Engineer

REST

🏛 webens arkitekturstil, bygd rundt ressurser

Hva det er

REST, en forkortelse for Representational State Transfer, er arkitekturstilen Roy Fielding beskrev i kapittel 5 av doktorgradsavhandlingen sin fra 2000. Det er verken en protokoll eller en standard. Det er et sett begrensninger for hvordan en klient, en server og mellomleddene mellom dem kan samhandle, og Fielding gjør rede for hva hver begrensning gir, og for de flestes del også hva den koster. RESTful er adjektivet: strengt tatt betegner det et system som overholder alle begrensningene. I dagligtale strekkes begge ordene mye lenger, til nesten ethvert HTTP-API som utveksler JSON, enten det overholder begrensningene nedenfor eller ikke.

Det bærende begrepet er ressursen. Fielding definerer den vidt: all informasjon som kan gis et navn, kan være en ressurs, for eksempel et dokument, et bilde, en samling andre ressurser, en person, ja, til og med dagens vær i en bestemt by. Det som krysser nettverket, er aldri selve ressursen, men en representasjon, en sekvens av byte pluss metadata som beskriver de bytene, for eksempel et HTML-dokument eller et JPEG-bilde, og en klient handler på en ressurs ved å sende eller motta en representasjon av den nåværende eller tiltenkte tilstanden dens.

Fielding bygger opp stilen fra et tomt sett begrensninger og legger dem til én om gangen. Klient-server skiller brukergrensesnittet fra datalagringen, slik at de to kan utvikle seg hver for seg. Tilstandsløshet betyr at hver forespørsel inneholder alt som trengs for å forstå den, og ikke lener seg på noen kontekst lagret på serveren, slik at sesjonstilstanden ligger helt hos klienten. Cache krever at hvert svar er merket, eksplisitt eller implisitt, som cachebart eller ikke, slik at en klient kan gjenbruke et cachebart svar til en senere, tilsvarende forespørsel. Enhetlig grensesnitt, som er beskrevet nedenfor, gir alle komponenter det samme generelle grensesnittet. Lagdelt system betyr at hver komponent bare ser det laget den snakker direkte med, og det gir rom for proxyer, gatewayer og brannmurer imellom uten at grensesnittene på noen av sidene må endres.

Kode ved behov lar en server utvide en klient ved å sende den skript den kan kjøre, og Fielding gjør den til den eneste valgfrie begrensningen, fordi den reduserer synligheten.

Han kaller det enhetlige grensesnittet det sentrale trekket som skiller REST fra andre nettverksbaserte stiler, og deler det opp i fire begrensninger: ressurser identifiseres, på weben med URL-er; ressurser manipuleres gjennom representasjoner; meldinger er selvbeskrivende, slik at betydningen deres er synlig for ethvert mellomledd; og hypermedier er motoren i applikasjonstilstanden, noe som betyr at klienten beveger seg fra én tilstand til den neste ved å velge blant overgangene som tilbys i representasjonene den allerede har, slik et menneske følger lenker i en nettleser.

Bruk begrepet i en beslutning

Bruk «Hva er REST?» 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

Hver begrensning gir noe konkret tilbake. Tilstandsløshet lar et overvåkingssystem forstå en forespørsel uten å se ut over den, gjør det lettere å komme seg etter delvise feil og lar en server frigjøre ressurser så snart den har svart. En cache kan fjerne noen interaksjoner helt. Lag lar mellomledd fordele belastningen over flere nettverk og maskiner og lar én cache i kanten av en organisasjon betjene alle innenfor den. Og fordi det enhetlige grensesnittet frikobler implementasjonene fra tjenestene de leverer, kan klient og server hver for seg endre seg uten at den andre må følge med i samme takt.

Fielding legger like lite skjul på kostnadene. Tilstandsløshet gjentar i hver forespørsel data som en felles kontekst på serveren ville ha spart. En cache kan levere ut foreldede data. Hvert lag legger til overhead og latens. Det enhetlige grensesnittet svekker effektiviteten, fordi informasjonen overføres i en standardisert form i stedet for en som er skreddersydd for behovene i ett bestemt program. Fielding utformet det for overføring av hypermediedata i store biter, det vanlige tilfellet på weben, og sier rett ut at det ikke er optimalt

for andre former for samhandling. Det er det grunnlaget alternativene står på. En GraphQL-server tar imot alle forespørsler på ett endepunkt, identifiserer entitetene sine med noe annet enn URL-er og lar klienten velge nøyaktig hvilke felt som kommer tilbake, mens gRPC kaller metoder som er deklarert i en tjenestedefinisjon. Hver av dem gir opp en del av det enhetlige grensesnittet for å passe til en annen type trafikk.

For et HTTP-API følger mye av det enhetlige grensesnittet med fra start. HTTP definerer allerede metoder med en avtalt betydning, statuskoder og headerfelt for caching og for betingede forespørsler. Fordi den betydningen er felles, kan en proxy eller cache underveis handle på en melding uten å vite noe om det aktuelle API-et, og et API som bruker disse delene slik de er definert, får det maskineriet gratis. Et API som gir klientene sine URL-maler å fylle ut i stedet for lenker å følge, gir opp hypermediebegrensningen, og klientene blir bundet til URL-strukturen på serveren: når strukturen endres, går de i stykker, og det er nettopp den koblingen det enhetlige grensesnittet finnes for å fjerne.

Slik ser det ut i praksis

Utvekslingen nedenfor oppretter en ordre og får som svar adressen til den nye ressursen i `Location` og en entity tag i `ETag`. RFC 9110 bruker selv en `ETag` i et 201-svar som eksempel: taggen kan brukes som validator i senere betingede forespørsler, slik at to klienter ikke i det stille kan overskrive hverandres endringer.

```
POST /orders HTTP/1.1
Host: api.example.com
Content-Type: application/json

{"item": "notebook", "quantity": 2}

HTTP/1.1 201 Created
Location: /orders/7421
ETag: "v1"
Content-Type: application/json

{"id": 7421, "item": "notebook", "quantity": 2, "status": "open"}
```

Hver metode har en betydning som ethvert mellomledd kan stole på. RFC 9110 definerer GET, HEAD, OPTIONS og TRACE som sikre: klienten verken ber om eller forventer noen endring av tilstanden på serveren. PUT og DELETE er sammen med de sikre metodene idempotente, slik at flere like forespørsler har den

samme tiltenkte virkningen som én. POST er ingen av delene, og RFC 5789 sier det samme om PATCH. Det er de to egenskapene som lar en søkerobot følge lenker uten å gjøre skade, lar en cache hente på forhånd og lar en klient sende en PUT på nytt når forbindelsen brøt før svaret kom fram.

Et REST-API kan styres med ingenting annet enn en generisk HTTP-klient, og her på nettstedet ser man både hva det sparer, og hva det koster. Da en server nummer to ble klargjort fra kode, ble begge leverandørenes brannmurer skrevet direkte mot REST-API-ene deres gjennom en generisk HTTP-oppgave i stedet for gjennom en collection fra leverandøren. Det fjerner en avhengighet og versjonsdriften dens, og prisen er at formen på forespørslene må skrives for hånd. Samme side nevner en annen pris: den generiske HTTP-oppgaven deklarerer ikke støtte for check mode, så en tørrkjøring av playet som oppretter maskinen, hopper over hver oppgave i det.

Hvor det går galt

En GET endrer noe. En adresse som `/orders/7421/cancel`, som kansellerer en ordre når den hentes, bryter løftet om at GET er sikker. RFC 9110 forklarer hvorfor det løftet finnes: slik at søkeroboter og cacher som henter på forhånd, kan arbeide uten frykt for å gjøre skade. Den første søkeroboten eller forhåndshentende cachen som kommer over lenken, kansellerer ordren. Legg i stedet handlingen i metoden, som en POST til ordren eller en PATCH av statusen dens.

Hvert svar er 200 OK. Noen API-er returnerer 200 for alt og legger det egentlige utfallet, feil eller ikke, i meldingskroppen. Meldingene slutter å være selvbeskrivende: cacher, proxyer, overvåking og klientbiblioteker leser alle statuskoden, og samtlige ser suksess. RFC 9457 definerer en standardisert meldingskropp for feil, `application/problem+json`, med medlemmer som `type`, `title`, `status` og `detail`. `status`-medlemmet er bare veiledende: serveren må sende den samme statuskoden i selve HTTP-svaret, slik at generisk HTTP-programvare som ikke kjenner formatet, oppfører seg riktig.

En gjentatt POST oppretter to ordrer. Når en forbindelse brytes etter at serveren har behandlet en POST, men før klienten har lest svaret, kan ikke klienten vite om ordren finnes. RFC 9110 sier at en klient ikke bør gjenta en ikke-idempotent forespørsel automatisk med mindre den vet at forespørselen likevel er idempotent, eller kan se at det første forsøket aldri ble utført, og at en proxy slett ikke får gjøre det. Den påpeker også at noen klienter velger den mer

risikable veien og likevel gjentar en POST når forbindelsen lukkes før noe av svaret er kommet fram, særlig på en vedvarende forbindelse som har stått ubrukt. En utvei er å la klienten velge identifikatoren og sende ordren dit med PUT, fordi PUT er idempotent: den første forespørselen oppretter ordren med 201, og en gjentakelse erstatter den bare med den samme tilstanden.

To klienter overskriver hverandre. Begge leser ordren, begge endrer hvert sitt felt, og begge sender det hele tilbake med PUT: den andre skrivingen opphever i det stille den første. RFC 9110 kaller dette lost update-problemet. Løsningen er entity-taggen hver klient fikk da den leste ordren, sendt tilbake i `If-Match`. Serveren utfører da bare endringen hvis ressursen fortsatt er i den tilstanden klienten sist så; er den ikke det, får serveren ikke utføre den og kan si fra med 412 Precondition Failed. Klienten leser deretter ordren på nytt, legger inn endringen sin igjen og prøver en gang til.

```
PUT /orders/7421 HTTP/1.1
Host: api.example.com
If-Match: "v1"
Content-Type: application/json

{"item": "notebook", "quantity": 3, "status": "open"}

HTTP/1.1 412 Precondition Failed
```

Serveren husker samtalen. En server som holder tilstand per klient mellom forespørsler, for eksempel gjeldende side i en liste eller et halvt utfylt webskjema i minnet, har gitt opp tilstandsløsheten. En forespørsel gir ikke lenger mening alene, lastbalansereren må sende hver klient tilbake til den samme maskinen, og en omstart kaster tilstanden. Fieldings grunner til tilstandsløshet er nettopp de som går tapt: synlighet, enklere gjenoppretting etter delvise feil og en server som kan frigjøre ressurser når den har svart. Legg tilstanden i forespørselen, som et sidetoken i URL-en, eller gjør det halvferdige til en egen ressurs, for eksempel et ordreutkast, som enhver server kan lese.

Les mer

- Fielding, chapter 5: Representational State Transfer (REST) — definisjonen: hver begrensning, egenskapene den gir, og hva den ofrer for dem.
- RFC 9110: HTTP Semantics — metodene, hvilke av dem som er sikre og idempotente, statuskodene og betingede forespørsler med `ETag` og `If-Match`.

- RFC 5789: PATCH Method for HTTP — delvis endring av en ressurs, og hvorfor PATCH verken er sikker eller idempotent.
- RFC 9457: Problem Details for HTTP APIs — en standardisert JSON-kropp for feilsvar som bevarer betydningen av statuskoden.

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 REST?» 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/rest/