Engineer

immutable infrastructure

❄ å bytte ut servere i stedet for å endre dem på stedet

Hva det er

Immutable infrastructure er en måte å drifte servere på der en maskin aldri endres etter at den er satt i drift. På norsk ser man også betegnelsen uforanderlig infrastruktur. HashiCorps Well-Architected Framework beskriver det som infrastruktur som aldri endres etter at den er satt i produksjon, men bare byttes ut. En ny versjon av applikasjonen, et oppgradert bibliotek, en sikkerhetsoppdatering og en endret innstilling kommer alle samme vei: som en ny maskin eller container, bygget fra et nytt image, som tar plassen til den gamle. Deretter slås den gamle av og slettes.

Det motsatte er mutable infrastructure, der en server lever lenge og oppdateres der den står. I juni 2013 beskrev Chad Fowler hva en slik server blir til etter år med oppgraderinger: patchet gang på gang, med konfigurasjonsendringer gjort for hånd midt under et driftsavbrudd og med cron-jobber som bare én person vet om, helt til teamet verken tør å endre den eller å bytte den ut. Løsningen hans, som han kalte immutable deployments, var å bygge hvert system automatisk og deretter la det være urørt. En oppgradering eller en ny versjon betyr da å bygge en ny server eller et nytt image og kaste den gamle.

To typer image gjør tilnærmingen praktisk mulig. Et maskinimage er ifølge Packers dokumentasjon én statisk enhet med et forhåndskonfigurert operativsystem og den installerte programvaren. Packer selv er et community-verktøy som lager tilsvarende maskinimager for flere plattformer fra én kildekonfigurasjon. Containerimager er den andre typen, og Dockers dokumentasjon sier det rett ut: når et image først er laget, kan det ikke endres, og man kan bare lage et nytt image eller legge endringer oppå som nye lag.

Begrepet beskriver en måte å drifte et system på, ikke et produkt man installerer. Det har også en grense. Applikasjonsserverne kan byttes ut ved hver utgivelse, men dataene de arbeider med, må ligge et sted som overlever hver utskiftning, og det er ved den grensen det meste av det praktiske designarbeidet ligger.

Bruk begrepet i en beslutning

Bruk «Hva er immutable infrastructure?» i en konkret beslutning. Vi kan knytte begrepet til systemet dere bygger eller kjøper. Hele oppsettet skrevet ned som filer, slik at det kan gjenskapes nøyaktig og aldri bare finnes i hodet på én person.

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. Skrev 1 000 RBAC-regler for å sikre Linux-servere — 1,000 tilgangsregler skrevet for å tette et sikkerhetshull

Skrev 1 000 RBAC-regler for å sikre Linux-servere Ta kontakt →

Hvorfor det er viktig

Den første grunnen er avdrift. HashiCorps rammeverk sier det rett ut: manuelle oppdateringer og endringer på stedet får servere til å gli bort fra den opprinnelige konfigurasjonen sin over tid. To servere som ble satt opp likt med et års mellomrom, ender ikke nødvendigvis like, fordi hver av dem har fått sin egen rekke med rettelser, nye forsøk og håndredigeringer. Når hver server er en fersk kopi av ett image, er imaget den fullstendige beskrivelsen av serveren, og det er ingenting igjen som kan gli.

Den andre grunnen er tilbakerulling. Hvis en utgivelse oppfører seg feil, er rammeverkets svar å sette det forrige artefaktet i produksjon på nytt i stedet for å reparere den nåværende infrastrukturen. Det forrige imaget er en kjent tilstand som allerede har kjørt i produksjon, så teamet slipper å gjenskape under press hvordan serveren så ut i går, ut fra hukommelsen og shell-historikken.

Den tredje grunnen er færre hender på produksjon. Rammeverket påpeker at det å logge inn på produksjonsservere og oppdatere dem for hånd gir menneskelige feil som skrivefeil og glemte trinn, og at immutable infrastructure fjerner behovet for manuell tilgang til serverne. Færre personer trenger i det hele tatt en innlogging på en kjørende maskin, og de som har en, trenger sjelden å bruke den.

Den fjerde grunnen er sporbarhet. Revisorer vil ifølge samme side ha bevis for at produksjon ikke er endret utenfor de godkjente prosessene, og manuelle endringer etterlater ufullstendige spor. Når hver endring er et nytt artefakt, viser

versjonskontrollen og CI/CD-loggene hva som ble endret, når, og hvem som godkjente det. En sikkerhetsoppdatering følger samme vei som en ny funksjon: den blir et nytt artefakt som erstatter det som kjører, så den bygges, testes og registreres på samme måte.

Ingenting av dette er gratis. Rammeverket bemerker at images for virtuelle maskiner krever mer ressurser enn containere og bruker lengre tid på å starte, mens containere krever en orkestrator for å kjøre. Hver endring, også en innstilling på én linje, betyr nå et fullt bygg og en utskiftning, så bygget må være automatisk og raskt nok til at ingen fristes til å hoppe over det. Tilstanden må flyttes bort fra maskinene som byttes ut, før den første utskiftningen, ikke etterpå.

Slik ser det ut i praksis

En Packer-template beskriver hvordan et maskinimage bygges. Denne bygger på eksemplet i dokumentasjonen til Packers Amazon EBS-builder, med en `build`-blokk lagt til i den formen som referansen for build-blokken viser:

```
source "amazon-ebs" "web" {
  access_key    = var.aws_access_key
  secret_key    = var.aws_secret_key
  region        = "us-east-1"
  source_ami    = "ami-fce3c696"
  instance_type = "t2.micro"
  ssh_username  = "ubuntu"
  ami_name      = "web {{timestamp}}"
}

build {
  sources = ["source.amazon-ebs.web"]

  provisioner "shell" {
    scripts = fileset(".", "scripts/{install,secure}.sh")
  }
}
```

`source`-blokken angir builderen og basisimaget, `source_ami`, der rotvolumet kopieres og provisjoneres på en kjørende instans. `build`-blokken sier hvilke kilder som skal bygges, og hvordan de skal provisjoneres. Shell-provisjonereren kjører de to skriptene inne i den kjørende maskinen før den gjøres om til

et statisk artefakt, og artefaktet er et nytt image. Deklarasjonene av de to variablene er utelatt her. Én kommando kjører alle byggene i templaten og lager artefaktene:

```
packer build web.pkr.hcl
```

Produksjonssettingen er den andre halvdelen. HashiCorps rammeverk stiller de to verktøyene side om side: images bygges med Packer, settes i produksjon med Terraform og byttes ut helt når de trenger en oppdatering. Når en endring ikke kan gjøres på stedet, er Terraforms standard ifølge referansen for lifecycle å slette det eksisterende objektet og deretter opprette erstatningen. Oppdateringer i denne flyten krever ingen innlogging på en kjørende produksjonsserver. En endring i skriptene gir et nytt image, imaget gir nye maskiner, og de gamle fjernes.

Samme disiplin fungerer også på én enkelt vert. Porteføljeeksemplet om containere på Podman og Quadlet holder provisjonering og utrulling adskilt. Root forbereder stien for utrulling én gang, og deretter tar en vanlig bruker uten særlige rettigheter seg av utgivelsene, med lov til å kjøre nøyaktig ett skript og ingenting annet. En utgivelse er et nytt image fulgt av en omstart av tjenesten, uten root-skall og uten fritt valgte kommandoer. Eksemplet om CI/CD med GitHub Actions går lenger på image-siden: frontenden i produksjon kjører på et distroless-grunnlag som verken har skall eller pakkebehandler, og det er arbeidsflytene, ikke en person, som fører et bygg videre fra ett miljø til det neste. Et slikt image har nesten ingenting å logge inn på og ikke noe verktøy å installere med, så å bygge en etterfølger er den eneste fornuftige måten å endre det på.

Siden om servere du kan bygge opp igjen fra bunnen av viser den andre modellen utført med omhu. Ansible bringer en vert som er i drift, til tilstanden repoet beskriver, hvert play er idempotent, og repoet inneholder konfigurasjonen for å bygge opp det deklarerte miljøet på nytt. Siden legger ikke skjul på hva endringer på stedet koster. Alt tar lengre tid enn en direkte retting på serveren, og en kjøring som stopper halvveis, etterlater verten i dårligere stand enn en kjøring som avviser å begynne. Derfor kjører et preflight-play nå foran den. En vert som bare er halvt endret, er et skritt på veien mot den skjøre serveren Fowler beskrev, og å bytte ut maskinen i stedet for å endre den er den andre veien utenom.

Hvor det går galt

Modellen er enkel å formulere og lett å uthule. De fleste feil skyldes ett unntak som virket harmløst da det ble gjort.

En rask rettelse over SSH bringer avdriften tilbake. Under press logger noen inn på en kjørende server og retter en fil. Rettelsen virker til neste produksjonssetting bytter ut maskinen, og da er den borte sammen med enhver forklaring på hvorfor den trengtes. Fram til da stemmer den kjørende serveren ikke lenger med imaget sitt, og det er nettopp den tilstanden tilnærmingen skal hindre. Løsningen er å sende hver endring, også den som haster, gjennom image-bygget og holde bygget raskt nok til å være den raskeste veien. HashiCorps veiledning er aldri å logge inn på kjørende servere for å gjøre manuelle endringer eller kjøre oppdateringskommandoer.

Data som skrives på maskinen, slettes sammen med den. Dockers dokumentasjon sier at når en container slettes, forsvinner også filendringene som er gjort inne i den. Opplastinger, genererte filer eller en lokal database på en maskin som byttes ut, forsvinner ved neste utgivelse. Løsningen er å holde tilstanden utenfor den delen som byttes ut. Docker-volumer lagrer data ut over levetiden til den enkelte containeren, og en ny container som kobles til samme volum, finner filene igjen; for maskiner fyller en separat disk eller en databasetjeneste utenfor imaget samme rolle.

Utskiftningen tar tjenesten ned. Terraforms standardrekkefølge er først å slette det eksisterende objektet og deretter opprette erstatningen, og det etterlater et vindu der tjenesten står stille. Innstillingen `create_before_destroy` i `lifecycle` -blokken til en ressurs snur rekkefølgen:

```
resource "aws_instance" "web" {
  # ...
  lifecycle {
    create_before_destroy = true
  }
}
```

HashiCorp gjør dette til et tilvalg fordi mange objekttyper har krav om unike navn eller andre begrensninger som to eksemplarer samtidig ville bryte, og referansen ber deg forstå begrensningene for hver ressurstype før du stoler på det.

Et nytt bygg starter i det stille fra en nyere base. I stedet for en fast `source_ami` kan Amazon EBS-builderen fylle den ut fra et `source_ami_filter`, og med `most_recent` satt til true velger filteret det nyeste imaget som oppfyller kriteriene, noe dokumentasjonen kaller mest nyttig for et daglig distro-bygg. Det er praktisk hvis man vil holde seg oppdatert, men da starter to bygg av samme template på ulike dager fra ulike baser, og en gammel utgivelse blir ikke den samme når den bygges på nytt. Løsningen for utgivelsesimager er å låse `source_ami` til et bestemt image og flytte det fram med vilje, i en commit som er gjennomgått.

Hemmeligheter blir bakt inn i imaget. HashiCorps rammeverk er klart: artefakter må aldri inneholde hemmeligheter som databasepassord, API-nøkler eller krypteringsnøkler, fordi alle med tilgang til artefaktet kan hente dem ut, og fordi rotasjon av en tilgangsopplysning krever at hvert artefakt bygges og settes i produksjon på nytt. Løsningen er å hente hemmelighetene når maskinen eller containeren starter. HashiCorps eget mønster bruker Vault: en maskin som starter, autentiserer seg og får midlertidige tilgangsopplysninger som utløper, og tilgangsopplysningene roteres i Vault uten at noe må bygges på nytt. Templaten ovenfor følger samme regel for sine egne nøkler, som kommer inn som variabler og ikke står som tekst i filen.

Det forrige artefaktet er borte når det trengs. Tilbakerulling betyr å sette det forrige artefaktet i produksjon på nytt, og det forutsetter at det fortsatt finnes. En oppryddingsjobb som sletter gamle imager, eller en vane med å gjenbruke ett image-navn for hvert bygg, gjør en tilbakerulling til en gjenoppbygging fra gammel kildekode på en base som kan ha flyttet seg. Løsningen er å beholde imagene fra de siste utgivelsene som har kjørt i produksjon, gi hvert av dem et navn som aldri gjenbrukes, og øve på en tilbakerulling til et av dem før dagen det gjelder.

Les mer

- Create immutable infrastructure — Well-Architected Framework om avdrift, tilbakerulling, image-typer og hemmeligheter.
- Trash Your Servers and Burn Your Code: Immutable Infrastructure and Disposable Components — Chad Fowlers innlegg fra juni 2013, lest i en kopi hos Internet Archive.
- Introduction to Packer — hva Packer bygger, og hva et maskinimage inneholder.
- Amazon EBS builder — builderen fra eksemplet, med `source_ami` og `source_ami_filter`.

- lifecycle meta-argument reference — Terraforms standard med slett og opprett, og `create_before_destroy`.
- Persisting container data — hvorfor filer i en container forsvinner med den, og hvordan volumer tar vare på dem.

Ordliste

Velg ut fra deres faktiske rammer

Ta med utrullingsløpet, driftsrisikoene og dem som har vakt. Vi kan avgrense en endring med kontroller, en vei tilbake og tydelig ansvar.

Se hva det koster Ta kontakt →

Gjør begrepene praktiske

Spør hvordan «Hva er immutable infrastructure?» 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/immutable-infrastructure/