Engineer

Mikrotjenester

en arkitektur av små tjenester som hver rulles ut for seg

Hva det er

Mikrotjenester er en arkitekturstil som deler én applikasjon opp i små tjenester. Hver tjeneste er et selvstendig program som kjører som en egen prosess og kan rulles ut uten de andre. James Lewis og Martin Fowler, som i en artikkel fra 2014 ga stilen den mest kjente beskrivelsen, omtaler slike applikasjoner som «suites of independently deployable services». De lister opp hva slike systemer gjerne har felles: tjenestene snakker sammen over lette mekanismer, ofte et HTTP-API for ressurser; de bygges rundt forretningsevner; utrullingen er helautomatisert; den sentrale styringen er holdt på et minimum, og tjenestene kan være skrevet i ulike programmeringsspråk og bruke ulike teknologier for datalagring. De understreker selv at det ikke finnes noen formell definisjon, bare kjennetegn som de fleste systemer med denne merkelappen deler.

Ordet er yngre enn ideen. En fotnote i samme artikkel forteller at en workshop med programvarearkitekter i nærheten av Venezia diskuterte begrepet i mai 2011, og at den samme gruppen i mai 2012 landet på «microservices» som navn. Adrian Cockcroft i Netflix kalte tilnærmingen «fine grained SOA», og Lewis og Fowler hevder ikke at stilen er ny: de finner røttene minst så langt tilbake som prinsippene Unix ble bygd på.

Det stilen vanligvis måles mot, er monolitten: en applikasjon på serversiden som bygges og rulles ut som én enhet. All logikken for en forespørsel kjører i én prosess. Med litt omtanke kan hele applikasjonen kjøres og testes på den bærbare maskinen til én utvikler, og den skaleres ved å kjøre flere kopier bak en lastbalanserer. Friksjonen kommer senere. En liten endring betyr at alt må bygges og rulles ut på nytt, en god modulær struktur er vanskelig å holde ved like over år, og den delen som trenger mer kapasitet, kan bare skaleres sammen med alt det andre.

Lewis og Fowler definerer en komponent som en enhet av programvare som kan byttes ut og oppgraderes for seg. Et bibliotek lenkes inn i et program og kalles i minnet. En tjeneste er en komponent utenfor prosessen, som nås over nettverket med en webforespørsel eller et fjernkall. Forskjellen har betydning for koblingen. I de fleste språk er det bare dokumentasjon og disiplin som hindrer annen kode i å gripe inn i bibliotekets indre, mens en tjeneste bare kan nås gjennom eksplisitte fjernkall, så grensesnittet er langt vanskeligere å gå rundt. En tjeneste kan likevel bestå av flere prosesser, for eksempel en applikasjon sammen med en database som bare den bruker.

Forstavelsen «mikro» sier mindre enn det ser ut til. Lewis og Fowler mener at navnet trekker oppmerksomheten mot størrelse og fører til diskusjoner om hva som kan kalles «mikro». De største tjenestene de hadde hørt om, fulgte Amazons idé om et to-pizza-team, altså ikke flere enn et dusin personer, og i den andre enden av skalaen så de et team på et halvt dusin som hadde ansvaret for et halvt dusin tjenester.

Bruk begrepet i en beslutning

Bruk «Hva er mikrotjenester?» i en konkret beslutning. Vi kan knytte begrepet til systemet dere bygger eller kjøper. Delen brukerne aldri ser: hvor dataene ligger, hvilke regler som gjelder, og hvordan andre systemer får lov til å snakke med den.

Se tjenestene våre Ta kontakt →

Undersøk antakelsene bak ordet

Bruk forklaringen og det tilknyttede arbeidet til å formulere spørsmål for prosjektet. Vi kan undersøke antakelsene sammen før et verktøy velges. Designet og driftet 6 ETL/ELT-pipelines — 6 ETL-pipelines på tvers av tre databasemotorer

Designet og driftet 6 ETL/ELT-pipelines Ta kontakt →

Hvorfor det er viktig

Den første gevinsten er at hver tjeneste kan slippes i sitt eget tempo. Når en applikasjon er delt i tjenester, krever mange endringer bare at én tjeneste rulles ut på nytt, så ett team kan levere uten å vente på de andre. Lewis og Fowler er tydelige på at dette ikke er absolutt: en endring i grensesnittet til en tjeneste må fortsatt koordineres med tjenestene som kaller den.

Den andre er at programvaren kan følge organisasjonen i stedet for å kjempe mot den. I 1968 hevdet Melvin Conway i Datamation at organisasjoner som designer systemer, er tvunget til å lage design som kopierer deres egne kommunikasjonsstrukturer. Lewis og Fowler bruker dette på den vanlige måten å dele opp en stor applikasjon på, etter tekniske lag, med ett team for UI-et, ett for logikken på serversiden og ett for databasen. Da blir selv en enkel endring et

prosjekt på tvers av team, og logikken havner der et team tilfeldigvis har tilgang. Mikrotjenestestilen snur dette på hodet. Hver tjeneste dekker én forretningsevne, som ordrer, lager eller betaling, fra skjermbildet og helt ned til lagringen, og hører til et tverrfaglig team som har alle ferdighetene som trengs hele veien ned. Teamet bør eie tjenesten gjennom hele levetiden i stedet for å levere den fra seg når et prosjekt er ferdig, og Amazons variant av regelen er at den som bygger en tjeneste, også drifter den i produksjon.

Den tredje er at logikken blir i tjenestene. Lewis og Fowler oppsummerer preferansen som smart endpoints and dumb pipes: smarte endepunkter og enkle rør imellom. Tjenestene snakker sammen med HTTP-forespørsler mot ressurs-API-er, eller via en lett meldingsbuss som ikke gjør annet enn å rute meldinger, og avgjørelsene tas i tjenestene i hver ende. Det er et bevisst steg bort fra enterprise service bus, som ruter, transformerer og anvender forretningsregler midt i nettverket i stedet for i endene.

Data deles langs de samme linjene. Hver tjeneste styrer sin egen database, og ulike tjenester kan bruke ulike databasesystemer hvis det passer arbeidet deres. Eric Evans' domenedrevne design gir begrunnelsen: et stort system rommer flere modeller, og hver av dem trenger en eksplisitt avgrenset kontekst (bounded context), fastsatt ut fra hvordan teamene er organisert, hvilke deler av applikasjonen modellen gjelder for, og fysiske ting som kodebaser og databaseskjemaer. Da kan salgsavdelingens bilde av en kunde og kundeservicens bilde av den samme kunden være forskjellige, uten at noen må forene dem i én tabell.

Kostnadene er en del av definisjonen, og Lewis og Fowler sier dem rett ut. Et fjernkall koster langt mer enn et kall i minnet, så grensesnittene mellom tjenester må bli grovere. Det er tungt å flytte kode over en tjenestegrense: endringer i grensesnitt må koordineres, lag med bakoverkompatibilitet må legges til, og testingen blir mer komplisert. Distribuerte transaksjoner er vanskelige å implementere, så tjenestene koordinerer uten dem, godtar at dataene kanskje først blir konsistente etter en stund, og retter feil med kompenserende operasjoner. Ethvert kall kan feile. Og passer tjenestene ikke rent sammen, er kompleksiteten ikke borte; den har bare flyttet ut i koblingene mellom dem, der den er vanskeligere å se. Lewis og Fowler gjengir et argument de mener er rimelig: bygg en monolitt til å begynne med, gi den ryddige moduler, og del den først opp når den begynner å skape problemer. De legger til et forbehold: et grensesnitt som fungerer godt inne i én prosess, er sjelden et godt grensesnitt mellom tjenester.

Slik ser det ut i praksis

Notatet Å selge den siste billetten nøyaktig én gang tyr til en egen tjeneste først når én database ikke lenger er nok, og formulerer da regelen som holder designet sammen: hver vare skal ha én eneste eier av lagerbeholdningen, nemlig lagertjenesten, og ingen andre får ta avgjørelser om den. Tjenesten shardes etter leietaker og vare, og de største leietakerne får egne celler. Bare reservasjonen trenger å være synkron. Betaling, ordre og kvittering er separate systemer, og det finnes ingen felles transaksjon for dem. Notatet avviser tofase-commit, fordi den blokkerer så lenge en deltaker er treg og betalingsleverandører ikke deltar i den, og bruker i stedet en saga: en rekke lokale steg, hvert med et eget steg som omgjør det.

Hvert steg legger meldingen sin i en outbox-tabell, innenfor den transaksjonen som også gjør endringen meldingen varsler om. I PostgreSQL kan en datamodifiserende setning inne i `WITH` sende resultatet sitt videre til en innsetting, så lagertjenesten kan reservere en enhet og registrere meldingen i én og samme setning:

```
-- 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;
-- utsolgt: ingen rad treffer, så ingen melding skrives
```

Change data capture publiserer så radene fra outbox-tabellen. Siden en melding kan bli levert mer enn én gang, må hver mottaker være idempotent, med ordren som nøkkel. I denne skissen rører ingen annen tjeneste tabellen `inventory`: de andre får vite om beholdningen gjennom disse meldingene, eller ved å spørre lagertjenesten.

Det samme notatet viser også hvor oppdelingen hører hjemme i rekkefølgen. I listen over hva som skal gjøres først, kommer en egen tjeneste og en egen eier for lagerbeholdningen som nummer fire: etter måling, etter at skrivingen er gjort

om til én betinget setning, og etter at lasten er dempet med et venterom, en port og et hurtigbufret «utsolgt»-svar. De billigere grepene kommer først.

Porteføljeprosjektet En maritim plattform bygd i ryddige lag dokumenterer grunnleggerens arkitekturbidrag hos NextMariner og viser den andre siden av valget. Det var et ungt produkt der kravene kom til å endre seg mye, og målet var at en endret forretningsregel ikke skulle kreve endringer i frontenden, API-et og databasen samtidig. Løsningen ble tre lag med én oppgave hver: en Next.js-frontend for presentasjon, et bevisst tynt Go-API som ruter, validerer, filtrerer for sikkerhet og orkestrerer, og all forretningslogikk i PostgreSQL-funksjoner. Grensene ble skrevet ned og håndhevet i gjennomganger. Lewis og Fowler påpeker at modulære grenser inne i én applikasjon krever mye disiplin for å opprettholdes, og at det tydeligere skillet mellom tjenester gjør det lettere å holde teamgrensene klare. Denne plattformen beholdt disiplinen uten å dele forretningslogikken sin opp i separate tjenester.

Hvor det går galt

Kjeder av synkrone kall mangedobler nedetiden. Når svaret på én forespørsel krever at den første tjenesten kaller en andre, som kaller en tredje, lykkes forespørselen bare mens alle tre er oppe, og den venter på hver av dem etter tur. Lewis og Fowler kaller dette den multiplikative effekten av nedetid og viser til Guardians regel om ett synkront kall per brukerforespørsel. Hold den synkrone stien like kort som notatet holder den, der bare reservasjonen venter på svar og alt etter den er meldinger. Gi hvert kall som er igjen, en timeout og en circuit breaker, mønstrene Lewis og Fowler viser til i boka Release It!, slik at en tjeneste som feiler, blir koblet ut i stedet for å bli ventet på.

Tjenester som alltid endres sammen. En oppdeling langs feil linjer gir tjenester som ikke kan slippes hver for seg: hver ny funksjonalitet berører tre av dem, og de leveres i takt. Lewis og Fowler gir testen: endrer du stadig to tjenester samtidig, bør de slås sammen. En beslektet feil er å gjøre metodekall om til fjernkall én til én, slik at tjenestene prater frem og tilbake over nettverket og yter dårlig. Trekk grensene rundt forretningsevner og avgrensede kontekster, gjør grensesnittene grove, og lag hver tjeneste slik at den tåler endringer i tjenestene den kaller. Da kan en leverandør utvikle seg uten å ødelegge for konsumentene sine, og versjonering ser Lewis og Fowler på som en siste utvei.

To tjenester deler én database. Når ordretjenesten leser og skriver lagertabellene direkte, blir skjemaet et grensesnitt ingen eier: ingen av teamene kan endre det alene, og beholdningen får enda en skriver, en som lagertjenestens kontroller aldri ser. Gi alle data én eier, slik notatet gir beholdningen én, og la andre tjenester spørre eieren eller lese meldingene dens. Når en oppdatering spenner over flere tjenester, bruk en saga med kompenserende steg i stedet for en distribuert transaksjon.

Ingen kan følge en forespørsel. En forespørsel som går gjennom flere tjenester, etterlater en bit av historien sin i loggene til hver av dem. Lewis og Fowler gir overvåking i sanntid en sentral plass, der tekniske tall som forespørsler per sekund mot databasen følges side om side med forretningstall som ordrer per minutt. De advarer også om at tjenester som reagerer på hverandres hendelser, gir emergent atferd som overvåkingen må fange opp raskt. Spor hver forespørsel. I modellen OpenTelemetry bruker, er et spor veien én forespørsel tar gjennom systemet, bygd av spenn som deler en spor-ID og peker på forelderen sin, og kontekstpropagering er det som gjør at spenn fra ulike tjenester kan settes sammen til ett spor. Send sporkonteksten videre med hvert kall og hver melding, også dem som går gjennom outbox-tabellen, ellers stopper sporet ved det første asynkrone hoppet.

Les mer

- Microservices — James Lewis og Martin Fowler, 2014: de ni kjennetegnene, historien bak navnet, sidestoffet om størrelse og synkrone kall, og argumentene for å være forsiktig.
- How Do Committees Invent? — Melvin Conways artikkel fra Datamation i 1968, der strukturen i et system kopierer kommunikasjonsstrukturen i organisasjonen som designet det.
- Domain-Driven Design Reference — Eric Evans' sammendrag av mønstrene, blant dem avgrenset kontekst og kartene over hvordan kontekster henger sammen.
- OpenTelemetry: Traces — spenn, spor-ID-er, foreldrespenn og kontekstpropagering, delene som gjør én forespørsel synlig på tvers av tjenester.
- PostgreSQL: WITH Queries — datamodifiserende setninger i `WITH`, som lar én setning endre en rad og registrere endringen et annet sted.

Ordliste

Velg ut fra deres faktiske rammer

Beskriv arbeidsflyten, eksisterende grensesnitt og brukerne som trenger endringen. Vi kan definere en testbar leveranse og en overlevering som gjør teamet i stand til å vedlikeholde løsningen.

Se hva det koster Ta kontakt →

Gjør begrepene praktiske

Spør hvordan «Hva er mikrotjenester?» gjelder i organisasjonen deres. Et konkret eksempel fra dagens system er nok til å starte samtalen.

Se tjenestene våre

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

Ta med deg denne siden

Visittkort

Laget med kjærlighet av D.

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

Opphavsrett © 2025 – nå · Engineer ApS · Alle rettigheter forbeholdt.

Skrevet ut fra https://engineer.company/nb/glossary/microservices/