UCL ERHVERVSAKADEMI OG
PROFESSIONSHØJSKOLE

SYNOPSIS

.NET Core og Microservices

Koordinering af forretningsprocesser på tværs af services

Julian Mathias Kock

Underviser
Kaj BROMOSE

24. maj 2020

Indhold

1 Indledning

Vores verden er blevet en digital verden. Alt lige fra ens køleskab og elkedler til valutaer og telefoner er blevet digitale. Internetforbindelserne bevæger sig nu med lysetshastighed og en efterspørgelse på ingen nedetid, hurtige svartider og overskuelighed er blevet hverdag for udviklere.

Softwaren der udvikles har aldrig været under så stort et pres. Udviklere bliver sat til at løse endnu mere komplekse forretningsprocesser hvor hver enkelt kodeblok skal være let læselig, skalerbart og omskiftelig. Samtidig skal koden være gennemtestet, virke i alle mulige forskellige miljøer og integrere let mod andet software.

Med andre ord er arkitekturen samt infrastrukturen blevet vigtigere end aldrig før. For at imødekomme alle disse krav har man kigget mod opdeling af kode i en service baseret arkitektur, hvor hver service tildeles et ansvar og kan blive udviklet og skaleret individuelt. Fokus i dette fag har været Microservices som er et term for mange autonome services der arbejder sammen for at løs komplekse forretningsmæssige processer[2].

Denne opgave vil fokusere på den koordinering der skal til for at Microservices kan løse forretningsmæssigeopgaver, samtidig med at services skal være autonome og opnå eventual consistency.

2 Problemformulering

Hvordan opnår man en fornuftig koordinering af Microservices til at løse forretningsmæssigeopgaver?

Del spørgsmål

1 Hvad er SAGA-mønstret?

2 Hvordan kan man sikre autonome Microservices?

3 Hvad er eventual consistency?

3 Hvad er SAGA-mønstret?

3.1 Saga pattern

Når der tales om koordineringen af Microservices menes der den interne kommunikation/dataudveksling der skal til for at services kan udføre en forretningsproces. Et eksempel på en forretningsproces kunne f.eks. være at lave en ordre der tilsidst afsendes og modtages af en kunde. I en traditionel arkitektur kunne man bruge ACID transaktioner, men når der er tale om flere services er det svært at lave isoleret transaktioner. En måde hvorpå man kan imødekomme dette er ved brug af distrubureret transaktioner, så som 2PC (two-phase commit). Denne løsning kan være tilstrækkelig, men den giver dog nogle ekstra problemer. Ideén med distrubureret transaktioner er at alle services der indgår enten committer transaktioner eller laver en rollback. Det betyder altså at du nu er direkte afhængig af, om alle services er klar til og modtage requestet, eller ej. Derfor vil negativ downtime for en service have effekt på alle andre services der indgår i den distribureret transaktion[3]. Dette bryder på autonom princippet bag Microservices, og der er nu en direkte afhængighed videre fra service til service.

For at imødekomme kommunikation der skal gå på tværs af services kan man her bruge et Saga pattern. Saga mønstret er et mønstre hvorpå man kan lave transaktioner på tværs af services. Der findes to typer af sager. Choreography baseret og orchestration baseret[3].

3.2 Orchestration saga

For at forstå orchestration bruges oftest en analogi til den virkelige verden hvor man kigger på et orkester. I et orkester er der en person der orkestrerer orkestret, nemlig dirigenten. Dirigenten sørger for at alle i orkesteret rammer deres toner på de rigtige tidspunkt og aggregere alle instrumenters toner til at få leveret et stykke musik. Med denne analogi kan vi kigge på services som de enkelte instrumenter, og dirigenten som en Saga der sørger for at aggregere og præsentere data fra forskellige services. Et simpel eksempel kunne være en request/response hvor en klient beder om at få

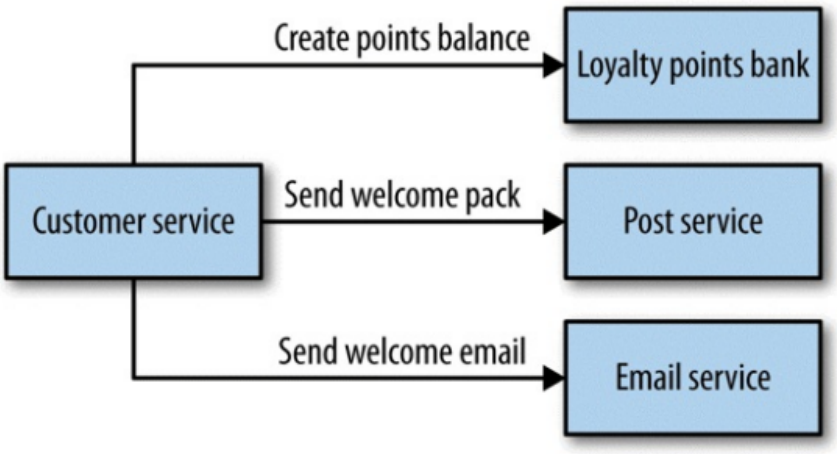


Figure 4-3. Handling customer creation via orchestration

Figur 1: [2, p. 91]

alle informationer om en kunde og kundens betalinger baseret på kundens id. Saga'en vil requeste kundeservicen der kan returnere detaljer om kunden og requeste betalingsservicen om alle betalinger oprettet af kunden. Saga'en vil så aggregere dette data og præsentere det i et response. En definition på en orkesteringssaga kunne være følgende.

Orchestration saga - I en orkesterings baseret saga centraliserer man koordinations logikken i en enkelt service. Denne service står for at sende beskeder afsted til services der fortæller dem hvilke funktioner de skal udføre og eventuelt aggregere data til et response[3].

3.3 Choreography saga

En choreography baseret saga er anderledes end en orkestreringssaga på den måde at der ingen logisk centralisering er. Her står services selv for koordinationen. Det kunne f.eks. være via. et event som overstående illustration. En analogi her kunne være en flok balletdansere. Her følger de oftest en koreografi hvor alle har hver sin rolle, og på bestemte tidspunkter skal reagerer på en bestemt måde. I dette tilfælde er SAGAens medlemmer balletdanserne og de ved internt hvordan de skal reagerer

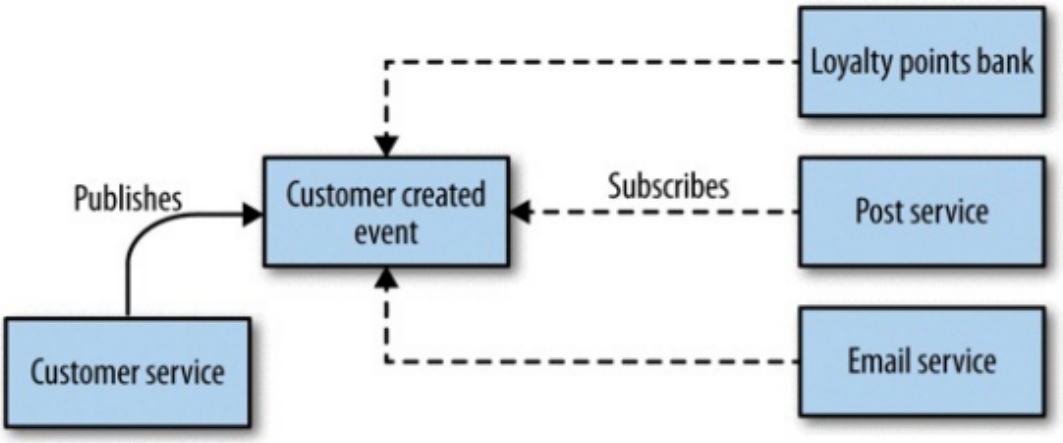


Figure 4-4. Handling customer creation via choreography

Figur 2: [2, p. 92]

når forskellige events opstår. Der opstår derfor ikke et central point of failure som ved orkestreringen og alt kan foregå asynkront via. events. En definition på en koreografi baseret SAGA kunne være følgende:

Choreography saga - I en koreografi baseret saga delegerer man koordinations logikken ud i hver enkelt service, og får dem til at kommunikere via. events[3].

4 Hvordan kan man sikre autonome Microservices?

I en Microservice arkitektur bestående af flere forskellige autonome services opstår der mange consumer-provider forhold. Ved dette menes der at der er nogle services der bruger (consumer) andre services (providers). Det kan f.eks. ses ved SAGA'er hvor en SAGA oftest har flere grænseflader hvor den er consumer. Disse grænseflader er også det man kalder kontrakter - den dataudveksling consumers og providers er blevet enige om[3]. Men hvordan sikrer man sig, at man ikke bryder denne kontrakt og dermed ødelægger dataudvekslingen?

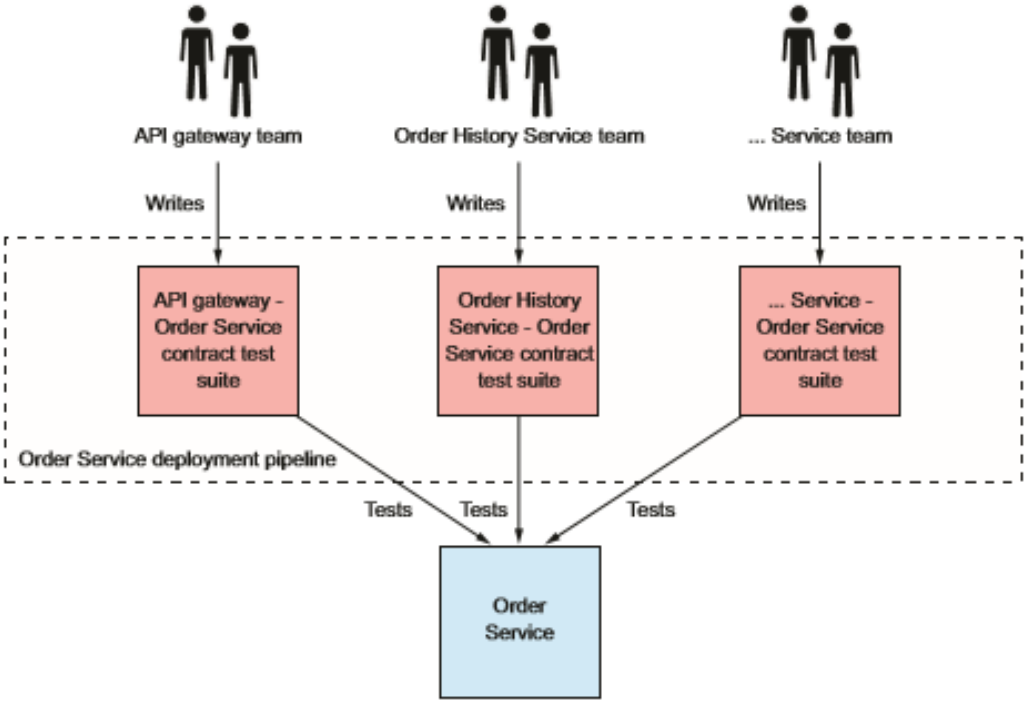
Her kommer API-versionering og Consumer-Driven contracts ind i billedet.

4.1 API-versionering

API versionering er en måde hvorpå man kan afholde dine consumers for breaking-changes. Breaking-changes er ændringer til din services grænseflade der potentielt kan give problemer for dine consumers. Det kunne være at ændre et endpoints sti, eller ændre navnet på et felt i dit objekt.

Ved API versionering følges oftest semantisk versionering: `https://semver.org/`, hvor API'ens sti oftest følges af et versioneringsnummer. Semantisk versionering forklarer hvordan du skal inkrementere din versionering. Versioneringen følger et skema som er således MAJOR.MINOR.PATCH, hvor en breaking change vil resulterer i, at MAJOR inkrementerer med en[1].

4.2 Consumer-Driven Contracts



Figur 3: [3, p. 302]

I et normal udvikling scenarie vil en provider have udviklet nogle endpoints med dertilhørende unittests der sørger for, at deres grænseflade/kontrakt opfylder og sender korrekt data. Dette kaldes en Provider-Driven contract - det er provider der bestemmer hvad kontrakten indeholder, og ikke consumeren. Ved consumer-driven contracts skriver consumeren tests til provideren, som så bliver en del af providerens test suite[3].

5 Hvad er eventual consistency?

En af de store udfordringer ved brug af Microservices af dataudveksling da alle services har deres eget database skema. Det betyder at vi har flere transaktionelle grænser for vores services, og ikke en, som i en monolit. Derfor er man nød til at kigge på, hvordan dataen på et tidspunkt kan være konsistent på tværs af flere transaktionelle grænser. Dette kaldes eventual consistency - man accepterer faktummet om, at alle processer ikke arbejder med det nyeste data, men på et tidspunkt i processen vil komme til det[2].

6 Sammenfatning

Det overordnede spørgsmål lød således: **Hvordan opnår man en fornuftig koordinering af Microservices til at løse forretningsmæssigeopgaver?**

For at opnå en fornuftig koordinering af services er det vigtigt at:

gøre brug af SAGA mønstret,

versionere sit API til sine consumers,

lave Consumer-Driven Contracts og dertilhørende tests,

acceptere et eventual consistency princip,

Overstående liste af overvejelser giver følgende problemstillinger der kan drøftes:

7 Problemstillinger til drøftelse

1 Hvornår skal man enten bruge orchestration eller choreography?

2 Hvornår er noget en 'breaking-change' for dine grænseflader?

3 Hvordan tager man højde for eventual consistency?

Litteratur

[1] Semantic Versioning summary. `https://semver.org/#summary`. Accessed: 2020-05-24.

[2] Sam Newman. *Building microservices: designing fine-grained systems*. "O'Reilly Media, Inc.", 2015.

[3] Chris Richardson. *Microservices Patterns: With Examples in Java*. Manning Publications, 2019.