Engineer

Repository

Speicher für gespeicherte Versionen eines Dateibestands

Was es ist

Ein Repository, im Gespräch fast immer zu Repo verkürzt, ist der Ort, an dem eine Versionsverwaltung die vollständige Geschichte eines Projekts aufbewahrt: jeden aufgezeichneten Zustand jeder Datei, wer welche Änderung aus welchem Grund vorgenommen hat und die Namen, die in diese Geschichte führen. Die Arbeitsdateien, die Sie bearbeiten, zeigen einen Ausschnitt davon; das Repository ist alles hinter diesem Ausschnitt.

Git definiert den Begriff im eigenen Glossar recht genau: „Eine Sammlung von Referenzen zusammen mit einer Objektdatenbank, die alle von den Referenzen erreichbaren Objekte enthält." Beide Teile verdienen eine Erklärung. Die Objektdatenbank enthält die Inhalte. Jede Version einer Datei, jede Verzeichnisauflistung und jeder Commit werden als Objekte gespeichert und durch einen Hash ihres jeweiligen Inhalts benannt. Das Git-Buch beschreibt dies als inhaltsadressiertes Dateisystem: einen Schlüssel-Wert-Speicher, dessen Schlüssel aus dem Wert berechnet wird. Zwei identische Dateien sind ein einziges Objekt, und jede noch so kleine Änderung erzeugt einen neuen Namen. Die Referenzen sind die menschliche Seite: Branches und Tags, kurze Namen, die jeweils auf einen Commit zeigen und einen Einstieg ermöglichen.

Die meisten Repositories, in denen gearbeitet wird, haben einen zugehörigen Arbeitsbaum. Die Dateien im Editor liegen im Projektverzeichnis, das Repository selbst daneben in einem verborgenen Verzeichnis namens `.git`. Ein Bare-Repository hat überhaupt keinen Arbeitsbaum. Sein Verzeichnisname endet üblicherweise auf `.git`, und es enthält nur die Historie. So bewahrt ein Server ein Repository auf, weil dort niemand Dateien bearbeitet: Änderungen werden dorthin gepusht und von dort abgerufen. Das Git-Glossar sagt es ausdrücklich: Wer öffentliche Repositories veröffentlicht, stellt sie meist als Bare-Repositories bereit. Das Git-Buch zeigt, wie sich mit `git clone --bare my_project my_project.git` eines aus einem vorhandenen Projekt erzeugen lässt.

Beim Klonen zeigt sich der Entwurf von Git. Ein Klon erhält fast alle Daten des Servers: Standardmäßig lädt er jede Version jeder Datei aus der gesamten Projektgeschichte herunter. Jeder Klon ist deshalb ein vollständiges Repository und kann den Server ersetzen, wenn dieser verloren geht. Die Ausnahmen sind klein, aber wichtig, und das Git-Buch nennt sie: Serverseitige Hooks und ähnliche Einstellungen bleiben zurück.

Zwei Varianten kommen oft vor. Ein flaches Repository, das beim Klonen mit `--depth` entsteht, enthält nur die jüngeren Commits. Laut Glossar werden die älteren Eltern-Commits „wegkauterisiert“; später lässt sich die Historie durch einen Fetch vertiefen. Ein Submodul ist ein Repository innerhalb eines anderen, mit eigener Historie, das das äußere Repository auf genau einen Commit festlegt.

Das Konzept für eine Entscheidung nutzen

Nutzen Sie „Was ist ein Repository?“ für eine konkrete Entscheidung. Wir können helfen, das Konzept mit dem System in Verbindung zu bringen, das Sie entwickeln oder einkaufen. Geprüft, veröffentlicht und rückgängig zu machen: Änderungen erreichen Ihre Kunden am Dienstagnachmittag statt spät am Freitagabend.

Unsere Leistungen entdecken Kontakt aufnehmen →

Die Annahmen hinter dem Begriff prüfen

Nutzen Sie die Erklärung und die verlinkten Arbeiten, um Fragen für Ihr Projekt zu formulieren. Wir können die Annahmen gemeinsam mit Ihnen prüfen, bevor ein Werkzeug ausgewählt wird. GIS-SaaS-Deployments mit GitHub automatisiert — Bereitstellung, Daten und Berichte ohne Handarbeit

GIS-SaaS-Deployments mit GitHub automatisiert Kontakt aufnehmen →

Warum es wichtig ist

Ein Repository ist das Gedächtnis eines Projekts. Wenn eine Codezeile falsch aussieht, zeigt ihre Historie, wann sie hinzugekommen ist, in welchem Commit und mit welcher Erklärung. Das funktioniert nur so gut wie die geschriebenen Commits. Deshalb geben Teams, denen ihre Historie wichtig ist, Regeln für Commit-Nachrichten vor und setzen sie mit Hooks durch, statt auf guten Willen zu vertrauen.

Es ist auch die Einheit, auf der die meisten Werkzeuge arbeiten. Zugriffsrechte werden je Repository vergeben, Hooks laufen je Repository, und Continuous Integration beginnt meist bei einem Repository. Wer die Grenze eines Repositorys zieht, grenzt damit auch ab, welchen Bereich eine einzelne Begutachtung, eine Prüfung und eine Berechtigung jeweils abdecken.

Repositories behalten, was sie bekommen. Ein Commit, der eine Datei löscht, fügt einen neuen Zustand hinzu. Alle früheren Zustände bleiben, wo sie waren, in jedem Klon, der sie bereits enthält. Diese Beständigkeit ist der Zweck einer Versionsverwaltung, und sie wirkt in beide Richtungen: Eine alte Version lässt sich wiederherstellen, aber ebenso ein versehentlich eingechecktes Passwort.

Schließlich kann ein Repository weit mehr als Anwendungscode enthalten. Infrastrukturdefinitionen, Dokumentation und Konfiguration gewinnen dasselbe: eine Historie, einen Review-Schritt und einen Weg, einen bekannten Zustand wiederaufzubauen. Ein Server, dessen Einrichtung im Repository beschrieben ist, kann von jemandem neu aufgebaut werden, der ihn noch nie gesehen hat.

Wie es in der Praxis aussieht

Vier Repositories, die eine Person über achtzehn Monate aufgebaut hat, teilen einen Commit-Standard: ein konventionelles Präfix für die Änderungsart, einen Geltungsbereich, eine Betreffzeile unterhalb einer festen Längengrenze, keine Emojis und keine Attributionszeilen am Nachrichtenende. In jedem Repository, in dem Arbeit entsteht, lehnt ein Commit-Message-Hook jeden Verstoß ab. So ist der Standard eine Eigenschaft des Repositorys und nicht der Person, die committet. Die Historie zeigt anschließend, worin die Arbeit bestand. Zum dokumentierten Implementierungsstand vom Juli 2026 enthielt die Historie des größten Repositorys 156 Feature-Commits, 133 Fehlerkorrekturen, 128 Dokumentationsänderungen, 81 Wartungsarbeiten, 26 Refactorings, 20 Stiländerungen, 5 Performance-Änderungen und 3 Teständerungen. Der Hook endet vor der Qualitätsfrage: Er prüft das Format, nicht den Inhalt.

Für die eigene Infrastruktur des Unternehmens, die bereitgestellten öffentlich zugänglichen Dienste und die Konfiguration weiterer Komponenten lautete der Auftrag, sie in einem Repository zu beschreiben, statt sie nur im Kopf zu behalten. Die aufgezeichnete Implementierung umfasste 12.065 YAML-Zeilen, verteilt auf 19 Playbooks und 34 Rollen. Diese Zahlen beschreiben den damaligen Stand, nicht das heutige Repository. Auch die Automatisierung für Backups und E-Mail ist dort konfiguriert; ihre betriebliche Aktivierung ist ein eigener Schritt. Die Infrastruktur lässt sich aus dem Repository reproduzieren: Ein Host gehört zu einer Gruppe für seine Infrastrukturebene, deren Variablen seine auszuführenden Rollen festlegen, und ein bereits konvergierter Host meldet null Änderungen. Jede Änderung dauert länger als eine direkte Bearbeitung auf dem Server. Später wurde der Konvergenz ein Play mit Vorabprüfungen vorgeschaltet.

Ein Repository behält auch, was hineingeraten ist, während die Aufmerksamkeit anderswo lag. Arbeit für ein anderes Unternehmen wurde durch eine Umstrukturierung des Repositorys mitgeschleppt. Dazu gehörte ein provisorisches Protokoll mit noch gültigen Zugangsdaten im Klartext, das während des größten Teils der Repository-Historie unbemerkt an allen Kontrollen vorbeikam und im Baum blieb. Nichts davon wurde ausgeführt, und genau deshalb überlebte es. Die Antwort war eine Regel zum Geltungsbereich ganz oben in den Repository-Anweisungen: die Infrastruktur genau eines Unternehmens und nichts anderes. Hinzu kam eine Quarantäne statt einer Löschung. So bleibt die Historie lesbar, während Secret-Scanner, Vault-Kontrolle und Emoji-Prüfung die Dateien in Quarantäne weiterhin lesen.

Wo es schiefgeht

- **Ein einmal eingechecktes Geheimnis überlebt seine Löschung.** Wird die Datei im nächsten Commit entfernt, bleibt sie im vorigen und in jedem seither erstellten Klon erhalten. Behandeln Sie die Zugangsdaten als öffentlich: Tauschen Sie sie zuerst aus und entscheiden Sie dann, ob ein Umschreiben der Historie die Störungen für alle mit einer Kopie rechtfertigt.
- **Ein Push trifft auf einen ausgecheckten Branch.** Ein Push in ein Repository mit Arbeitsbaum, dessen aktueller Branch aktualisiert werden soll, würde dessen Arbeitsbaum und Historie auseinanderlaufen lassen. Git verweigert einen solchen Push deshalb standardmäßig über `receive.denyCurrentBranch`. Ein Repository, in das gepusht wird, sollte ein Bare-Repository sein.
- **Ein flacher Klon beantwortet Fragen zur Historie mit einem Ausschnitt.** Ein mit `--depth` erstellter Klon enthält nur jüngere Commits. Deshalb sehen `git log`, `git blame` und jedes Skript, das eine Version aus dem letzten Git-Tag ableitet, nur diesen Teil. Build- und Release-Jobs, die die Vergangenheit benötigen, sollten die vollständige Historie abrufen.
- **Ein lokaler Hook endet beim nächsten Klon.** Clientseitige Hooks liegen in `.git/hooks`, und das Git-Buch stellt klar, dass ein Klon sie nicht kopiert. Eine nur dort durchgesetzte Regel endet an dem Tag, an dem jemand frisch klont. Versionieren Sie die Hook-Skripte im Repository und lassen Sie `core.hooksPath` auf sie zeigen, oder erzwingen Sie die Regel serverseitig mit `pre-receive`.
- **Große Binärdateien blähen jeden Klon dauerhaft auf.** Weil ein Klon jede Version jeder Datei enthält, bleibt jede ersetzte Fassung einer großen Binärdatei in der Historie. Jeder neue Klon lädt sie alle herunter.

- **Dateien, die nirgends ausgeführt werden, liest niemand.** Eine deaktivierte Aufgabe, ein veraltetes Inventar oder ein provisorisches Protokoll liegt weiterhin in jedem Klon. Kein Test, Build oder Deployment berührt es, also entgeht es der Überprüfung. Der Inhalt eines Repositorys sollte zu den Verantwortlichkeiten des Projekts passen. Alles andere sollte entfernt oder so abgesondert werden, dass die Kontrollen es weiterhin sehen.

Weiterführende Lektüre

- gitglossary: Repository — Gits eigene Definition, neben Bare-Repository, Arbeitsbaum, flachem Repository und Submodul.
- Ein Git-Repository anlegen — Die beiden Wege, in einem Verzeichnis neu anzufangen oder zu klonen, und was ein Klon herunterlädt.
- Git-Objekte — Wie die Objektdatenbank Inhalte unter deren Hash speichert.
- gitrepository-layout — Wozu jede Datei und jedes Verzeichnis innerhalb eines Repositorys dient.
- Git auf einem Server einrichten — Ein vorhandenes Repository in ein Bare-Repository umwandeln und teilen.
- Git-Hooks — Client- und serverseitige Hooks und warum ein Klon die clientseitigen zurücklässt.

Glossar

Für Ihre tatsächlichen Rahmenbedingungen auswählen

Beschreiben Sie den Bereitstellungsweg, die Betriebsrisiken und die Bereitschaftsverantwortlichen. Wir können eine Änderung mit Prüfungen, einem Wiederherstellungsweg und klaren Zuständigkeiten abgrenzen.

Preise ansehen Kontakt aufnehmen →

Die Begriffe praktisch nutzbar machen

Fragen Sie, wie „Was ist ein Repository?“ auf Ihre Organisation zutrifft. Ein konkretes Beispiel aus Ihrem bestehenden System reicht aus, um das Gespräch zu beginnen.

Unsere Leistungen entdecken

welcome@engineer.company +45 50 10 61 84 Kopenhagen, Dänemark

Nehmen Sie diese Seite mit

Kontaktkarte

Mit Sorgfalt und Liebe gestaltet von D.

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

Copyright © 2025 – heute · Engineer ApS · Alle Rechte vorbehalten.

Gedruckt von https://engineer.company/de/glossary/repository/