Engineer

Microservices

en arkitektur af små tjenester, der hver udrulles for sig

Hvad det er

Microservices er en arkitekturstil, der deler én applikation op i små tjenester, hvor hver tjeneste er et selvstændigt program, der kører som en separat proces og kan udrulles uden de andre. James Lewis og Martin Fowler, hvis artikel fra 2014 gav stilen dens mest kendte beskrivelse, sammenfatter den som at bygge applikationer som »suites of independently deployable services«. De opregner, hvad den slags systemer typisk har til fælles: tjenester, der taler sammen via lette mekanismer, ofte et HTTP-API til ressourcer; tjenester bygget op om forretningsfunktioner; fuldt automatiseret udrulning; et absolut minimum af central styring; og frihed for hver tjeneste til at vælge sit eget programmeringssprog og sin egen datalagring. De understreger, at der ikke findes nogen formel definition, kun kendetegn, som de fleste systemer med betegnelsen deler.

Ordet er yngre end idéen. En fodnote i samme artikel fortæller, at en workshop for softwarearkitekter nær Venedig drøftede betegnelsen i maj 2011, og at den samme gruppe slog sig fast på »microservices« i maj 2012. Adrian Cockcroft hos Netflix havde kaldt tilgangen en finkornet form for SOA, og Lewis og Fowler påstår ikke, at stilen er ny: de finder dens rødder mindst så langt tilbage som de principper, Unix blev bygget på.

Det, stilen som regel holdes op imod, er monolitten: en applikation på serversiden, der bygges og udrulles som én enhed. Al logik for en forespørgsel kører i én proces. Med lidt omhu kan hele applikationen køres og testes på en udviklers bærbare, og den skaleres ved at køre flere kopier bag en load balancer. Gnidningerne kommer senere. En lille ændring betyder, at det hele skal bygges og udrulles igen, en god modulær struktur er svær at holde ved lige år efter år, og den del, der mangler kapacitet, kan kun skaleres sammen med alt det andet.

Lewis og Fowler definerer en komponent som en enhed af software, der kan udskiftes og opgraderes for sig. Et bibliotek linkes ind i et program og kaldes i hukommelsen. En tjeneste er en komponent uden for processen, som man når over netværket med en webforespørgsel eller et fjernkald (remote procedure call). Forskellen har betydning for koblingen. I de fleste sprog er det kun dokumentation og disciplin, der holder anden kode ude af et biblioteks indre, mens en tjeneste kun kan nås gennem eksplicitte fjernkald, så dens grænseflade er langt sværere at gå uden om. En tjeneste kan godt bestå af mere end én proces, for eksempel en applikation sammen med en database, som kun den bruger.

»Micro« siger mindre, end det giver indtryk af. Lewis og Fowler påpeger, at navnet leder opmærksomheden hen på størrelsen og fører til diskussioner om, hvad der tæller som »micro«. De største tjenester, de havde hørt om, fulgte Amazons idé om et two-pizza team, altså højst et dusin mennesker, og i den lille ende så de et team på et halvt dusin passe et halvt dusin tjenester.

Brug begrebet i en beslutning

Brug »Hvad er microservices?« i en konkret beslutning. Vi kan knytte begrebet til det system, I bygger eller køber. Den del, brugerne aldrig ser: hvor data ligger, hvilke regler der gælder, og hvordan andre systemer må tale med den.

Se vores ydelser Kontakt os →

Undersøg antagelserne bag ordet

Brug forklaringen og det tilknyttede arbejde til at formulere spørgsmål til jeres projekt. Vi kan undersøge antagelserne sammen, før et værktøj vælges. Har designet og drevet 6 ETL/ELT-pipelines — 6 ETL-pipelines på tre databasemotorer

Har designet og drevet 6 ETL/ELT-pipelines Kontakt os →

Hvorfor det betyder noget

Den første gevinst er, at hver del kan udgives i sit eget tempo. Når en applikation er delt op i tjenester, kræver mange ændringer kun, at én tjeneste udrulles igen, så et team kan levere uden at vente på de andre. Lewis og Fowler gør det klart, at det ikke gælder uden undtagelse: en ændring i en tjenestes grænseflade skal stadig koordineres med de tjenester, der kalder den.

Den anden er, at softwaren kan følge organisationen i stedet for at kæmpe imod den. I 1968 argumenterede Melvin Conway i Datamation for, at organisationer, der designer systemer, er bundet til at frembringe design, der kopierer deres egne kommunikationsstrukturer. Lewis og Fowler holder det op mod den gængse måde at dele en stor applikation på, nemlig efter tekniske lag, med ét team til UI, ét til logikken på serversiden og ét til databasen. Selv en simpel

ændring bliver så et projekt på tværs af teams, og logikken havner der, hvor et team tilfældigvis har adgang. Med microservices vender man det om. Hver tjeneste dækker én forretningsfunktion, for eksempel ordrer, lager eller betalinger, hele vejen fra brugerfladen ned til lagring, og hører til et tværfagligt team med de færdigheder, hele skiven kræver. Teamet bør eje tjenesten i hele dens levetid i stedet for at aflevere den, når projektet slutter, og Amazons udgave af reglen er, at den, der bygger en tjeneste, også driver den i produktion.

Den tredje er, at logikken bliver i tjenesterne. Lewis og Fowler kalder princippet smart endpoints and dumb pipes. Tjenesterne taler sammen via HTTP-forespørgsler til ressource-API'er eller via en let beskedbus, der ikke gør andet end at route beskeder, og beslutningerne træffes i tjenesterne i hver ende. Det er et bevidst skridt væk fra enterprise service bus'en, som router, transformerer og anvender forretningsregler midt i netværket i stedet for i dets ender.

Data deles langs de samme linjer. Hver tjeneste styrer sin egen database, og forskellige tjenester kan bruge forskellige databasesystemer, hvis det passer til opgaven. Eric Evans' domænedrevne design giver begrundelsen: et stort system rummer flere modeller, og hver af dem har brug for en eksplicit afgrænset kontekst (bounded context), fastlagt ud fra teamets organisering, de dele af applikationen, modellen gælder for, og fysiske ting som kodebaser og databaseskemaer. Salgsafdelingens billede af en kunde og supportens billede af den samme kunde kan så være forskellige, uden at nogen skal forlige dem i én tabel.

Omkostningerne hører med til definitionen, og Lewis og Fowler lægger dem frem uden omsvøb. Et fjernkald koster langt mere end et kald i hukommelsen, så grænsefladerne mellem tjenester må være grovere. Det er svært at flytte kode hen over en tjenestegrænse: ændringer i grænseflader skal koordineres, der skal bygges lag af bagudkompatibilitet, og testen bliver mere kompliceret. Distribuerede transaktioner er svære at implementere, så tjenesterne koordinerer uden dem, accepterer, at data måske først bliver konsistente med tiden (eventual consistency), og retter fejl med kompenserende operationer. Ethvert kald kan fejle. Og passer tjenesterne ikke rent sammen, er kompleksiteten ikke forsvundet, den er flyttet ud i forbindelserne mellem dem, hvor den er sværere at få øje på. Lewis og Fowler gengiver ét argument, de finder rimeligt: byg en monolit til at begynde med, giv den rene moduler, og bryd den først op, når den begynder at give problemer. De tilføjer et forbehold: en grænseflade, der fungerer godt inde i én proces, er sjældent en god grænseflade mellem tjenester.

Sådan ser det ud i praksis

Noten At sælge den sidste billet præcis én gang griber først til en tjeneste, når én database ikke længere slår til, og fastslår så den regel, der holder designet sammen: hver vare har præcis én ejer af sit lager, en lagertjeneste, og intet andet bestemmer. Tjenesten shardes efter lejer og vare, og de største lejere får deres egne celler. Kun reservationen af lager behøver at være synkron. Betaling, ordre og kvittering er separate systemer, og ingen transaktion spænder over dem. Noten afviser tofaset commit, fordi det blokerer, mens en deltager er langsom, og fordi betalingsudbydere ikke deltager i det. I stedet bruger den en saga: en række lokale trin, hver med et trin, der gør det ugjort.

Hvert trin lægger sin besked i en outbox-tabel inden for den samme transaktion som den ændring, beskeden fortæller om. I PostgreSQL kan en datamodificerende sætning inde i `WITH` give sit resultat videre til en indsættelse, så lagertjenesten kan reservere en enhed og registrere beskeden i én og samme sætning:

```
-- lagertjenesten, med sin egen database
WITH held AS (
    UPDATE inventory
    SET available = available - 1
    WHERE tenant_id = 't-7' AND item_id = 'item-881'
      AND available >= 1
    RETURNING item_id
)
INSERT INTO outbox (order_id, item_id, event, units)
SELECT 'A-17', item_id, 'StockReserved', 1 FROM held;
-- udsolgt: ingen række matcher, så der skrives ingen besked
```

Bagefter udgiver change data capture rækkerne fra outbox-tabellen, og hver forbruger er idempotent med ordren som nøgle, fordi beskeder kommer mindst én gang. I denne skitse rører ingen anden tjeneste tabellen `inventory`: de andre hører om lageret gennem disse beskeder eller ved at spørge lagertjenesten.

Den samme note viser også, hvor opdelingen hører hjemme i rækkefølgen. På dens liste over, hvad man gør først, kommer det at give lageret sin egen tjeneste og sin egen ejer ind som nummer fire, efter at måle, efter at gøre skrivningen til én betinget sætning og efter at skære belastningen ned med et venteværelse, en port og et cachet »udsolgt«. De billige løsninger kommer først.

Porteføljecasen En maritim platform bygget i rene lag dokumenterer grundlæggerens arkitekturbidrag til NextMariner og viser den anden side af valget. Det var et ungt produkt, hvis krav ville flytte sig en hel del, og målet var, at en ændret forretningsregel ikke skulle betyde rettelser i frontenden, API'et og databasen på én gang. Løsningen blev tre lag med én opgave hver: en Next.js-frontend til præsentationen, et bevidst tyndt Go-API, der router, validerer, står for sikkerhedsfiltreringen og orkestrerer, og al forretningslogik i PostgreSQL-funktioner. Grænserne blev skrevet ned og håndhævet i review. Lewis og Fowler bemærker, at modulære grænser inde i én applikation kræver megen disciplin at håndhæve, og at den mere eksplicitte adskillelse mellem tjenester gør det lettere at holde teamgrænserne klare. Platformen holdt disciplinen uden at opdele sin forretningslogik i separate tjenester.

Hvor det går galt

Kæder af synkrone kald ganger nedetiden op. Når svaret på én forespørgsel kræver, at den første tjeneste kalder en anden, som kalder en tredje, lykkes forespørgslen kun, så længe alle tre kører, og den venter på hver af dem efter tur. Lewis og Fowler kalder det nedetidens multiplikative effekt og nævner, at The Guardian har indført en regel om ét synkront kald pr. brugerforespørgsel. Hold den synkrone sti lige så kort, som noten holder den, hvor kun reservationen venter på et svar, og alt derefter er en besked. Giv hvert tilbageværende kald en timeout og en circuit breaker, de mønstre, Lewis og Fowler henviser til i bogen Release It!, så en tjeneste, der fejler, bliver koblet fra i stedet for at blive ventet på.

Tjenester, der altid ændres sammen. En opdeling langs de forkerte linjer giver tjenester, der ikke kan udgives hver for sig: hver ny funktion rører tre af dem, og de udgives i samlet trop. Lewis og Fowler giver testen: bliver man ved med at ændre to tjenester sammen, bør de slås sammen. En beslægtet fejl er at gøre metodekald til fjernkald ét for ét, så tjenesterne snakker frem og tilbage over netværket og yder dårligt. Træk grænserne om forretningsfunktioner og afgrænsede kontekster, gør grænsefladerne grove, og design hver tjeneste til at tåle ændringer i de tjenester, den kalder, så en leverandør kan udvikle sig uden at bryde sine forbrugere. Versionering ser Lewis og Fowler som en sidste udvej.

To tjenester deler én database. Når ordretjenesten læser og skriver direkte i lagerets tabeller, bliver skemaet en grænseflade, som ingen ejer, intet team kan ændre det alene, og lagerbeholdningen får en ekstra skriver, som lagertjenestens kontroller aldrig ser. Giv hvert stykke data én ejer, sådan som

noten giver lageret én, og lad andre tjenester spørge ejeren eller læse dens beskeder. Når en opdatering spænder over flere tjenester, så brug en saga med kompenserende trin frem for en distribueret transaktion.

Ingen kan følge en forespørgsel. En forespørgsel, der krydser flere tjenester, efterlader et brudstykke af sin historie i hver tjenestes log. Lewis og Fowler giver overvågning i realtid en central plads, hvor tekniske tal som forespørgsler pr. sekund til databasen følges side om side med forretningstal som ordrer pr. minut. De advarer også om, at tjenester, der reagerer på hinandens hændelser, giver emergent adfærd, som overvågningen skal fange hurtigt. Spor hver forespørgsel. I den model, OpenTelemetry bruger, er et trace den vej, én forespørgsel tager gennem systemet, bygget af spans, der deler et trace-id og peger på deres forælder, og context propagation er det, der gør, at spans fra forskellige tjenester kan samles til ét trace. Send trace-konteksten med på hvert kald og med hver besked, også dem, der går gennem outbox-tabellen, ellers slutter sporet ved det første asynkrone hop.

Læs mere

- Microservices — James Lewis og Martin Fowler, 2014: de ni kendetegn, navnets historie, sidebjælkerne om størrelse og synkrone kald og argumenterne for at gå forsigtigt frem.
- How Do Committees Invent? — Melvin Conways artikel fra 1968 i Datamation, hvor et systems struktur kopierer kommunikationsstrukturen i den organisation, der designede det.
- Domain-Driven Design Reference — Eric Evans' sammenfatning af mønstrene, herunder bounded context og kortene over, hvordan kontekster forholder sig til hinanden.
- OpenTelemetry: Traces — spans, trace-id'er, overordnede spans og context propagation, de brikker, der gør én forespørgsel synlig på tværs af tjenester.
- PostgreSQL: WITH Queries — datamodificerende sætninger i `WITH`, som lader én sætning ændre en række og registrere ændringen et andet sted.

Ordliste

Vælg ud fra jeres faktiske vilkår

Beskriv arbejdsgangen, de eksisterende grænseflader og de brugere, der har brug for ændringen. Vi kan definere en testbar leverance og en overdragelse, der gør teamet i stand til at vedligeholde løsningen.

Se, hvad det koster Kontakt os →

Gør begreberne praktiske

Spørg, hvordan »Hvad er microservices?« gælder i jeres organisation. Et konkret eksempel fra jeres nuværende system er nok til at begynde samtalen.

Se vores ydelser

welcome@engineer.company +45 50 10 61 84 København, Danmark

Tag denne side med dig

Visitkort

Lavet med kærlighed af D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – nu · Engineer ApS · Alle rettigheder forbeholdes.

Udskrevet fra https://engineer.company/da/glossary/microservices/