Engineer

Mikrotjänster

en arkitektur av små tjänster som driftsätts var för sig

Vad det är

Mikrotjänster är en arkitekturstil som delar upp en applikation i små tjänster. Varje tjänst är ett fristående program som körs som en egen process och kan driftsättas utan de andra. James Lewis och Martin Fowler, vars artikel från 2014 gav stilen dess mest kända beskrivning, kallar sådana applikationer ”suites of independently deployable services”. De räknar upp vad sådana system brukar ha gemensamt: tjänsterna pratar med varandra över lätta mekanismer, ofta ett HTTP-API för resurser; de byggs kring affärsförmågor; driftsättningen är helt automatiserad; den centrala styrningen hålls till ett minimum, och tjänsterna får skrivas i olika programspråk och använda olika tekniker för datalagring. De påpekar själva att det inte finns någon formell definition, bara kännetecken som de flesta system med den etiketten delar.

Ordet är yngre än idén. En fotnot i samma artikel berättar att en workshop med mjukvaruarkitekter nära Venedig diskuterade begreppet i maj 2011, och att samma grupp i maj 2012 bestämde sig för ”microservices” som namn. Adrian Cockcroft på Netflix kallade angreppssättet ”fine grained SOA”, och Lewis och Fowler hävdar inte att stilen är ny: de hittar rötterna åtminstone så långt tillbaka som principerna Unix byggdes på.

Det stilen oftast jämförs med är monoliten: en applikation på serversidan som byggs och driftsätts som en enda enhet. All logik för en förfrågan körs i en och samma process. Med lite omsorg kan hela applikationen köras och testas på en enskild utvecklares bärbara dator, och den skalas genom att fler kopior körs bakom en lastbalanserare. Friktionen kommer senare. En liten ändring innebär att allt måste byggas och driftsättas på nytt, en god modulär struktur är svår att bevara år efter år, och den del som behöver mer kapacitet kan bara skalas tillsammans med allt annat.

Lewis och Fowler definierar en komponent som en enhet av programvara som kan bytas ut och uppgraderas för sig. Ett bibliotek länkas in i ett program och anropas i minnet. En tjänst är en komponent utanför processen, som nås över nätverket med en webbförfrågan eller ett fjärranrop. Skillnaden spelar roll för kopplingen. I de flesta språk är det bara dokumentation och disciplin som hindrar annan kod från att peta i bibliotekets inre, medan en tjänst bara kan nås genom uttryckliga fjärranrop, så gränssnittet är betydligt svårare att gå runt. En tjänst kan ändå bestå av flera processer, till exempel en applikation tillsammans med en databas som bara den använder.

Förledet ”mikro” säger mindre än det verkar. Lewis och Fowler påpekar att namnet riktar uppmärksamheten mot storlek och leder till diskussioner om vad som räknas som ”mikro”. De största tjänsterna de hade hört talas om följde Amazons idé om ett team som två pizzor räcker till, alltså högst ett dussin personer, och i andra änden av skalan såg de ett team på ett halvdussin som tog hand om ett halvdussin tjänster.

Använd begreppet i ett beslut

Använd ”Vad är mikrotjänster?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. Den del användarna aldrig ser: där data lagras, vilka reglerna är och hur andra system får kommunicera med den.

Se våra tjänster Hör av er →

Granska antagandena bakom ordet

Använd förklaringen och det länkade arbetet för att formulera frågor till projektet. Vi kan undersöka antagandena tillsammans innan ett verktyg väljs. Utformade och drev 6 ETL/ELT-pipelines — 6 ETL-pipelines över tre databasmotorer

Utformade och drev 6 ETL/ELT-pipelines Hör av er →

Varför det spelar roll

Den första vinsten är att varje tjänst kan släppas i sin egen takt. När en applikation är uppdelad i tjänster kräver många ändringar bara att en tjänst driftsätts på nytt, så ett team kan leverera utan att vänta på de andra. Lewis och Fowler är tydliga med att det inte gäller fullt ut: en ändring i en tjänsts gränssnitt måste fortfarande samordnas med de tjänster som anropar den.

Den andra är att mjukvaran kan följa organisationen i stället för att kämpa emot den. År 1968 hävdade Melvin Conway i Datamation att organisationer som konstruerar system är tvungna att ta fram konstruktioner som efterliknar deras egna kommunikationsstrukturer. Lewis och Fowler tillämpar det på det vanliga sättet att dela upp en stor applikation, efter tekniska skikt, med ett team för UI:t, ett för logiken på serversidan och ett för databasen. Då blir även en enkel

ändring ett projekt över teamgränserna, och logiken hamnar där något team råkar ha åtkomst. Mikrotjänststilen vänder på det. Varje tjänst täcker en affärsförmåga, som order, lager eller betalning, från skärmen ända ned till lagringen, och tillhör ett tvärfunktionellt team som har alla kompetenser som behövs hela vägen ned. Teamet bör äga tjänsten under hela dess livstid i stället för att lämna över den när ett projekt är slut, och Amazons version av regeln är att den som bygger en tjänst också kör den i produktion.

Den tredje är att logiken stannar i tjänsterna. Lewis och Fowler sammanfattar hållningen som smart endpoints and dumb pipes: smarta ändpunkter och enkla rör emellan. Tjänsterna kommunicerar med HTTP-förfrågningar mot resurs-API:er, eller via en lätt meddelandebuss som inte gör något annat än att routa meddelanden, och besluten fattas i tjänsterna i vardera änden. Det är ett medvetet steg bort från en enterprise service bus, som routar, transformerar och tillämpar affärsregler mitt i nätverket i stället för i dess ändar.

Data delas upp längs samma linjer. Varje tjänst sköter sin egen databas, och olika tjänster kan använda olika databassystem om det passar deras arbete. Eric Evans domändrivna design står för motiveringen: ett stort system rymmer flera modeller, och var och en behöver en uttryckligt avgränsad kontext (bounded context), bestämd utifrån hur teamen är organiserade, vilka delar av applikationen modellen gäller för och fysiska saker som kodbaser och databasscheman. Säljavdelningens bild av en kund och kundtjänstens bild av samma kund kan då skilja sig åt, utan att någon behöver jämka ihop dem i en tabell.

Kostnaderna hör till definitionen, och Lewis och Fowler säger dem rakt ut. Ett fjärranrop kostar mycket mer än ett anrop i minnet, så gränssnitten mellan tjänster måste bli grövre. Det är besvärligt att flytta kod över en tjänstegräns: ändringar i gränssnitt måste samordnas, skikt för bakåtkompatibilitet måste läggas till och testningen blir krångligare. Distribuerade transaktioner är svåra att implementera, så tjänsterna samordnar sig utan dem, godtar att data kanske blir konsistenta först efter en stund och rättar till fel med kompenserande operationer. Vilket anrop som helst kan misslyckas. Och om tjänsterna inte passar ihop snyggt har komplexiteten inte försvunnit; den har bara flyttat ut i kopplingarna mellan dem, där den är svårare att se. Lewis och Fowler återger ett argument som de tycker är rimligt: bygg en monolit till att börja med, ge den rena moduler och dela upp den först när den börjar ställa till problem. De lägger till ett förbehåll: ett gränssnitt som fungerar bra inom en process blir sällan ett bra gränssnitt mellan tjänster.

Så ser det ut i praktiken

Anteckningen Att sälja den sista biljetten exakt en gång tar till en egen tjänst först när en databas inte längre räcker, och formulerar då regeln som håller ihop designen: varje vara ska ha en enda ägare av sitt lagersaldo, lagertjänsten, och ingen annan får fatta beslut om det. Tjänsten shardas på tenant och vara, och de största tenanterna får egna celler. Bara reservationen behöver vara synkron. Betalningen, ordern och kvittot är separata system, och ingen gemensam transaktion omfattar dem. Anteckningen avvisar tvåfas-commit, eftersom den blockerar så länge någon deltagare är långsam och betalningsleverantörer inte deltar i den, och använder i stället en saga: en följd av lokala steg, vart och ett med ett eget steg som ångrar det.

Varje steg lägger sitt meddelande i en outbox-tabell, inom samma transaktion som den ändring meddelandet berättar om. I PostgreSQL kan en datamodifierande sats inuti `WITH` skicka sitt resultat vidare till en insättning, så lagertjänsten kan reservera en enhet och registrera meddelandet i en och samma sats:

```
-- lagertjänsten, med en egen databas
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;
-- slutsålt: ingen rad matchar, så inget meddelande skrivs
```

Change data capture publicerar sedan raderna i outbox-tabellen. Eftersom ett meddelande kan levereras mer än en gång måste varje mottagare vara idempotent, med ordern som nyckel. I den här skissen rör ingen annan tjänst tabellen `inventory` : de andra får veta hur det står till med lagret genom de här meddelandena, eller genom att fråga lagertjänsten.

Samma anteckning visar också var uppdelningen hör hemma i arbetsordningen. I listan över vad som ska göras först kommer en egen tjänst och en egen ägare för lagret på fjärde plats: efter mätningen, efter att skrivningen har gjorts om till en

enda villkorad sats och efter att lasten har minskats med ett väntrum, en spärr och ett cachat ”slutsålt”-svar. De billigare åtgärderna kommer först.

Portföljfallet En sjöfartsplattform byggd i rena skikt dokumenterar grundarens arkitekturbidrag till NextMariner och visar den andra sidan av valet. Det var en ung produkt där kraven skulle komma att ändras en hel del, och målet var att en ändrad affärsregel inte skulle innebära att frontenden, API:et och databasen måste röras på en gång. Lösningen blev tre skikt med ett jobb var: en Next.js-frontend för presentationen, ett medvetet tunt Go-API som routar, validerar, filtrerar för säkerhet och orkestrerar, och all affärslogik i PostgreSQL-funktioner. Gränserna dokumenterades och upprätthölls i granskningen. Lewis och Fowler konstaterar att modulära gränser inom en applikation kräver mycket disciplin för att hålla, och att den tydligare uppdelningen i tjänster gör det lättare att hålla teamgränserna klara. Den här plattformen höll disciplinen utan att dela upp sin affärslogik i separata tjänster.

Var det går fel

Kedjor av synkrona anrop multiplicerar driftstopp. När svaret på en förfrågan kräver att den första tjänsten anropar en andra, som anropar en tredje, lyckas förfrågan bara så länge alla tre är uppe, och den väntar på var och en av dem i tur och ordning. Lewis och Fowler kallar det den multiplikativa effekten av driftstopp och hänvisar till Guardians regel om ett synkront anrop per användarförfrågan. Håll den synkrona vägen lika kort som anteckningen håller den, där bara reservationen väntar på svar och allt efter den är meddelanden. Ge varje anrop som återstår en timeout och en circuit breaker, de mönster Lewis och Fowler hänvisar till i boken Release It!, så att en tjänst som fallerar kopplas bort i stället för att bli väntad på.

Tjänster som alltid ändras tillsammans. En uppdelning längs fel linjer ger tjänster som inte kan släppas var för sig: varje ny funktion berör tre av dem, och de levereras i takt. Lewis och Fowler ger testet: om du gång på gång ändrar två tjänster samtidigt bör de slås ihop. Ett närliggande misstag är att göra om metodanrop till fjärranrop ett till ett, så att tjänsterna pratar fram och tillbaka över nätverket och presterar dåligt. Dra gränserna kring affärsförmågor och avgränsade kontexter, gör gränssnitten grova och bygg varje tjänst så att den tål ändringar i de tjänster den anropar. Då kan en leverantör utvecklas utan att ställa till det för sina konsumenter, och versionshantering ser Lewis och Fowler som en sista utväg.

Två tjänster delar en databas. När ordertjänsten läser och skriver lagertabellerna direkt blir schemat ett gränssnitt som ingen äger: inget av teamen kan ändra det på egen hand, och lagersaldot får ännu en skribent, en som lagertjänstens kontroller aldrig ser. Ge all data en ägare, så som anteckningen ger lagersaldot en, och låt andra tjänster fråga ägaren eller läsa dess meddelanden. När en uppdatering sträcker sig över flera tjänster, använd en saga med kompenserande steg i stället för en distribuerad transaktion.

Ingen kan följa en förfrågan. En förfrågan som passerar flera tjänster lämnar en bit av sin historia i loggarna hos var och en av dem. Lewis och Fowler ger övervakning i realtid en central plats, där tekniska mått som förfrågningar per sekund mot databasen följs sida vid sida med affärsmått som order per minut. De varnar också för att tjänster som reagerar på varandras händelser ger ett emergent beteende som övervakningen snabbt måste fånga upp. Spåra varje förfrågan. I den modell OpenTelemetry använder är ett spår den väg en förfrågan tar genom systemet, uppbyggt av span som delar ett spår-ID och pekar på sin förälder, och kontextpropagering är det som gör att span från olika tjänster kan fogas ihop till ett spår. Skicka spårkontexten vidare med varje anrop och varje meddelande, även dem som går via outbox-tabellen, annars tar spåret slut vid det första asynkrona hoppet.

Läs mer

- Microservices — James Lewis och Martin Fowler, 2014: de nio kännetecknen, historien bakom namnet, faktarutorna om storlek och synkrona anrop och skälen att gå försiktigt fram.
- How Do Committees Invent? — Melvin Conways artikel i Datamation från 1968, där ett systems struktur efterliknar kommunikationsstrukturen i organisationen som konstruerade det.
- Domain-Driven Design Reference — Eric Evans sammanfattning av mönstren, bland dem avgränsad kontext och kartorna över hur kontexter hänger ihop.
- OpenTelemetry: Traces — span, spår-ID, föräldraspan och kontextpropagering, delarna som gör en förfrågan synlig över tjänstegränserna.
- PostgreSQL: WITH Queries — datamodifierande satser i `WITH`, som låter en sats ändra en rad och registrera ändringen någon annanstans.

Ordlista

Välj utifrån era faktiska förutsättningar

Beskriv arbetsflödet, befintliga gränssnitt och användarna som behöver förändringen. Vi kan definiera en testbar leverans och en överlämning som gör att teamet kan underhålla lösningen.

Se vad det kostar Hör av er →

Gör begreppen praktiska

Fråga hur ”Vad är mikrotjänster?” gäller i er organisation. Ett konkret exempel från ert nuvarande system räcker för att börja samtalet.

Se våra tjänster

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

Ta med dig den här sidan

Visitkort

Omsorgsfullt skapad med kärlek av D.

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

Upphovsrätt © 2025 – nu · Engineer ApS · Alla rättigheter förbehållna.

Utskrivet från https://engineer.company/sv/glossary/microservices/