# Infrastrcuture as Code

## Oppsummering

Forfattere:
Tor Ivar Melling

Eiere:
Forfatterne og Forskningsstiftelsen TISIP

## Innhold

## 1. INTRODUKSJON

I den moderne verden av programvareutvikling, har begrepet *Infrastructure as Code* (IaC) blitt sentralt i forbindelse med å effektivisere og automatisere administrasjonen av IT-infrastruktur. IaC representerer en tilnærming hvor infrastruktur - som servere, nettverk, lagringsenheter, og så videre - konfigureres og administreres gjennom kode, i stedet for manuelle prosesser eller interaktive konfigurasjonsverktøy. Denne praksisen fremmer konsistens, sporbar endringshåndtering og raskere utrullingsprosesser.

Organiseringen av prosjekter og repostitories (repos) i forbindelse med IaC er avgjørende for å sikre en smidig, skalerbar og sikker implementering av infrastrukturen. Dette innebærer etablering av klar struktur for hvordan kode og ressurser lagres, dokumenteres og oppdateres. Ved å adoptere beste praksiser som modularitet, versjonskontroll, grundig dokumentasjon, standardisering, og integrasjon med kontinuerlig integrasjon/kontinuerlig leveranse (CI/CD) prosesser, kan organisasjoner oppnå mer robust og fleksible IT-systemer.

Sikkerhet og overholdelse av forskrifter er også sentralt i IaC. Ved å bake inn sikkerhetspraksiser og compliance-krav inn i kodebasen fra begynnelsen, reduseres risikoen for menneskelige feil og sikkerhetsbrudd. En effektiv feilhåndtering og logging er avgjørende for å opprettholde høy tilgjengelighet og ytelse av infrastrukturen.

Organisering av prosjekter og repo i IaC er ikke bare en teknisk øvelse, men også en strategisk tilnærming som krever nøye planlegging og vedvarende forvaltning. Den vellykkede implementeringen av IaC kan fundamentalt transformere måten IT-avdelinger opererer, noe som fører til høyere effektivitet og kanskje til og med bedre innovasjonsevne.

## 2. ORGANISERING AV KODE OG PROSJEKTER

Når det gjelder IaC, er organiseringen av kode i repositorier en viktig beslutning som påvirker mange aspekter av prosjektstyring, samarbeid, og implementering. Et sentral repository for alt, også kjent som en monorepo, innebærer at all kode, for alle prosjekter og tjenester, lagres i et enkelt, samlet repo. På den annen side representerer microrepo-strategien en tilnærming hvor hvert prosjekt eller tjeneste har sitt eget dedikerte repo.

### 2.1. Mono-repo

Bruken av enkelt repo har flere fordeler, blant annet forenkler det avhengighetshåndtering, ettersom alle prosjekter deler samme kodebase og dermed kan bruke samme sett av biblioteker og verktøy. Dette kan også forenkle prosessene for kontinuerlig integrasjon og leveranse (CI/CD), siden alle endringer og tester skjer i samme kontekst. En annen fordel er at det fremmer samarbeid på tvers av team, da alle har tilgang til hele kodebasen og lettere kan se hvordan ulike deler av systemet interagerer med hverandre.

På den andre siden kan et mono-repo bli ganske stor og uhåndterlig, spesielt i store organisasjoner med mange prosjekter. Dette kan føre til utfordringer med ytelse i versjonskontrollsystemer og kan gjøre det vanskeligere å vedlikeholde og å holde oversikt. Det kan medføre risiko for "kodebase-forurensning", hvor endringer i én del av systemet utilsiktet påvirker andre deler.

### 2.2. Micro-repo

Microrepo-tilnærmingen, derimot, involverer å ha separate repos for hvert prosjekt eller tjeneste. Dette bidrar til å holde kodebasene små og håndterlige. Det forenkler også tilgangskontroll og sikkerhetsstyring, ettersom man kan gi adgang til kun relevante repos for hvert team eller individ. En slik tilnærming kan også redusere risikoen for utilsiktet påvirkning mellom prosjekter, siden hver kodebase er isolert.

Men denne tilnærmingen har sine egne ulemper. Det kan være mer komplisert å håndtere avhengigheter mellom prosjekter, og det kan være mer utfordrende å implementere endringer som påvirker flere prosjekter samtidig. Det krever også mer arbeid med konfigurasjon og vedlikehold av flere CI/CD pipelines. Dessuten kan det gjøre samarbeid og deling av kode mellom team mer komplisert.

Valget mellom en monorepo og microrepo-tilnærming er avhengig av organisasjonens størrelse, struktur, og spesifikke behov. Et monorepo kan være mer effektiv for mindre organisasjoner eller de som verdsetter tett integrert samarbeid og enkelhet, kan en microrepo-strategi være bedre for større organisasjoner eller de som ønsker mer modulær og isolert utvikling. Det er viktig å vurdere både de tekniske og organisatoriske aspektene av hvert alternativ før man tar en beslutning.

# 3. CONTINUOUS TESTING AND DELIVERY (CI/CD)

CI/CD er en kritisk komponent i DevOps-praksiser, og spesielt relevant når det gjelder infrastruktur som kode. IaC er en tilnærming hvor infrastruktur - som servere, nettverk og lagring - konfigureres og administreres gjennom kode, fremfor manuelle prosesser. Terraform, et populært IaC-verktøy, som tillater brukere å definere og versjonere infrastruktur sikkert og effektivt ved hjelp av et deklarativt konfigurasjonsspråk. GitHub, på sin side, er et bredt brukt versjonskontrollsystem og en hostingplattform som støtter samarbeid og kodehåndtering.

## 3.1. Kontinuerlig testing i IaC med Terraform og GitHub

Kontinuerlig testing (CI) refererer til prosessen med å automatisk teste kode for hver endring som blir gjort, ofte hver gang kode blir pushet til et repo. I sammenhengen av Terraform og IaC, betyr dette å teste infrastrukturkonfigurasjoner for å sikre at de fungerer som forventet og ikke introduserer feil eller sikkerhetsproblemer.

- **Automatiserte Tester**: Dette kan omfatte enhetstester, integrasjonstester og akseptansetester (akseptansetest kan f.eks. være en kundes egen test for å verifisere at leveransen er i henhold til bestillingen). For Terraform, betyr det å validere at konfigurasjonsfiler er syntaktisk korrekte og oppfyller definerte krav. Verktøy som Terratest, tflint med fler kan brukes for å skrive og kjøre automatiserte tester på Terraform-koden.
- **Integrasjon med GitHub**: Ved å integrere Terraform med GitHub, kan kontinuerlige tester utløses automatisk hver gang kode blir pushet til et repo. GitHub Actions er et eksempel på et verktøy som kan konfigureres til å kjøre tester automatisk.

## 3.2. Kontinuerlig Levering i IaC med Terraform og GitHub

Kontinuerlig levering (CD) går et skritt videre ved ikke bare å teste kode kontinuerlig, men også ved å automatisk levere endringer til produksjon eller staging-miljøer etter at de har bestått tester.

- **Automatiser leveranser**: I forbindelse med Terraform, betyr dette å automatisk anvende godkjente og testede infrastrukturkonfigurasjoner til aktuelle miljøer. Dette sikrer at infrastruktur kan oppdateres og skaleres raskt og pålitelig.
- **Integrering med GitHub for CI/CD**: Ved å bruke GitHub Actions eller lignende CI/CD-verktøy, kan man konfigurere en arbeidsflyt der endringer automatisk testes og deretter anvendes til produksjon etter godkjenning. Dette reduserer manuell innsats og øker hastigheten og sikkerheten til utrullingsprosessen.

## 3.3. Sikkerhet og overvåking

I en verden hvor infrastruktur stadig endres og oppdateres, er det også viktig å sikre at sikkerhetspraksiser er integrert i den kontinuerlige testing og leveringsprosessen. Dette inkluderer kodegjennomgang, automatisert scanning for sikkerhetsproblemer, og overvåkning av infrastrukturen for å identifisere og reagere på sikkerhetsbrudd eller ytelsesproblemer.

CI/CD i IaC med Terraform og GitHub representerer en moderne tilnærming til infrastrukturforvaltning. Den integrerer kjent og god praksis fra programvareutvikling, som automatisert testing og kontinuerlig integrasjon/leveranse, inn i prosessen med å håndtere og oppdatere infrastruktur. Dette sikrer en mer effektiv, sikker og pålitelig infrastrukturadministrasjon, som er avgjørende i dagens raskt skiftende teknologilandskap.

# 4. DESIGNE FOR MODULARITET

Når vi snakker om infrastruktur som kode, er det viktig å anerkjenne hvordan vi strukturerer og organiserer koden vår. Designing for modularitet er en sentral del av denne prosessen. Dette innebærer en bevisst innsats for å dele opp komplekse infrastrukturkonfigurasjoner i mindre, mer håndterbare og gjenbrukbare deler. Med Terraform oppnår vi dette gjennom bruk av moduler. Disse modulene kan inneholde alt fra ressurser og variabler til konfigurasjonselementer, og de er utformet for å være gjenbrukbare over ulike prosjekter og miljøer.

Et kjennetegn på godt utformede komponenter i denne sammenhengen er deres evne til å stå alene. Hver komponent bør kunne fungere uavhengig av andre, noe som reduserer avhengigheter og gjør det enklere å forstå og vedlikeholde systemet. Gjenbrukbarhet er også nøkkelen her; en komponent bør kunne brukes i forskjellige deler av infrastrukturen uten behov for store endringer. Dette er hvor DRY-prinsippet (Don't Repeat Yourself) spiller en viktig rolle. Ved å unngå duplisering av kode og istedenfor fokusere på gjenbruk, sikrer en at systemet vårt er enklere å vedlikeholde og mindre utsatt for feil.

## 4.1. Terraform Modules - DRY – Don't Repeat Yourself

DRY-prinsippet oppfordrer til en strukturert tilnærming til kodeutvikling, hvor hver del av koden har en bestemt hensikt og unngår gjentakelse av kode. I Terraform betyr dette å definere infrastrukturkomponenter en gang og gjenbruke dem gjennom moduler, i stedet for å kopiere og lime inn lignende kode over flere steder. Dette fører ikke bare til enklere vedlikehold, men også enklere oppdateringer og mindre feil over tid.

For å håndtere disse modulene effektivt, kan en benytte seg av GitHub, eller lignende verktøy, som spiller en viktig rolle i IaC. Ved å bruke GitHub for versjonskontroll, kan vi holde oversikt over endringer, gjøre det enkelt å rulle tilbake til tidligere versjoner ved behov, og sikre at endringer er sporbare. Dette er avgjørende for et system der mange små endringer kontinuerlig blir gjort over tid. GitHub tilrettelegger også for samarbeid, slik at ulike team kan jobbe sammen på samme kodebase, dele moduler og infrastrukturkomponenter, og bidra til en samlet forståelse av systemet.

## 4.2. Hvor skal en sette grenser mellom komponenter?

Å sette grenser mellom komponenter i IaC handler om å identifisere og definere tydelige skillelinjer mellom forskjellige deler av infrastrukturen. Dette er en viktig del av det modulære designet, hvor målet er å skape systemer som er enkle å forstå, vedlikeholde, og utvikle videre.

En sentral del av denne prosessen er å finne "sømmer" i systemet. En søm i denne sammenhengen er et sted hvor du kan endre oppførselen i systemet ditt uten å faktisk måtte gjøre endringer i den delen av koden. Med andre ord, det er naturlige steder hvor man kan trekke grenser mellom ulike deler av systemet, og skape enkle, rene integrasjonspunkter. Dette tillater ulike komponenter å samhandle på en kontrollert måte, samtidig som det opprettholder deres individuelle uavhengighet.

I konteksten av Terraform, kan dette for eksempel bety å dele infrastrukturen inn i logiske enheter basert på funksjon eller bruk.

### 4.2.1.Nettverk og sikkerhet

I Azure kan en Terraform-modul bli brukt for å definere nettverksressurser. I Azure ville dette omfatte ressurser som Virtual Network (vNet), subnett, og Network Security Groups (NSG). En slik modul ville håndtere oppsettet av det virtuelle nettverket, definere subnett for segmentering av nettverket, og konfigurere NSG for å håndtere tilgangskontroll og sikkerhetsregler. Denne modulen ville være klart avgrenset fra andre deler av infrastrukturen, og gir et sikkert og godt definert nettverksmiljø for dine applikasjoner og tjenester.

### 4.2.2.Databaser

For databasetjenester i Azure, kan en annen Terraform-modul fokusere på å definere konfigurasjoner for Azure-spesifikke databasetjenester som Azure SQL Database eller Cosmos DB. Denne modulen håndterer opprettelse og konfigurasjon av databasetjenester, samt aspekter som ytelse, størrelse, og sikkerhetsinnstillinger. Selv om den er separat fra nettverksmodulen, vil det være integrasjonspunkter, som for eksempel nettverksendepunkter eller sikkerhetskonfigurasjoner som NSG-regler, for å sikre kommunikasjon og sikker tilgang mellom databasen og andre deler av infrastrukturen.

### 4.2.3.Applikasjonsservere

En tredje Terraform-modul i Azure-konteksten kan være fokusert på applikasjonsserver. Dette kan inneholde ressurser som Azure Virtual Machines (VM) for å kjøre serverapplikasjoner, eller Azure Kubernetes Service (AKS) for container-baserte applikasjoner. Denne modulen ville håndtere opprettelse og konfigurasjon av de nødvendige ressursene for å kjøre applikasjonene, inkludert VM-størrelser, nettverksinnstillinger og integrasjon med andre tjenester som lagring eller databaser. Selv om denne modulen ville samhandle med både nettverks- og databasemodulene, ville den ha sin egen unike omfang og ansvar for å sikre at applikasjonene kjører effektivt og sikkert.

I hver av disse eksemplene er nøkkelen med Terraform og Azure å utvikle selvstendige moduler som kan håndtere spesifikke aspekter av infrastrukturen, samtidig som de samhandler med andre moduler gjennom klare og veldefinerte grensesnitt. Dette bidrar til en mer organisert, vedlikeholdbar og skalerbar infrastruktur.

## 4.3. Grenser som støtter robusthet

Å skape grenser som støtter robusthet i IaC betyr å designe infrastrukturen slik at den kan håndtere og komme seg fra feil, uten at hele systemet blir påvirket. Dette innebærer å dele infrastrukturen i mindre, isolerte komponenter, slik at et problem i én del ikke sprer seg til hele systemet. Disse komponentene må kunne samhandle med hverandre, men også være i stand til å fungere uavhengig til en viss grad. Dette konseptet er kjent som "failure domain isolation".

La oss se på noen enkle eksempler ved hjelp av Terraform og Azure for å illustrere dette konseptet:

### 4.3.1.Isolerte nettverkssoner

I Azure kan du bruke Terraform til å definere og konfigurere Virtual Networks (vNets) med flere subnett. Disse subnettene kan representere isolerte nettverkssoner for forskjellige deler av applikasjonen din, for eksempel ett subnett for frontend-komponenter og et annet for backend-tjenester. Hver sone kan ha sine egne sikkerhetsregler og nettverkskontroller. Hvis det oppstår et sikkerhetsproblem eller en feil i ett subnett, vil det ikke nødvendigvis påvirke de andre subnettene, og dermed reduserer du risikoen for at hele systemet blir kompromittert eller går ned.

### 4.3.2.Skalerbare og robuste applikasjonstjenester

Med Terraform i Azure kan en opprette skalerbare og robuste applikasjonstjenester. For eksempel kan du bruke Azure Kubernetes Service (AKS) for å kjøre containeriserte applikasjoner. AKS støtter automatisk skalering og selvreparerende nodemiljøer, noe som betyr at hvis en node feiler, vil systemet automatisk erstatte den. Dette isolerer feilen til en enkelt node, i stedet for at hele applikasjonen blir påvirket. Videre kan du bruke Azure Load Balancer for å distribuere trafikk mellom ulike instanser av applikasjonen din, noe som gir høy tilgjengelighet og robusthet.

### 4.3.3.Desentralisert dataoppbevaring

Et annet aspekt ved å skape robusthet er å ha en desentralisert tilnærming til dataoppbevaring. Med Terraform kan du definere flere Azure SQL Database-instanser, hver konfigurert for ulike applikasjonsdeler. Denne tilnærmingen reduserer risikoen for at en enkelt feil i databasen påvirker hele applikasjonen. Videre kan du implementere replikering og backup-strategier for å sikre dataintegritet, selv i tilfelle av en feil.

Ved å skape grenser som støtter robusthet i IaC, sikrer vi at systemet vårt er bedre rustet til å håndtere feil og uforutsette hendelser. Gjennom bruk av Terraform og Azure, kan vi definere og implementere en infrastruktur som er både modulær og resilient, hvor feilisolering, automatisk skalering, og desentralisert dataoppbevaring bidrar til et mer robust og pålitelig system. Denne tilnærmingen er ikke bare viktig for å sikre kontinuerlig drift, men også for å opprettholde sikkerhet og ytelse på tross av uforutsette utfordringer.