customerTokenId: String
- customerID: String
- customerBankAccount: String
- merchantID: String
- merchantBankAccount: String
- amount: Integer
+ getCustomerTokenId(): String
+ setCustomerTokenId(id: String): void
+ getCustomerID(): String
+ setCustomerID(id: String): void
+ getCustomerBankAccount(): String
+ setCustomerBankAccount(account: String): void
+ getMerchantID(): String
+ setMerchantID(id: String): void
+ getMerchantBankAccount(): String
+ setMerchantBankAccount(account: String): void
+ getAmount(): Integer
+ setAmount(amount: Integer): void
User
- firstName: String
- lastName: String
- cpr: String
- id: String
- bankAccount: String
+ getId(): String
+ setId(id: String): void
+ getFirstName(): String
+ setFirstName(firstName: String): void
+ getLastName(): void
+ setLastName(lastName: String): void
+ getCpr(): String
+ setCpr(cpr: String): void
0..*
1
AccountMgmtService
+ queue: MessageQueue
+ users: HashMap<String, User>
+ handleUserIdReturned(event: Event<Payment>): void
+ handleUserRegistrationRequested(event: Event<Customer | Merchant>): void
+ handleUserDeregistrationRequested(event: Event<id: String>): void
+ getUser(id: String): User
+ addUser(user: User): String
- registrationHelper(ev, eventTopicNotRegistered, eventTopicRegistered, userClass): EventHelper
Merchant
Customer
EventHelper
- event: Event
- id: String
+ getEvent(): Event
+ getId(): String

Figure 6: Account Management Service

2.2 End-to-end Test client

This is the class diagram for the client responsible for conducting the end-to-end tests. It only communicates with DTU Pay through the REST endpoints via. its various service classes. To be able to do proper testing the client needs to know the Payment, User, Customer, Merchant, and Token classes.

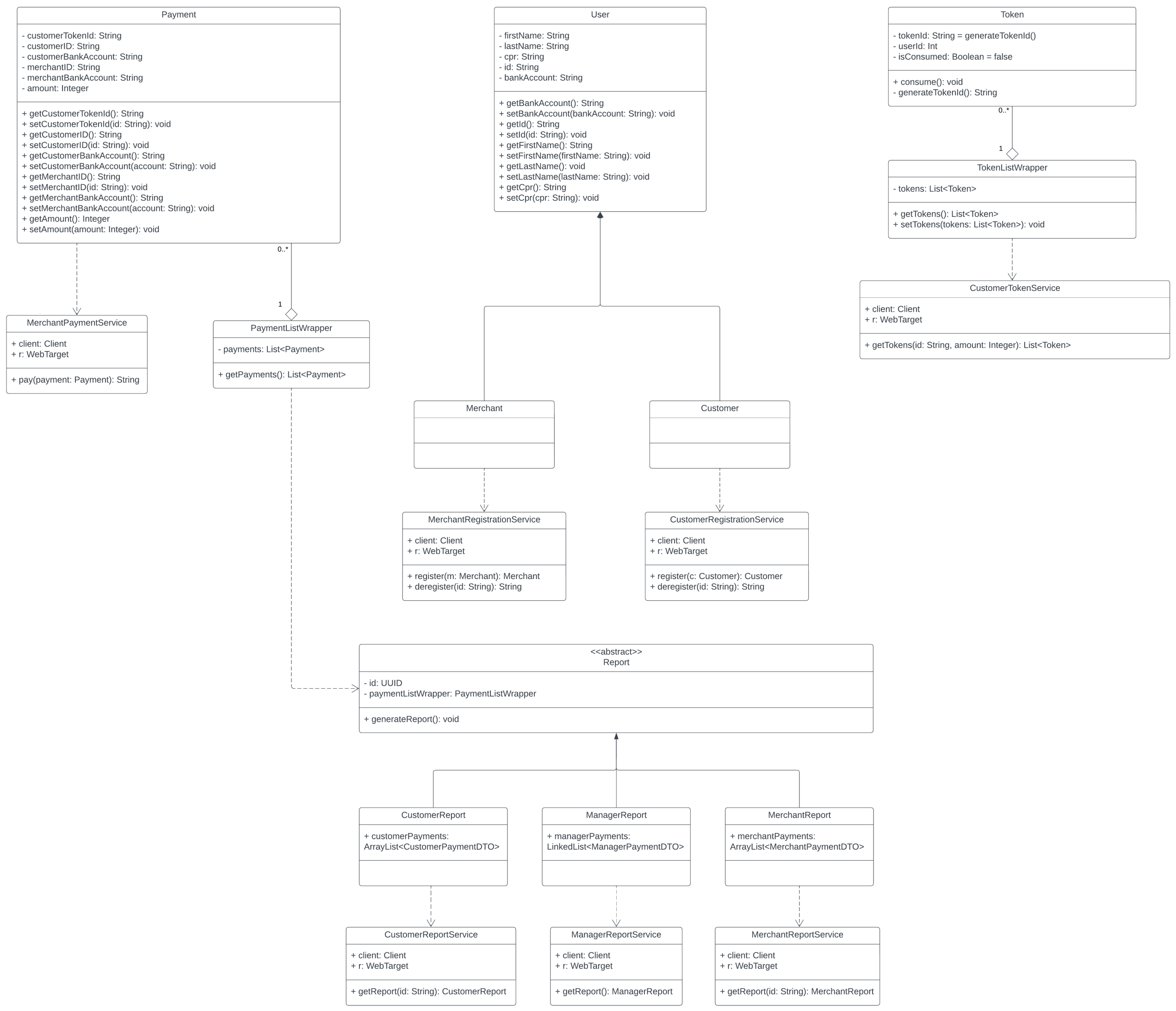

Payment
- customerTokenId: String
- customerID: String
- customerBankAccount: String
- merchantID: String
- merchantBankAccount: String
- amount: Integer
+ getCustomerTokenId(): String
+ setCustomerTokenId(id: String): void
+ getCustomerID(): String
+ setCustomerID(id: String): void
+ getCustomerBankAccount(): String
+ setCustomerBankAccount(account: String): void
+ getMerchantID(): String
+ setMerchantID(id: String): void
+ getMerchantBankAccount(): String
+ setMerchantBankAccount(account: String): void
+ getAmount(): Integer
+ setAmount(amount: Integer): void
0..*
1
MerchantPaymentService
+ client: Client
+ r: WebTarget
+ pay(payment: Payment): String
PaymentListWrapper
- payments: List<Payment>
+ getPayments(): List<Payment>
User
- firstName: String
- lastName: String
- cpr: String
- id: String
- bankAccount: String
+ getBankAccount(): String
+ setBankAccount(bankAccount: String): void
+ getId(): String
+ setId(id: String): void
+ getFirstName(): String
+ setFirstName(firstName: String): void
+ getLastName(): void
+ setLastName(lastName: String): void
+ getCpr(): String
+ setCpr(cpr: String): void
Merchant
Customer
MerchantRegistrationService
+ client: Client
+ r: WebTarget
+ register(m: Merchant): Merchant
+ deregister(id: String): String
CustomerRegistrationService
+ client: Client
+ r: WebTarget
+ register(c: Customer): Customer
+ deregister(id: String): String
Token
- tokenId: String = generateTokenId()
- userId: Int
- isConsumed: Boolean = false
+ consume(): void
- generateTokenId(): String
0..*
1
TokenListWrapper
- tokens: List<Token>
+ getTokens(): List<Token>
+ setTokens(tokens: List<Token>): void
CustomerTokenService
+ client: Client
+ r: WebTarget
+ getTokens(id: String, amount: Integer): List<Token>
<<abstract>>
Report
- id: UUID
- paymentListWrapper: PaymentListWrapper
+ generateReport(): void
CustomerReport
+ customerPayments:
ArrayList<CustomerPaymentDTO>
ManagerReport
+ managerPayments:
LinkedList<ManagerPaymentDTO>
MerchantReport
+ merchantPayments:
ArrayList<MerchantPaymentDTO>
CustomerReportService
+ client: Client
+ r: WebTarget
+ getReport(id: String): CustomerReport
ManagerReportService
+ client: Client
+ r: WebTarget
+ getReport(): ManagerReport
MerchantReportService
+ client: Client
+ r: WebTarget
+ getReport(id: String): MerchantReport

Figure 7: Test Client

2.3 REST interfaces

This section presents an overview of the available REST interfaces, which HTTP method to be used, and their functionality.

2.3.1 /customer

URI Path = Resource	HTTP Method	Function
/customer	POST	Register as customer
/customer?id=<id>	DELETE	Deregister as customer
/customer/report?id=<id>	GET	Get report as customer
/customer/tokens?id=<id>&amount=<amount>	GET	Get tokens

Table 1: Customer Interface

2.3.2 /merchant

URI Path = Resource	HTTP Method	Function
/merchant	POST	Register as merchant
/merchant	DELETE	Deregister as merchant
/merchant/payment	POST	Issue a payment
/merchant/report?id=<id>	GET	Get report as merchant

Table 2: Merchant Interface

2.3.3 /report

URI Path = Resource	HTTP Method	Function
/report	GET	Get all reports

Table 3: Report Interface

2.3.4 Design decisions

When formulating the REST interfaces, several design decisions were made to comply with the best practices for implementing a REST API. An example hereof is the principle of avoiding the use of verbs in URI paths and also maintaining a clear separation between the distinct interfaces.

The tables above provide details for the specific `/customer`, `/merchant`, and `/report` interfaces. Each of these interfaces is associated with a set of URI paths, HTTP methods, and the described functions. The URI paths are aligned with the resources they represent, just as the HTTP methods reflect their intended operations on said resources.

These decisions align with the best practices and aim to improve the robustness and maintainability of the system. Some of the key decisions are the following:

Registration error handling When attempting to register with an already registered CPR number, the system is designed to respond with a `400 Bad Request` status code. For merchants, the error message is specified as "Merchant already exists" while for customers, it is "Customer already exists". This approach ensures clear feedback to clients attempting registration.

Deregistration error handling When attempting to deregister with an ID that is not known, a `404 Not Found` status code will be received. In the case of a merchant, the error message is "Merchant does not exist", while for customers, it is “Customer does not exist”. Thereby, the client is receiving an appropriate status code while also receiving an informative error message.

Payment error handling If a payment is issued by a merchant, and the customer does not have a sufficient amount of money in the bank account, a `400 Bad Request` status code will be returned with the error message being “Unsuccessful payment”.

Empty report In case of requesting a report that holds no payments, a `204 No Content` status code will be returned to indicate to the client that the request was successful, however, it holds no content.

By incorporating these design decisions, the REST interfaces are not only aligned with best practices but also reflect a commitment to providing a user-friendly, and robust API for both customers and merchants. It is assessed that these decisions contribute to the overall effectiveness and reliability of the system.

2.4 Communication between microservices

DTU Pay works with 3 different communication methods. The first is REST, which is used by the Facade to expose the web service endpoints. Secondly, the communication between the payment service and the bank is done with SOAP, as this is the requirement from the bank side. Finally, all internal communication between the microservices is done via. message queues. A more collaborative description of these communications channels is described in section 3.

3 Major features/scenarios

In this section, we will present the main features and scenarios of DTU Pay. Other scenarios have been implemented as well and we refer to the source code for a more comprehensive list of implemented scenarios.

3.1 Payment to the bank

A payment is initiated by a merchant submitting a POST request to the REST API endpoint `/merchant/payment`, containing a `Payment` object with the merchant’s ID, a token from a customer, and the amount to be transferred from the customer to the merchant. This will trigger the `MerchantService` to publish a `PaymentRequested` message queue event containing the token, ID, and amount, which will be handled by the `TokenService`. The `TokenService` knows the customer associated with the token. Given that the token has not previously been consumed, the service will fetch the corresponding customer ID, add it to the `Payment`, and publish a `UserIDReturned` messages queue event containing the new `Payment`. The `AccountMgmtService` will handle the `UserIDReturned` event by adding the merchant’s and customer’s bank accounts to the `Payment`, and then publish a `TokenConsumptionRequested` message queue event. Now the `TokenService` will handle this event, by consuming the token and publishing the `TokenIsConsumed` event, which is picked up by the `PaymentService`. The `PaymentService` is then responsible for the SOAP communication with the bank. If the payment is successful, this is added to a list of payments that can then be used for reporting later, and the event `MoneyTransferred` is published containing a notification that the payment was successful. Finally, the `MerchantService` will handle the `MoneyTransferred` event and return the success message to the REST API and thus to the merchant.

This long chain of events is visualized in the event-storming diagram. Below is shown the end-to-end scenario for a successful payment and the scenario for an unsuccessful payment due to insufficient funds.

```
Scenario: Successful payment
Given the customer with a bank account with balance 1000
```

```
And a customer is registered with DTU Pay
And the merchant with a bank account with balance 1000
And a merchant is registered with DTU Pay
And the customer requests 5 tokens
When the customer has handed a token to the merchant
And the merchant initiates a payment of 100
Then the payment is successful
And the balance of the customer is 900
And the balance of the merchant is 1100

Scenario: Unsuccessful payment due to insufficient amount of money
Given the customer with a bank account with balance 0
And a customer is registered with DTU Pay
And the merchant with a bank account with balance 1000
And a merchant is registered with DTU Pay
And the customer requests 5 tokens
When the customer has handed a token to the merchant
And the merchant initiates a payment of 100
Then the payment is unsuccessful
And the balance of the customer is 0
And the balance of the merchant is 1000
```

3.2 Register and de-register users

Both merchants and customers can register and de-register with the only difference being that a merchant will initiate this from the `/merchant` REST API endpoint and the customers from the `/customer` endpoint. In the following, we will therefore only describe customer registration and de-registration.

Registration: When a customer wants to register with DTU Pay, the customer will submit a POST request to the `/customer` endpoint, containing a `Customer` with a first and last name, a CPR number, and a bank account. This will trigger the register function of the `CustomerService`. In the corresponding service, a `CustomerRegistrationRequested` message queue event will be published, carrying the `Customer` object. This event is handled by the `AccountMgmtService` which will check if a customer with an identical CPR number is already registered. If that is not the case, the `Customer` is assigned an ID, and a `CustomerRegistered` message queue event carrying the customer's ID is published. The Customer object is returned to the customer after the `CustomerService` handles the `CustomerRegistered` event.

De-registration: When a customer wants to de-register with DTU Pay, the customer will submit a DELETE request to the `/customer` endpoint, containing the customer's ID as a query parameter. This will trigger the de-register function of the `CustomerService` and a `CustomerDeregistrationRequested` message queue event will then be published, carrying the customer's ID. This event is handled by the `AccountMgmtService` which will check if a customer with that ID is indeed registered. If that is the case, the `Customer` is de-registered, and a `CustomerDeregistered` message queue event carrying the successful messages is published. The successful messages is returned to the customer after the `CustomerService` handles the `CustomerDeregistered` event.

Below is shown the end-to-end scenario for a successful registration and a successful de-registration of a customer.

```
Scenario: Customer registration
Given there is a customer with empty id
```

```
When the customer is being registered
Then the "CustomerRegistrationRequested" event is sent
When the "CustomerRegistered" registration event is sent with non-empty id
Then the customer is registered and his id is set

Scenario: Customer de-registration
Given there is a customer with id "123"
When the customer is being de-registered
Then the "CustomerDeregistrationRequested" event is sent
When the "CustomerDeregistered" deregistration event is sent
Then the customer is de-registered
```

3.3 Reports

Reports can be generated in three ways, each way returning a different kind of report. The available types of reports are `CustomerReport`, `MerchantReport`, and `ManagerReport`. A `CustomerReport` contains all the payments associated with a given customer. The payments will contain information on the merchant's ID, the token used and the amount transferred. A `MerchantReport` contains all the payments associated with a given merchant. The payments will contain information on the token used and the amount transferred. Finally, `ManagerReport` contains all the payments made and the payments will contain information on the merchant's ID, the customer's ID, the token used, and the amount transferred.

The retrieval of reports is initiated at the following endpoint with a GET request: `/customer/report` for `CustomerReports`, the GET request must have the ID of a customer as the query parameter. `/merchant/report` for `MerchantReports`, here the query parameter should be an ID of a merchant. Finally, `/report` for `ManagerReports`, this request requires no query parameter. The process of how a report is obtained is very similar for all three cases and we will therefore only describe the retrieval of a `MerchantReport` here.

Once the merchant has submitted a GET request to `/merchant/report` with his DTU Pay ID, the `getReport` function of the `MerchantService` is called. Then a `MerchantReportRequested` message queue event is published with the merchant's ID. The `PaymentMgmtService` will then handle the `MerchantReportRequested` by extracting all payments related to the merchant and publishing these in a `MerchantPaymentsReturned` event. Now the `ReportService` takes over, handling the event by generating a `MerchantReport` and publishing it with a `MerchantReportGenerated` event. Finally, once the `MerchantService` receives and handles this event, the report will be returned to the merchant.

Below is shown the end-to-end scenario for a successful `MerchantReport` retrieval.

```
Scenario: Merchant ask for report successful
Given the customer with a bank account with balance 1000
And a customer is registered with DTU Pay
And the merchant with a bank account with balance 1000
And a merchant is registered with DTU Pay
And the customer requests 5 tokens
When the customer has handed a token to the merchant
And the merchant initiates a payment of 100
Then the payment is successful
And the balance of the customer is 900
And the balance of the merchant is 1100
When the merchant ask for a report
```

```
Then a report is returned to the merchant
And the report contains the payment
```

3.4 Issuing new tokens

Once in a while the customer will run out of tokens and need new ones. To obtain new tokens, the customer will have to submit a GET request to `/customer/tokens`. The GET request takes two query parameters, namely a customer ID and the amount of new token requested. The GET request will trigger the `getTokens` function at the `CustomerService` which will then forward the ID and amount in a `IssueTokenRequested` message queue event. This event is handled by the `TokenMgmtService` which will check if the customer is allowed to get the requested amounts of tokens. If this is the case, then all tokens belonging to the customer are returned to the `CustomerService` by a `TokenIssued` event and then returned to the customer.

Below is shown the end-to-end scenario for a successful request of 5 `Token` retrieval and a fail scenario.

```
Scenario: Customer requests tokens
Given a customer is registered with DTU Pay
And the customer has 1 token
When the customer requests 5 tokens
Then the customer has 6 tokens
```

```
Scenario: Customer requests tokens fail
Given a customer is registered with DTU Pay
And the customer has 2 token
When the customer requests 5 tokens
Then the customer has 2 token
```