MICROSERVICES – DESIGN OG IMPLEMENTERING MED .NET CORE

UCL, PBA. Softwareudvikling

Synopsis

Event sourcing

Jesper Albrecht Madsen

Indhold

Problemformulering

Event sourcing er en naturlig forlængelse af arbejdet med DDD i microservices, men hvordan kan event sourcing implementeres?

Indledning

Denne opgave vil se på hvordan event sourcing kan implementeres i en microservice arkitektur. Jeg prøver at fremhæve nogle principper, men beskriver kun tingene generelt.

I opgaven kommer jeg ikke nærmere ind på DDD i øvrigt, og jeg styrer også bevidst udenom en dybdegående beskrivelse af hvordan persistering fungerer sammen med event sourcing.

Jeg fandt det naturligt at inkludere lidt om CQRS, da jeg har fået en fornemmelse af at det spiller godt sammen med event sourcing, det er dog et emne der fortjener sin egen fulde opgave, så heller ikke her går jeg i dybden.

Sammenligning mellem Eventsouring og traditionel tilgang

De traditionelle entiteter vi typisk bruger i OOP er flade, forstået på den måde at de altid vil vise sin seneste tilstand. Det samme gælder i princippet også for en eventsourced entitet, men denne indeholder samtidig en liste af ændringer, der starter med dens initialisering og slutter ved den seneste ændring. Dette betyder at når en eventsourced entitet skal hentes, vil alle dens events blive afspillet i rækkefølge for at genskabe den seneste tilstand. Derved får man også automatisk givet en log af ændringer, da disse netop ligger i en liste på entitets niveau. Dette giver den fordel at det er nemt at genskabe en bestemt tilstand, da man blot skal afspille alle events op til det givne tidspunkt eller den givne tilstand.

Dette er mere besværligt ved den traditionelle tilgang. Her skal man sørge for god logning, hvis man vil genskabe tidligere tilstande.

Vil man lave opslag i sit data, er der også ting man skal være bevist om. Ved den traditionelle tilgang, har alle entiteter altid deres seneste tilstand, og man kan nemt lave forespørgsler i en liste af dem.

Med de eventsourcede entiteter skal hver enkelt entitet først skal afspille alle sine events, før de endelige entiteter er klar til at blive forespurgt imod.

Dette imødekommes typisk ved at lave deciderede læse modeller, som opdateres efter entiteten er blevet ændret og ændringen er blevet persisteret. Læsemodellen skal altså også vedligeholdes, og det er denne der bruges til at lave læse-forespørgsler på entiteten igennem systemet.

I denne forbindelse skal man være bevidst om at der opstår en mindre race condition. Ændringen bliver persisteret før læsemodellen bliver opdateret. Man kan med andre ord, få en forældet repræsentation ud igennem læsemodellen, selvom der allerede er kørt en ændring ind i systemet. Hvad sker der så hvis brugeren forsøger at ændre tilstanden yderligere, men har et udgangspunkt der ikke er 100% aktuelt?

Hvordan er en eventsourced entity bygget op?

Det er vigtigt at man styrer hvordan man tilføjer ændringer på en entitet, så alle events bliver lagt i kø og kan påføres entiteten uden at dens tilstand bliver ugyldig.

Grundlæggende skal en entitet indeholde følgende:

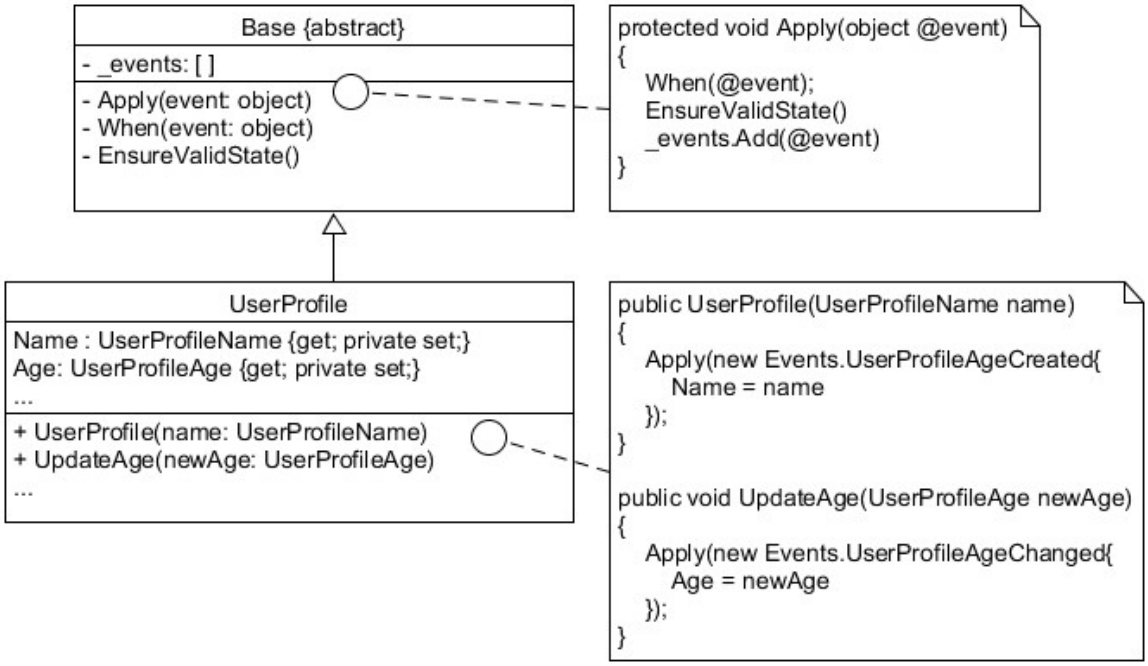
- En liste over ændringer (*_events*)
- En metode (*Apply()*) der koordinerer:
    - En eksekvering af ændringen.
    - At ændringen ikke ødelægger entitetens tilstand.
    - At ændringen bliver lagt i listen af ændringer.
- En metode (*When()*) der afspiller en ændring.
- En metode (*EnsureValidState()*) som sikrer at en ændring ikke ødelægger entitetens tilstand.

Alle ovenstående punkter skal være afskærmet fra resten af systemet, og kan kun tilgås fra indeni entiteten. Disse mekanismer kan placeres i en abstrakt klasse, så grundfunktionaliteten kan genbruges til andre entiteter.

Den interne logik kræver en korrekt opbygning af funktionalitet i entiteten. Det betyder at for hver type ændring der er, er der også en dedikerede metode, og det er denne metodes ansvar at:

- tage imod relevante værdier som parametre.
- omsætte værdierne, som repræsenterer ændringen, til et event.

- Eventet sendes videre til den interne Apply-metode som koordinerer resten



Det er sjældent at et system kun holder på en entitet. Så hver entitet bliver også udstyret med en form for identifier. På den måde kan vi også holde styr på hvor de forskellige events hører til, ved at stemple events med entitetens id. Entiteternes id'er bliver naturligvis også brugt i forbindelse med persistering.

Eventsourcing og CQRS

I et system kan du udføre mange handlinger. Når du benytter CQRS, som betyder "Command Query Responsibility Segregation", definerer du hver handling til enten:

- at ændre en tilstand i systemet – write-model.
- at læse en tilstand i systemet – read-model.

Den eventsourcede entitet er vores write-model. Entiteten er systemets source of truth. Alle ændgeringer kommer igennem her som beskrevet i afsnittet ovenfor.

Read-model

Men hvordan aflæser du tilstanden af en entitet der er event sourcede?

Du får entitetens aktuelle tilstand når du har hentet alle events og afspillet dem. Dette har en lille omkostning, som måske/måske ikke er i orden. Det store problem opstår når du vil lave forespørgsler på

flere entiteter. Tusindvis af entiteter skal gendannes på baggrund af millionvis af events. Først når dette arbejde er færdig, kan du udføre din egentlige forespørgsel, hvor resultatet måske kun indeholder dele af en eller flere entiteter. Det kan sagtens lade sig gøre, men det bliver omkostningsfuld og performer dårligt, jo flere events systemet skal holde på.

Svaret er at overføre events til en såkaldt ”read-model” af entiteten.

Read-modellen er altså en tilføjelse til systemet. Modellen holdes opdateret efterhånden som ændringerne kommer ind i systemet. Modellen er ”flad”, forstået på den måde at den ikke indeholder nogen tidligere ændringer, men skal tilsvare den mest aktuelle tilstand af den eventsourcede entitet. Gevinsten er at alle forespørgsler på read-modellen udføres i en mere traditionel forstand, og kræver ikke omfattende databehandling inden.

Read-modellen og eventual consistency

Read-modellen bliver først opdateret efter write-modellen, hvilken betyder at der kan komme tidspunkter hvor read-modellen ikke afspejler den seneste tilstand og derfor kommer der et problem med consistency mellem modellerne.

Systemet håndterer dette ved at inkludere en versionering i entiteterne på både read og write siden. Ved hjælp af versionsnummeret ved Read-modellen hvor langt den er nået, og der skal kun tilføjes ændringerne fra events der er højere end modellens versionsnummer.

Versioneringen er også brugbar når der indsendes en ændring til systemet. Hvis versionsnummeret på entitet og ændring stemmer overens, bliver ændringen udført.

Men hvis entiteten afspejler en nyere version, end ændringen afspejler, så skal systemet tage en beslutning om hvordan det skal fortsætte. Der er groft sagt to muligheder:

- Entiteten er blevet opdateret med event af typen X, og nu opdaterer du med event af typen Y.
    - Der er ikke nogen egentlig konflikt – og ændringen gennemføres alligevel.
- Entiteten er blevet opdateret med event af typen X, og nu opdaterer du med et tilsvarende event.
    - Man gennemfører ikke samme type event lige efterhinanden, hvis ikke versionen er helt aktuel.
    - Du vil formentlig få en fejlmeddelelse om at entiteten blev opdateret af en anden, og at du bør hente den nyeste tilstand for at afgøre om du stadig vil indsende din ændring.

Ovenstående eksempler er kun en version fra hinanden. Hvis der er en større forskel mellem versionsnummeret i systemet og på beskeden, bør ændringen formentlig ikke gå igennem og brugeren skal opfordres til at opdatere, inden han kan fortsætter med ændringen.

Afrunding

Event sourcing giver dig nogle overordnede principper du bør følge:

- Alle ændringer går igennem entiteten
    - Ændringer bliver til events
    - Der sikres at events ikke ødelægger entitetens tilstand
    - Eventet lægges i entitetens liste af ændringer
- Forespørgsler foregår igennem en dedikeret read-model
    - Modellen er en repræsentation af den egentlige entitet.
    - Modellen kan forespørges
    - Den er eventual consistent
        - det vil sige du får måske en forældet tilstand, selvom den egentlige entitet er blevet opdateret.
        - Den skal nok blive consistent – vi kan bare ikke garantere det i samme øjeblik du forespørger.

De to grundlæggende måder at tilgå systemets entitet på, spiller godt sammen med CQRS, der netop lægger op til en operation der kan skrive ændringer og en operation der kan læse.

Problemstillinger til drøftelse

- Kan det betale sig at implementere Event sourcing?
- Hvordan passer Event Sourcing ind i DDD-tankegangen?

Kilder:

Zimarev, Alexey: Hands-On Domain-Driven Design with .Net Core, 2019 Packt Publishing