Engineer

Repo

et lager som tar vare på lagrede versjoner av et sett filer

Hva det er

Et repo, kort for repository, er stedet der et versjonskontrollsystem holder hele historikken til et prosjekt: hver lagrede tilstand av hver fil, hvem som gjorde hver endring og hvorfor, og navnene som leder inn i den historikken. Arbeidsfilene du redigerer, er én visning av det; repoet er alt som ligger bak den visningen.

Git definerer ordet ganske presist i sin egen ordliste: en samling referanser (refs) sammen med en objektdatabase som inneholder alle objektene som kan nås fra disse referansene. Begge halvdelene er verdt å pakke ut. Objektdatabasen holder innholdet. Hver versjon av en fil, hver katalogliste og hver commit lagres som et objekt og får navn etter en hash av innholdet, noe Git-boken beskriver som et innholdsadressert filsystem: et nøkkel-verdi-lager der nøkkelen regnes ut fra verdien. To identiske filer er ett objekt, og enhver endring, uansett hvor liten, gir et nytt navn. Referansene er den menneskelige siden: grener og tagger, korte navn som hver peker på én commit og gir folk en vei inn.

De fleste repoer folk jobber i, har et arbeidstre knyttet til seg. Filene du åpner i en editor, ligger i prosjektkatalogen, og selve repoet bor ved siden av dem i en skjult `.git` -katalog. Et nakent repo (bare repository) har ikke noe arbeidstre i det hele tatt. Katalognavnet slutter etter konvensjon på `.git` , repoet holder bare historikken, og det er formen en server holder, fordi ingen redigerer filer der; folk pusher til det og henter fra det. Ordlisten til Git sier det samme: de som publiserer offentlige repoer, gjør som regel nakne repoer tilgjengelige. Git-boken viser hvordan du lager et fra et eksisterende prosjekt med `git clone --bare my_project my_project.git` .

Kloning er der designet til Git viser seg. En klone mottar nesten alle dataene serveren har: som standard kommer hver versjon av hver fil gjennom hele prosjektets historikk ned sammen med den. Hver klone er derfor et komplett

repo, som kan tre inn i stedet for serveren hvis serveren går tapt. Unntakene er små, men de betyr noe, og Git-boken nevner dem: hooks på serversiden og lignende innstillinger blir igjen.

To varianter dukker ofte opp. Et grunt repo (shallow repository), laget ved å klone med `--depth`, beholder bare de nyeste commitene; ordlisten sier at de eldre foreldrene er «cauterized away», og historikken kan gjøres dypere senere med en fetch. En submodul er et repo som ligger inne i et annet, med sin egen historikk, og som det ytre repoet låser til én bestemt commit.

Bruk begrepet i en beslutning

Bruk «Hva er et repo?» i en konkret beslutning. Vi kan knytte begrepet til systemet dere bygger eller kjøper. Testet, utgitt og mulig å reversere, slik at endringer når kundene en tirsdag ettermiddag i stedet for sent en fredagskveld.

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. Automatiserte GIS-SaaS-utrulling i GitHub Actions — produksjonssetting, data og rapportering, alt uten manuelle grep

Automatiserte GIS-SaaS-utrulling i GitHub Actions Ta kontakt →

Hvorfor det er viktig

Et repo er hukommelsen til et prosjekt. Når en kodelinje ser feil ut, sier historikken når den kom, i hvilken commit og med hvilken forklaring. Det er bare så godt som de commitene folk skriver, og derfor setter team som bryr seg om historikken sin, regler for commitmeldinger og håndhever dem med hooks i stedet for med god vilje.

Det er også enheten de fleste verktøy jobber på. Tilgang gis per repo, hooks kjører per repo, og kontinuerlig integrasjon starter som regel fra ett. Trekk grensen for et repo, og du har trukket grensen for hva én gjennomgang, én sjekk og én tillatelse dekker.

Repoer beholder det de får. En commit som sletter en fil, legger til en ny tilstand; den lar hver tidligere tilstand ligge der den var, i hver klone som allerede har den. Den varigheten er hele poenget med versjonskontroll, og den skjærer begge veier: en gammel versjon kan alltid hentes fram igjen, og det samme kan et passord noen committet ved en feil.

Til slutt kan et repo holde langt mer enn applikasjonskode. Infrastrukturdefinisjoner, dokumentasjon og konfigurasjon får alle det samme ut av det: en historikk, et steg for gjennomgang og en måte å gjenskape en kjent tilstand på. En server der oppsettet ligger i et repo, kan bygges opp igjen av noen som aldri har sett den.

Slik ser det ut i praksis

Fire repoer bygd over atten måneder av én person deler én commitstandard: et konvensjonelt prefiks som angir typen endring, et omfang, en emnelinje under en fast lengde, ingen emojier og ingen attribusjonslinjer. En hook for commitmeldinger i hvert repo der det gjøres arbeid, avviser alt som bryter med standarden, og det gjør standarden til en egenskap ved repoet og ikke ved den som committer. Historikken viser da hva arbeidet var. På det dokumenterte implementeringsstadiet i juli 2026 viste historikken i det største repoet 156 commits med funksjoner, 133 rettinger, 128 dokumentasjon, 81 vedlikehold, 26 refaktoreringer, 20 stil, 5 ytelse og 3 test. Siden nevner også grensen: formatet sjekkes, men innholdet gjør det ikke.

For selskapets egen infrastruktur, offentlig tilgjengelige tjenester som er satt i drift, og konfigurasjonen av andre komponenter var oppgaven å få dette beskrevet i et repo i stedet for i noens hode. Den dokumenterte implementeringen inneholdt 12 065 linjer YAML fordelt på 19 playbooks og 34 roller; tallene beskriver det stadiet, ikke det nåværende repoet. Automatisering av sikkerhetskopiering og e-post er konfigurert der, mens aktivering i drift er et separat trinn. Driftsmiljøet kan gjenskapes fra repoet: en vert tilhører en nivågruppe der variablene deklarerer rollene den kjører, og en konvergert vert rapporterer null endringer. Hver endring tar lengre tid enn en redigering direkte på serveren, og en preflight-play ble senere lagt inn foran konvergeringen.

Et tredje prosjekt viser hva et repo beholder når oppmerksomheten flytter seg et annet sted. Arbeid for et annet selskap fulgte med gjennom en omstrukturering av repoet, blant annet en kladdelogg med gyldige påloggingsopplysninger i klartekst, som lå i treet forbi alle sperrer gjennom det meste av repoets historikk. Det overlevde nettopp fordi ingenting av det kjørte. Svaret ble en omfangsregel øverst i repoets instruksjoner, ett selskaps infrastruktur og bare den, pluss karantene i stedet for sletting, slik at historikken forblir lesbar, mens hemmelighetsskanneren, vault-vakten og emoji-sjekken fortsatt leser filene i karantenen.

Hvor det går galt

- **En hemmelighet som er committet én gang, lever lenger enn slettingen.** Fjerner du filen i neste commit, ligger den fortsatt i den forrige og i hver klone som er laget siden. Behandle påloggingsopplysningen som offentlig: roter den først, og avgjør så om det er verdt å skrive om historikken, med bryderiet det gir alle som har en kopi.
- **En push treffer en utsjekket gren.** Å pushe inn i et repo som ikke er nakent, der den gjeldende grenen er den som oppdateres, ville satt arbeidstreet i utakt med historikken, så Git avviser en slik push som standard gjennom `receive.denyCurrentBranch`. Et repo folk pusher til, bør være nakent.
- **En grunn klone svarer på spørsmål om historikken ut fra en del av den.** En klone laget med `--depth` har bare de nyeste commitene, så `git log`, `git blame` og ethvert skript som utleder en versjon fra den siste taggen, ser bare den biten. Bygg- og utgivelsesjobber som trenger fortiden, bør hente hele historikken.
- **En lokal hook stopper ved neste klone.** Hooks på klientsiden ligger i `.git/hooks`, og Git-boken sier tydelig at en klone ikke kopierer dem. En regel som bare håndheves der, slutter å gjelde den dagen noen kloner på nytt. Hold hook-skriptene sporet i repoet og la `core.hooksPath` peke på dem, eller håndhev regelen på serversiden med `pre-receive`.
- **Store binærfiler gjør hver klone større for alltid.** Fordi en klone har med hver versjon av hver fil, blir hver erstattet kopi av en stor binærfil liggende i historikken, og hver ny klone laster ned alle sammen.
- **Filer som ingenting kjører, blir ikke lest.** En deaktivert oppgave, en utdatert inventarfil eller en kladdelogg ligger fortsatt i hver klone, men ingen test, intet bygg og ingen produksjonssetting rører den, så den slipper unna gjennomgang. Innholdet i repoet bør stemme med det prosjektet er ansvarlig for, og alt annet bør fjernes eller settes til side der sperrene fortsatt kan se det.

Les mer

- gitglossary: repository — Gits egen definisjon, ved siden av nakent repo, arbeidstre, grunt repo og submodul.
- Getting a Git Repository — de to måtene å få et på, å starte på nytt i en katalog eller å klone, og hva en klone tar med seg ned.
- Git Objects — hvordan objektdatabasen lagrer innhold under en hash av det samme innholdet.
- gitrepository-layout — hva hver fil og katalog inne i et repo er til for.
- Getting Git on a Server — å gjøre et eksisterende repo om til et nakent repo og dele det.

- Git Hooks — hooks på klientsiden og serversiden, og hvorfor en klone lar dem på klientsiden ligge igjen.

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 et repo?» 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/repository/