Projekt beskrivelse

2024-09-15, multilingualism

Denne rapport beskrivelse findes også som markdown i stedet for PDF.

Arkitektur

Dette projekt består af 5 microservices.

1. SimpleDTUPay
2. RabbitMQ
3. Payment
4. Bank
5. Account

SimpleDTUPay er facaden for hele systemet og står for at modtage og videregive instruktioner som en bruger beder om. Via RESTful API endpoints kan brugere interagere med systemet. SimpleDTUPay er forbundet med de andre microservices igennem RabbitMQ. SimpleDTUPay udsender events og lytter på andre events igennem RabbitMQ. En af de services som SimpleDTUPay er i kontakt med er Payment microservicen. Denne service står for at håndtere al der vedrører betaling. Payment har en in-memory liste over alle forrige betalinger og er i direkte kontakt via HTTP REST kald til Banken. Den er også i kontakt med Account igennem RabbitMQ events som beskrevet før. Account microservicen har til ansvar at beholde en liste over alle registrerede brugere. Eksempelvis hvis Payment servicen vil have et Bank konto ID går det igennem Account servicen.

SimpleDTUPay API

I roden af projektets zip-fil findes en OpenAPI specifikation for alle endpoints i SimpleDTUPay. Alle endpoints kan også udforskes ved at køre projektet i dev mode og navigere til `http://localhost:8080/q/swagger-ui/`.

Der er udarbejdet 3 unikke endpoints. Via `/customer` kan man oprette en ny customer via POST metoden, samt en angivelse af fornavn, efternavn, cpr og bankkontoid. Det kræver altså at man allerede er oprettet hos banken at blive oprettet i dette sytem.

`/merchants` fungerer ligeså, men med merchants.

`/payments` har to metoder, nemlig GET og POST. GET returnerer alle betalinger der er i systemet i en liste og POST opretter en ny betaling. POST metoden kræver også at man medsender en betalingsmængde, samt et customerid og merchantid. Det skal understreges at customerid og merchantid IKKE er ækvivalent med bankkontoid. Systemet har sine egne unikke ID'er.

Alle endpoints er RESTful, dvs. de følger principperne for Representational State Transfer (REST). Dette indebærer:

1. Ressourcebaseret arkitektur: Hver endpoint repræsenterer en specifik ressource (kunder, forhandlere, betalinger).
2. Stateless interaktioner: Serveren gemmer ikke klientens tilstand mellem anmodninger. Hver anmodning indeholder al minimalt nødvendig information.
3. Brug af HTTP-metoder: Endpointsene anvender standard HTTP-metoder korrekt:
    - POST til at oprette nye ressourcer (`/customer, /merchants, /payments`)
    - GET til at hente ressourcer (`/payments`)
4. Uniform interface: Endpointsene har en konsistent struktur og navngivning.
5. Repræsentation af data: Data udveksles i et standardformat, netop JSON.
6. Idempotente operationer: GET-anmodninger ændrer ikke systemets tilstand.

Event-baseret kommunikation

Systemet kan tilgås udefra via REST, men internt er det udelukkende message queues med RabbitMQ, hvis en funktionalitet ikke er i samme projektmappe. Quarkus har sit eget

abstraktionslag til at oprette og lytte til message queues. I dette projekt er der dermed blevet gjort brug af `Emitter` klassen, samt `@Incoming` og `@Outgoing` annotationerne.

Et helt basalt eksempel er når Payment microservicen får besked på at opsamle og udsende hele betalingshistorikken:

```
@Incoming("all-payments-requested")
@Broadcast
@Outgoing("all-payments-assembled")
@Blocking
public AllPaymentsAssembled getAllPayments(JsonObject jsonEvent) {
        AllPaymentsRequested event =
jsonEvent.mapTo(AllPaymentsRequested.class);
        return new AllPaymentsAssembled(
                        event.getCorrelationId(), new ArrayList<>
(paymentsRequests));
}
```

Her ser man at `getAllPayments` metoden er annoteret med `@Incoming("all-payments-requested")`, dvs når der er et nyt event på `all-payments-requested` kanalen så kaldes denne metode. `paymentsRequests` er listen over alle betalinger, en simpel variabel. Denne variabel returneres og udsendes via `@Outgoing("all-payments-assembled")` metoden. Dette er en fanout operation (grundet `@Broadcast` annotationen) og hele metoden er blokerende (pga. `@Blocking` annotationen).

Det er også muligt at separere udgående events til en `Emitter`. Dette gør nogle ting lettere ved højere kompleksitet.

```
@Inject
@Broadcast
@Channel("payment-completed")
Emitter<PaymentCompleted> paymentCompletedEmitter;

public void emitPaymentProcessed(PaymentCompleted paymentCompleted) {
        paymentCompletedEmitter.send(paymentCompleted);
}

@Incoming("payment-requested")
@Blocking
public void onPaymentRequested(JsonObject jsonEvent) {
        PaymentRequested event = jsonEvent.mapTo(PaymentRequested.class);
        // do some work...
        paymentCompletedEmitter.send(paymentCompleted);
}
```

Her specificerer `@Channel("payment-completed)` annotationen den udgående kanal ligesom `@Outgoing` ville.

Forbundet til disse annotationer og definerede kanaler specificerer man alle de indkommende og udgående kanaler i `application.properties`. Siden vi ønsker at alle kan lytte til alle events og at kanaler og events egentlig skal fungere ens, bruger vi fanout med samme navn for både kanalen og exchangen.

```
# out all-payments-assembled
# in all-payments-requested
mp.messaging.outgoing.all-payments-assembled.connector=smallrye-rabbitmq
mp.messaging.outgoing.all-payments-assembled.exchange=all-payments-
assembled
mp.messaging.outgoing.all-payments-assembled.exchange-type=fanout

mp.messaging.incoming.all-payments-requested.connector=smallrye-rabbitmq
mp.messaging.incoming.all-payments-requested.exchange=all-payments-
requested
mp.messaging.incoming.all-payments-requested.exchange-type=fanout
```

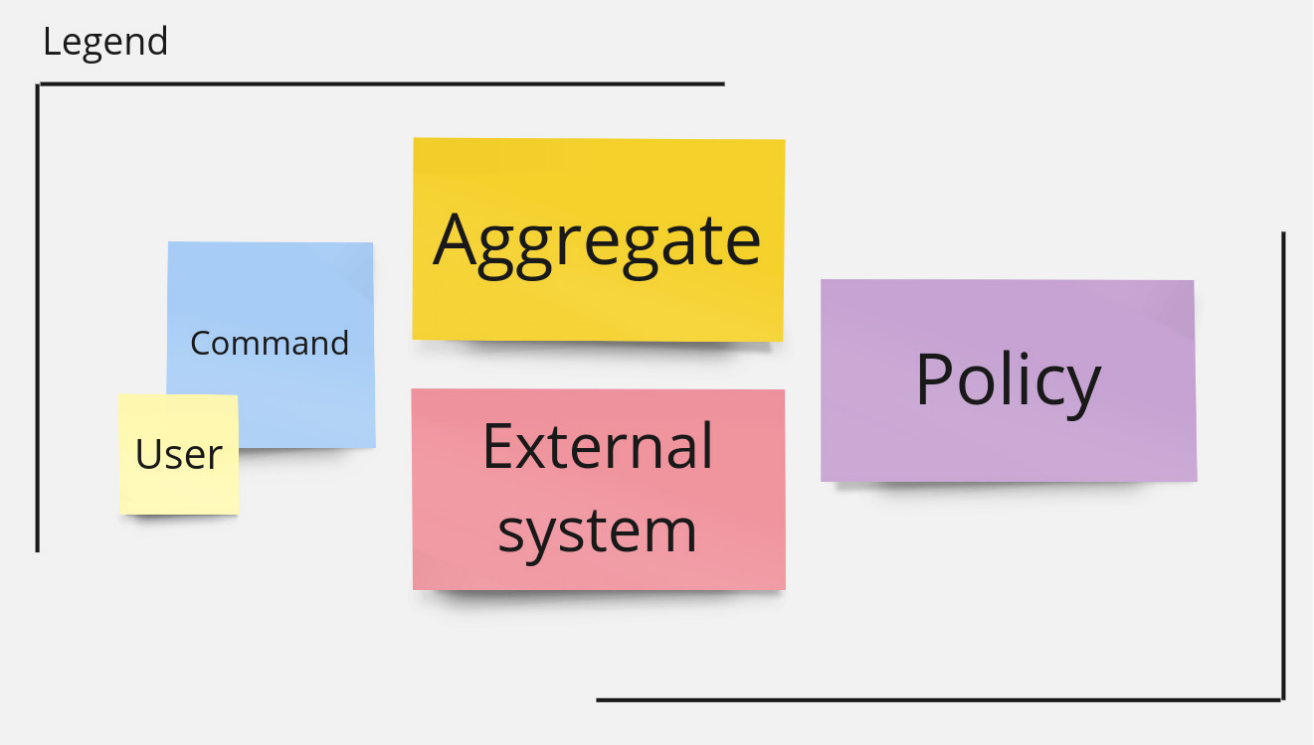
Der er i alt 19 unikke events fordelt på 19 kanaler.

Ligeså er al event baseret kommunikation defineret og opsat i systemet.

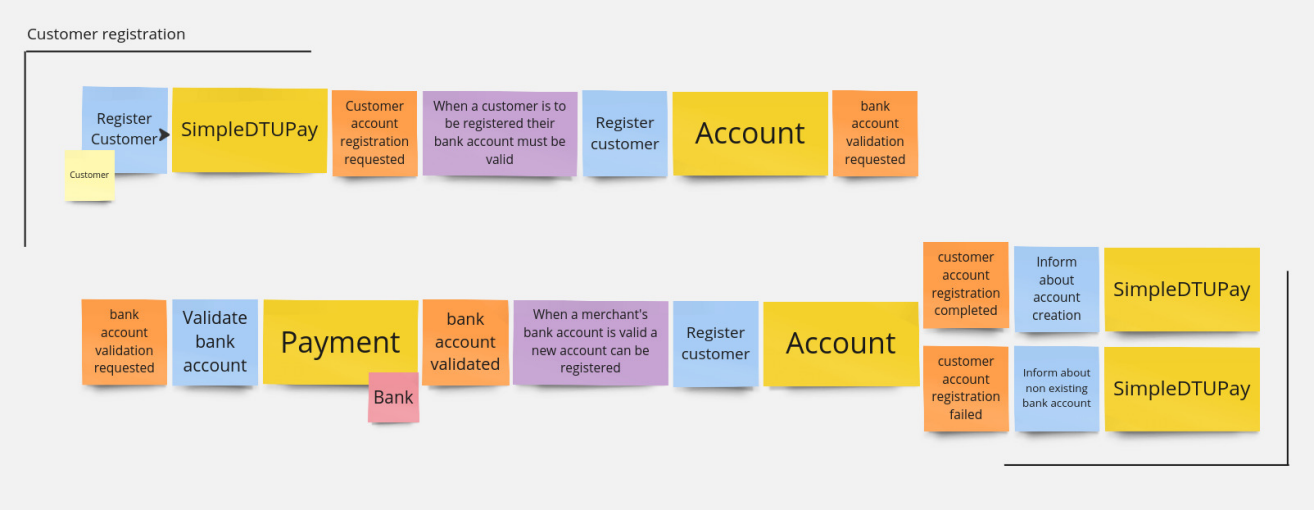
Event storming

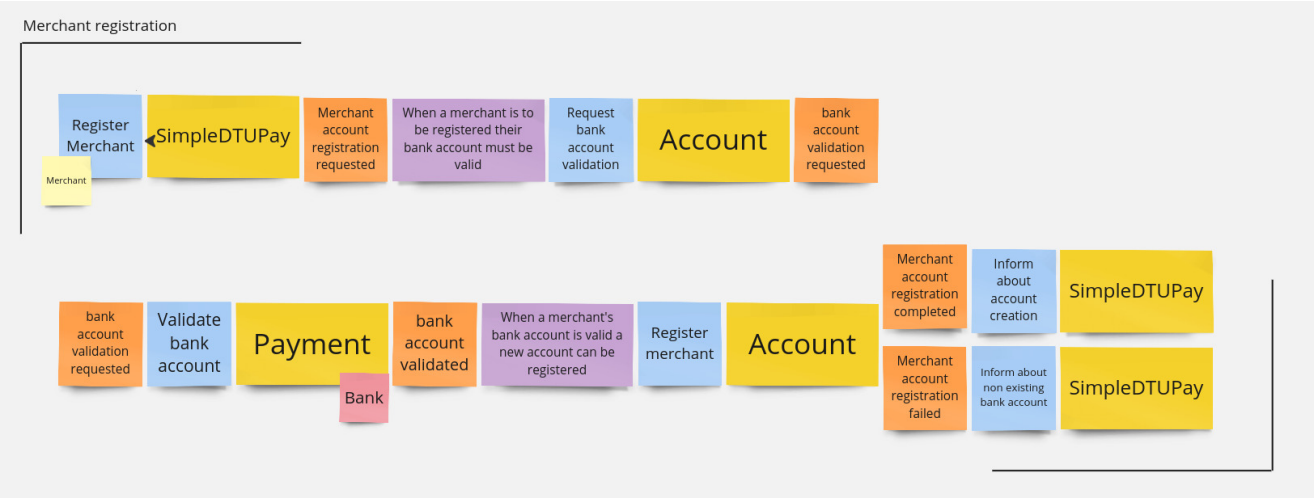
De følgende event-storming diagrammer er de udarbejdede user stories.

Det første diagram viser farvernes betydning.

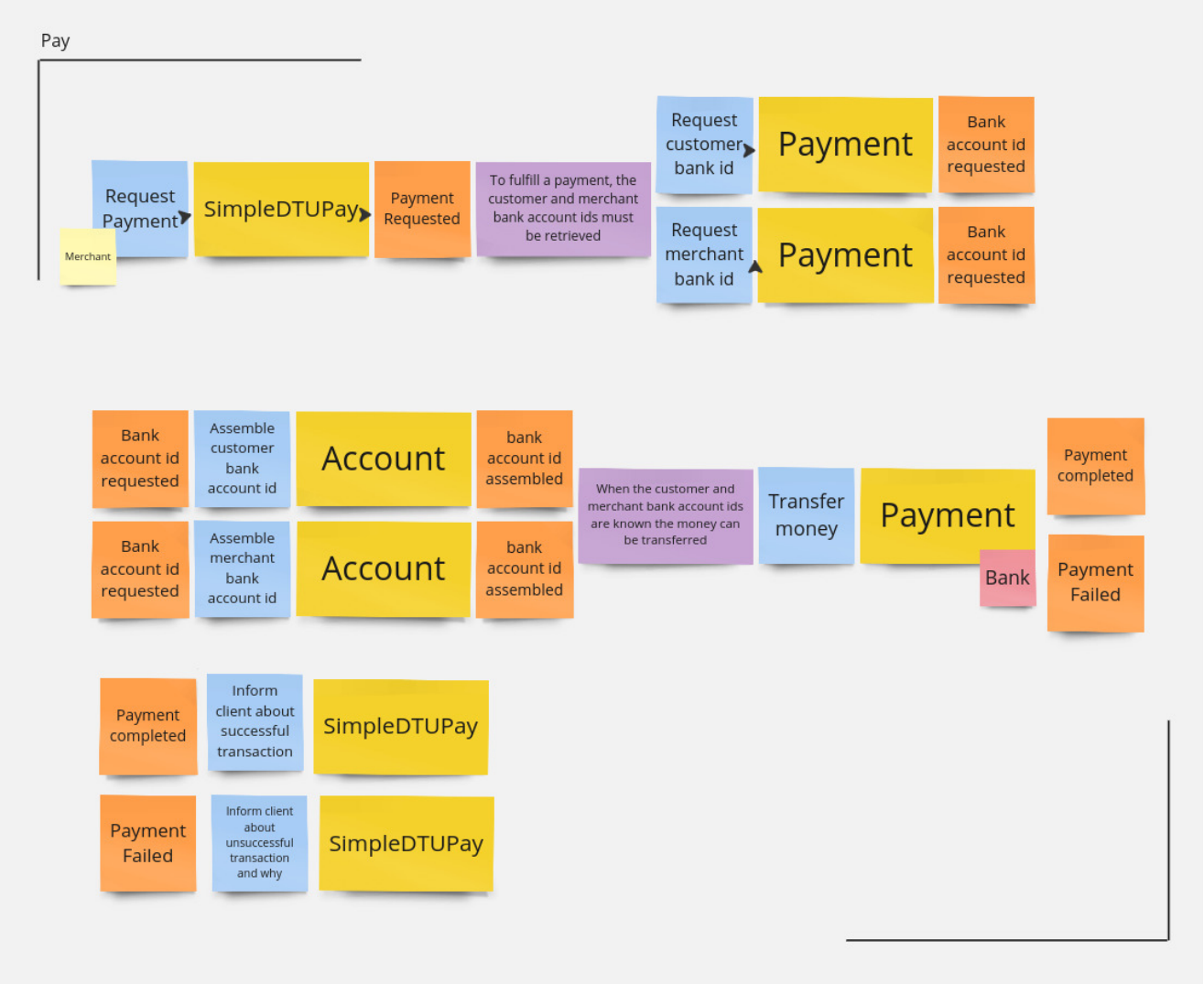


Når en customer skal registreres i SimpleDTUPay udsender SimpleDTUPay et event som Account microservicen lytter til. Account udsender heraf et event som Payment microservicen lytter til. Når Payment har valideret bankkontoen udsender den et event som Account lytter til og hvis bankkontoen var godkendt registrerer Account hermed den nye customer. Det samme flow er gældende for merchants

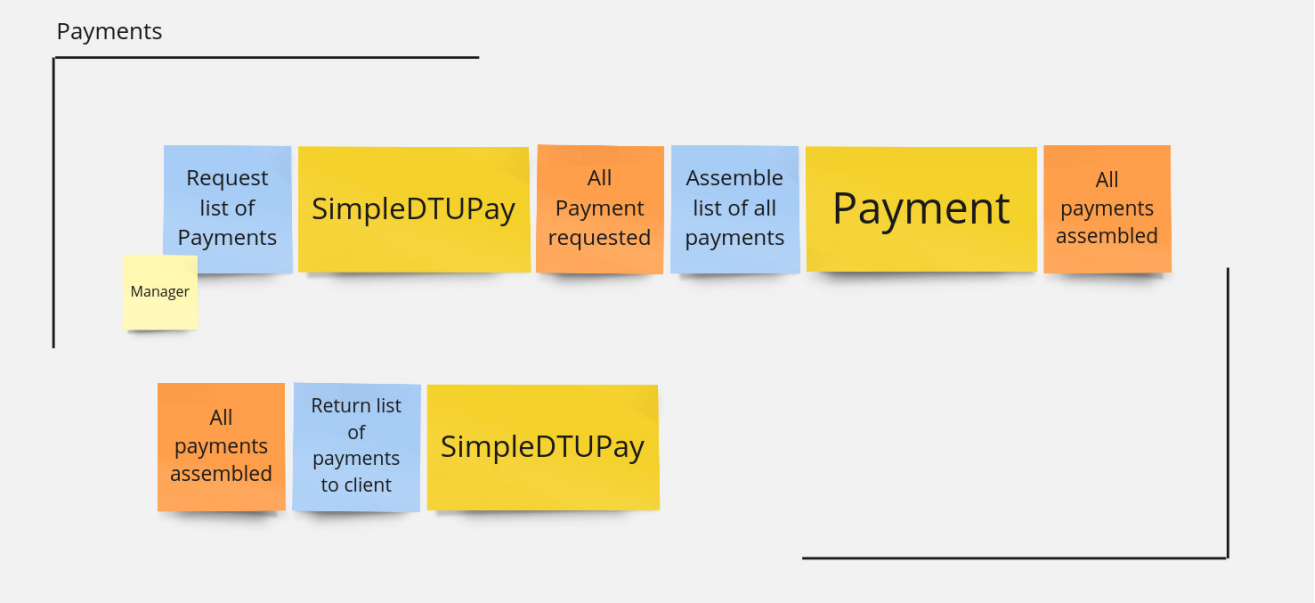




Når en betaling skal gennemføres er det igen først igennem SimpleDTUPay, som udsender et event. Heraf lytter Payment til eventet og sender et event ud for at få bank id'erne for customer og merchant. Account hører dette event og udsender de to bank konto ID'er og Payment lytter her. Når ID'erne er hentet kan transaktionen med det eksterne system Banken begynde. Heraf bliver der enten udsendt et Payment Failed eller Payment Completed event som SimpleDTUPay venter på.



Når en manager vil have en liste af alle betalinger kan de gøre det igennem SimpleDTUPay. Denne microservice udsender så et event og afventer et tilsvarende All Payments Assembled event. Det er Payment der modtager All Payments Requested og udsender All Payments Assembled.



Features

Der er i alt 4 unikke "happy-path" features i dette system.

1. Customer registrering
2. Merchant registrering
3. Betaling
4. Hent af liste over betalinger

Dette er svarende til de 3 endpoints beskrevet tidligere (hvor én af dem har dobbelt funktion afhængig af HTTP metoden anvendt).

Vi vil nu gennemgå hvordan betaling er implementeret, da det er den mest indviklede og den som har flest bevægende dele.

1. En bruger kalder `POST /payments` med en betalingsmængde, customerid og merchantid.

   ```
   {
     "amount": 0,
     "customerId": "string",
     "merchantId": "string"
   }
   ```

   Heraf modtager `SimpleDTUPay` microservicen dette og der udsendes et `PaymentRequested` event.

---

2. Payment microservicen lytter efter `PaymentRequested` events og handler på de medsendte data. Den udsender to events for at få fat i `customerId`'s tilsvarende bankaccountid og `merchantId` ligeså.

---

3. De to `BankAccountIdRequests` events håndterer Account microservicen. Den leder i sin in-memory liste over registrerede brugere og hvis den finder et tilsvarende bankaccountid sender den det ud via et `BankAccountIdAssembled` event.

---

4. Payment microservicen reagerer så på `BankAccountIdAssembled` eventet og kan fortsætte med at håndtere betalingen. Med de to fundne bankaccountids kan den nu kontakte Banken via REST og den medsender en JSON body med alle informationerne påkrævet at lave en betaling

```
{
  "amount": 0,
  "creditor": "string",
  "debtor": "string",
  "description": "string"
}
```

Hvor `creditor` og `debtor` er hhv. customer og merchant bankAccountIDs.

Nu hvor betalingen er succesfuld udsender Payment microservicen et `PaymentCompleted` event.

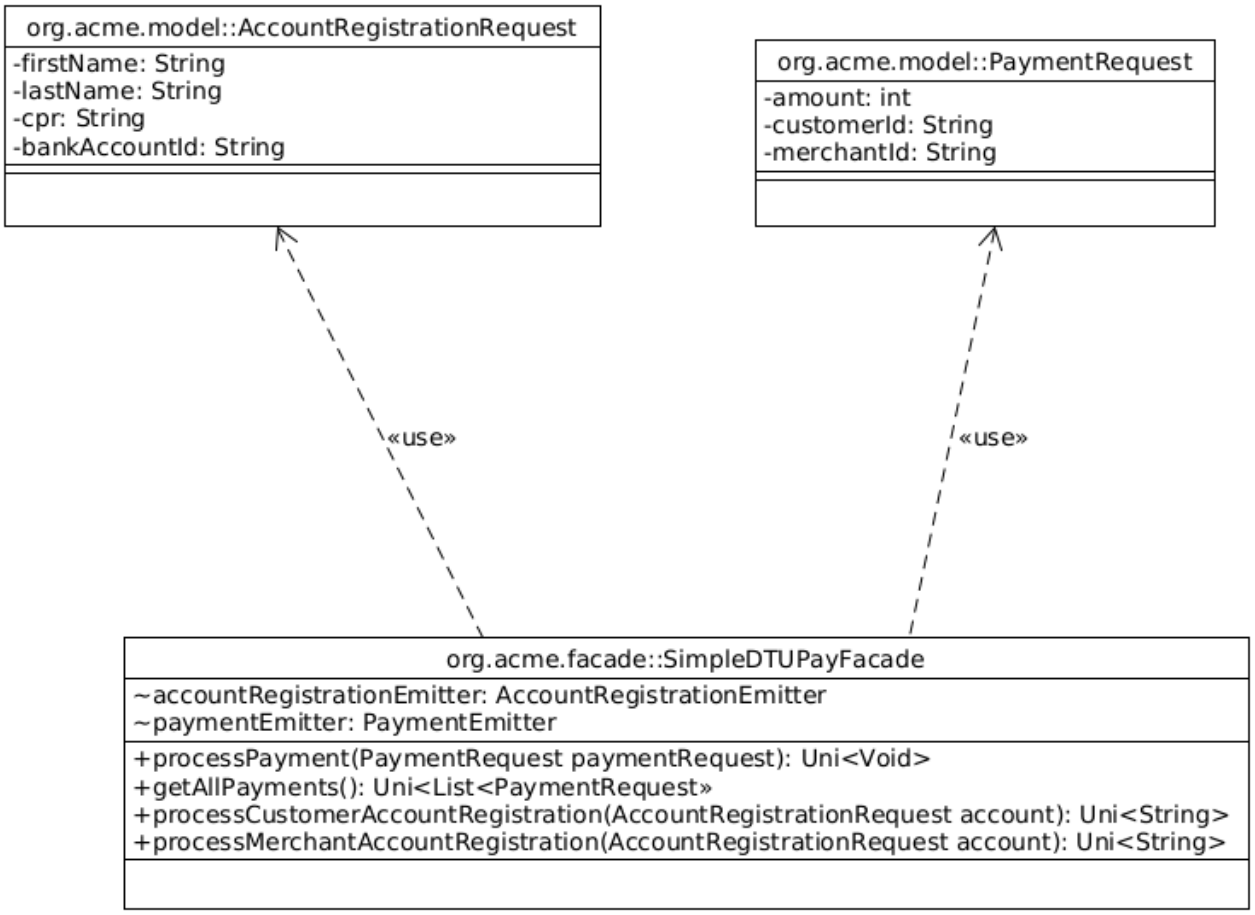
---

5. SimpleDTUPay servicen opfanger `PaymentCompleted` og returnerer en 200 HTTP kode til klienten.

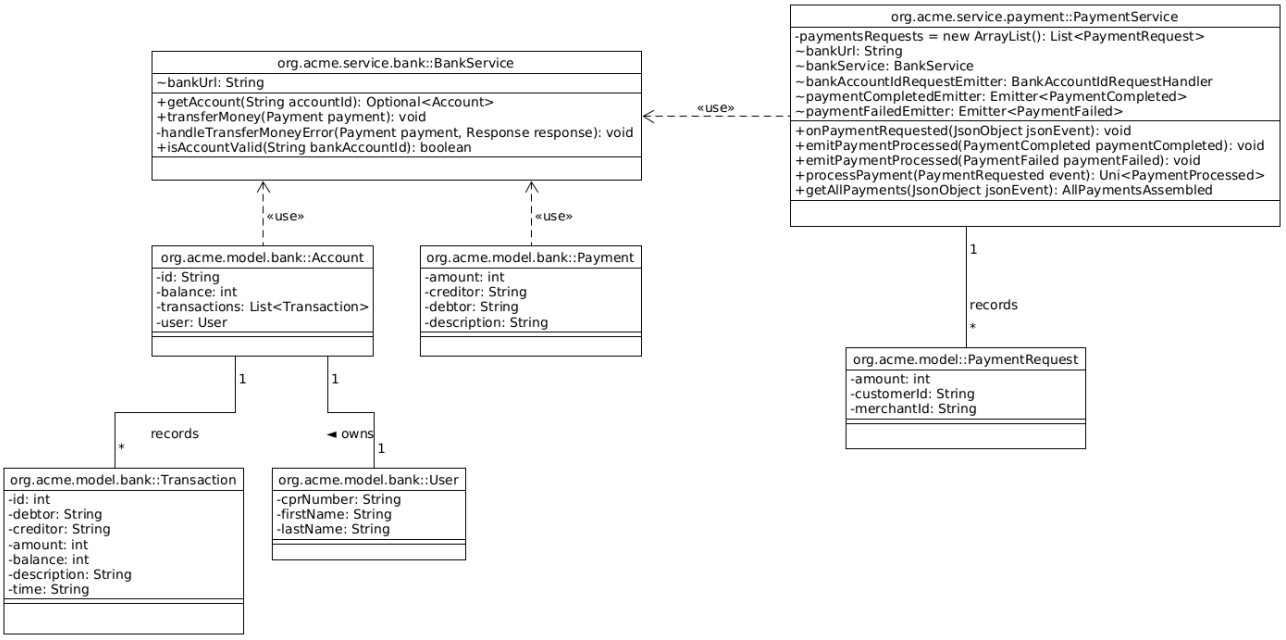
UML

Følgende er UML klasse-diagrammer over hver microservice

SimpleDTUPay

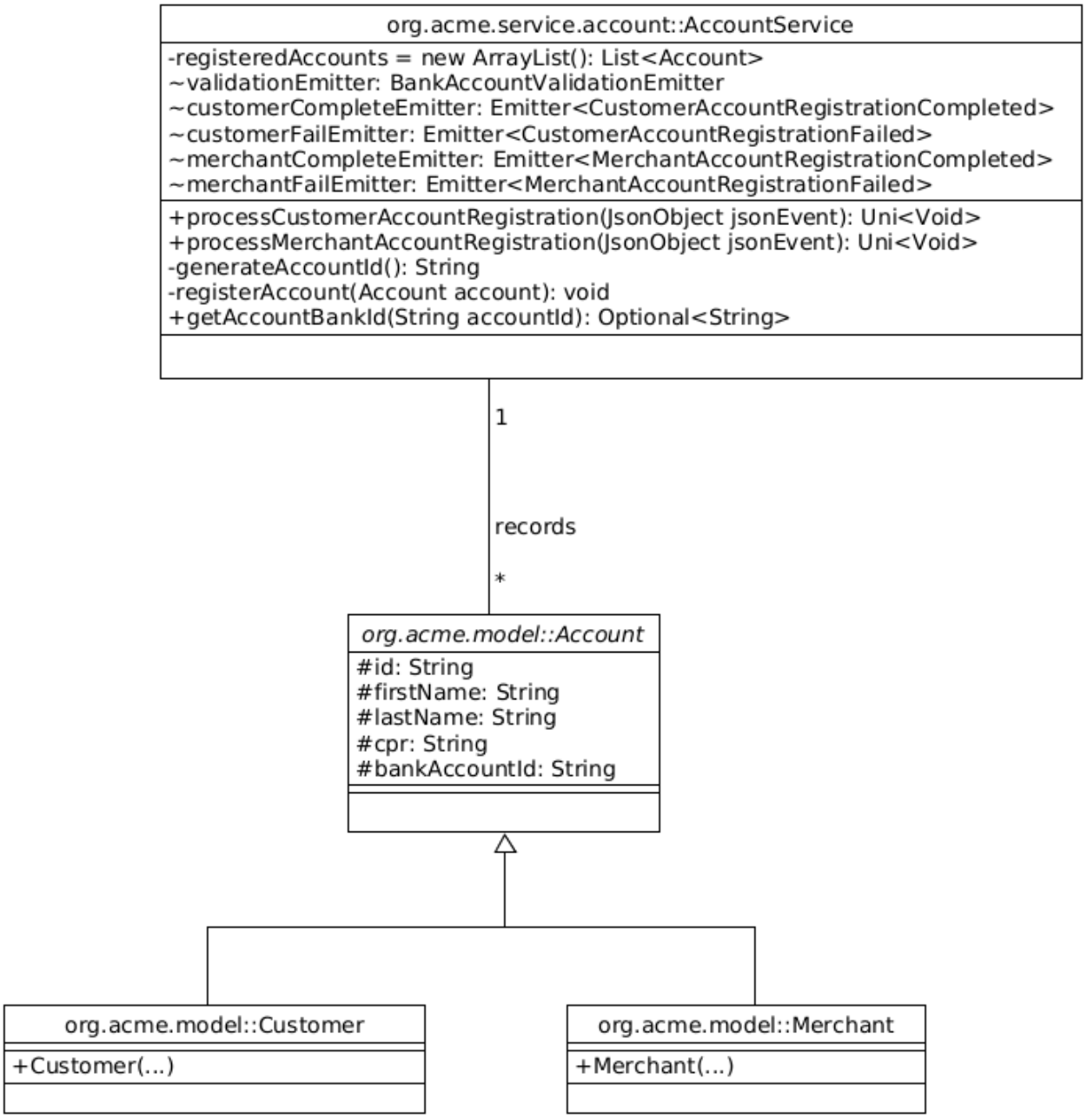


Payment



Payment microservicen gemmer alle paymentrequests i en in-memory liste.

Account



Account microservicen gemmer en liste over alle brugere i systemet.

Installation

Se installations PDF i roden af projektet.

Bilag

Kode

Følgende er github repositoriet hvor projektet er løbende udarbejdet.

https://github.com/multilingualism/02267-special-course

UML

UML diagrammerne findes i roden af projektet i `bilag/uml`. Der findes også `.uxf` format i undermappen `bilag/uml/uxf/`.

OpenAPI

OpenAPI specifikationen findes i `bilag/openapi`

Event-storming

Alle event-storming diagrammer findes som `.jpg` i `bilag/event-storming` der er også en legend til hvad hver farve betyder.