Bernd Öggl
Michael Kofler

Git

Projektverwaltung für Entwickler und DevOps-Teams

- Git effektiv nutzen und sicher administrieren
- Features von GitHub und GitLab einsetzen
- Best Practices & Workflows für eigene Repositories

3., aktualisierte Auflage

Rheinwerk Computing

Vorwort

Immer wenn mehrere Personen gemeinsam an einem Softwareprojekt arbeiten, braucht es ein System, um alle durchgeführten Änderungen nachvollziehbar zu speichern. Ein derartiges Versionsverwaltungssystem gibt allen an der Entwicklung Beteiligten Zugriff auf das gesamte Projekt. Jeder Programmierer weiß, was die anderen zuletzt gemacht haben, jede Entwicklerin kann den Code der anderen ausprobieren und das Zusammenspiel mit ihren eigenen Änderungen testen.

In der Vergangenheit gab es viele Versionsverwaltungssysteme, z. B. CVS, Subversion (SVN) oder Visual SourceSafe. Im vergangenen Jahrzehnt hat sich aber Git zum neuen De-facto-Standard entwickelt, andere Programme spielen nur noch eine untergeordnete Rolle.

Einen wesentlichen Anteil an diesem Erfolg hatte die Webplattform GitHub, die den Einstieg in und die Nutzung von Git wesentlich vereinfachte. Unzählige Open-Source-Projekte nutzen das kostenlose Angebot GitHubs zum Projekt-Hosting. Kommerzielle Kunden, die den Quellcode nicht veröffentlichen wollten, zahlen für diesen Service. GitHub ist natürlich nicht die einzige Git-Plattform: Wichtige Konkurrenten sind z. B. GitLab, Azure DevOps und Bitbucket. Dessen ungeachtet kaufte Microsoft 2018 GitHub für 7,5 Milliarden US-$. Microsoft ahnte damals vermutlich schon, welchen Wert der Zugriff auf Milliarden Code-Dateien für das Training von KI-Sprachmodellen haben würde.

20 Jahre Git

Git entstand vor zwei Jahrzehnten, weil Linus Torvalds für die Weiterentwicklung des Linux-Kernels ein neues Versionsverwaltungssystem brauchte. Die Entwicklergemeinde hatte zuvor das Programm BitKeeper verwendet. Linus Torvalds war mit dem Programm grundsätzlich zufrieden, eine Lizenzänderung machte aber einen Wechsel erforderlich. Von den damals verfügbaren Open-Source-Programmen genügte keines seinen hohen Ansprüchen.

So stoppte der Linux-Chefentwickler kurzzeitig seine Hauptarbeit und schuf im April 2005 in nur zwei Wochen das Grundgerüst von Git. Der Name *Git* steht sinngemäß für *Blödmann* oder *Depp*, und auch die Hilfeseite `man git` bezeichnet das Programm als *the stupid content tracker*.

Was für ein Understatement das ist, wurde erst nach und nach klar, als Linus Torvalds die Weiterentwicklung von Git längst wieder aus der Hand gegeben hatte: Nicht

nur die Kernel-Entwickler stellten ihre Arbeit rasch und problemlos auf Git um, in den folgenden Jahren wechselten immer mehr Softwareprojekte auch außerhalb der Open-Source-Welt zu Git.

Den endgültigen Durchbruch schaffte Git, als sich Webplattformen wie GitHub und GitLab etablierten. Diese Websites vereinfachen das Hosting von Git-Projekten enorm und sind heute aus dem Git-Alltag nicht mehr wegzudenken. (Selbst der Linux-Kernel befindet sich mittlerweile auf GitHub!)

Ein bisschen ist das eine Ironie des Schicksals: Linus Torvalds wichtigstes Ziel beim Design von Git war es, ein dezentrales Versionsverwaltungssystem zu schaffen. Aber erst der zentralistische Ansatz von GitHub und Co. machte Git für Entwicklerinnen und Entwickler abseits der Guru-Liga richtig attraktiv.

Es gibt heute Stimmen, die die Bedeutung von Git ebenso hoch einschätzen wie die von Linux. Damit ist es Linus Torvalds gleich zwei Mal gelungen, einen Bereich des Software-Universums vollständig auf den Kopf zu stellen.

Jeder verwendet es, keiner versteht es

Bei aller Begeisterung für Git: Es ist unübersehbar, dass Git von Profis für Profis konzipiert wurde. Wir wollen in diesem Buch gar nicht erst den Eindruck erwecken, Git sei einfach. Das ist es nicht:

- Häufig führt nicht ein Weg zum Ziel, vielmehr gibt es mehrere Wege. Für die, die Git schon beherrschen, ist das nützlich; aber wenn Sie Git gerade lernen, verwirrt diese Vielfalt.
- Vielen Open-Source-Projekten wird der Vorwurf gemacht, sie seien schlecht dokumentiert. Das kann man bei Git wirklich nicht sagen. Im Gegenteil! Jedes Git-Kommando, jede Anwendungsmöglichkeit wird in `man`-Seiten sowie auf der Website *https://git-scm.com/docs* so ausführlich und mit allen erdenklichen Sonderfällen erläutert, dass man sich in den Details geradezu verliert.
- Erschwerend kommt hinzu, dass es ähnliche Begriffe mit unterschiedlichen Bedeutungen gibt, leicht zu verwechselnde Subkommandos, die stark voneinander abweichende Aufgaben erfüllen.

Wir geben es ganz offen zu: Trotz jahrelanger Git-Praxis haben wir beim Schreiben dieses Buchs noch eine Menge Details dazugelernt!

Über dieses Buch

Natürlich ist es möglich, Git sehr minimalistisch zu verwenden. Allerdings können kleine Abweichungen von der täglichen Routine dann zu überraschenden und oft unverständlichen Nebenwirkungen oder Fehlern führen.

Alle Git-Einsteigerinnen und -Einsteiger kennt das Gefühl, wenn ein Git-Kommando eine unverständliche Fehlermeldung liefert: Mit kaltem Schweiß überlegt man, ob man gerade das Repository für sämtliche Beteiligten nachhaltig zerstört hat und wen man bitten könnte, Git mit den richtigen Kommandos doch zur Weiterarbeit zu überreden.

Deswegen ist es nicht zielführend, Git zu beschreiben, ohne dabei in die Tiefe zu gehen. Erst ein gutes Verständnis für die Funktionsweise von Git gibt die notwendige Sicherheit, Merge-Konflikte oder andere Probleme sauber beheben zu können.

Gleichzeitig war uns aber klar, dass dieses Buch nur funktionieren kann, wenn wir den wesentlichen Funktionen den Vorrang geben. Trotz 400 Seiten ist dieses Buch *nicht* die allumfassende Anleitung zu Git, die auch den letzten Sonderfall berücksichtigt und jedes noch so exotische Git-Subkommando vorstellt. Wir haben daher in diesem Buch die Spreu vom Weizen getrennt.

Es ist in überschaubare Kapitel gegliedert, die Sie wie bei einem Bausteinsystem nach Bedarf lesen können:

- Nach einer kurzen Einführung (»Git in zehn Minuten«) führen wir in den Kapiteln »Learning by Doing«, »Git-Grundlagen« und schließlich »Datenanalyse im Git-Repository« in den Umgang mit Git ein. Dabei konzentrieren wir uns auf die Nutzung von Git auf Kommandoebene und gehen nur am Rande auf Plattformen wie GitHub oder auf andere Benutzeroberflächen ein.

  Git-Einsteigerinnen und -Einsteigern empfehlen wir, mit diesen vier Kapiteln zu starten. Selbst wenn Sie schon etwas Git-Erfahrung haben, sollten Sie sich unbedingt ein paar Stunden Zeit nehmen, um »Git-Grundlagen« zu lesen und einige der dort vorgestellten Techniken (Merging, Rebasing etc.) in einem Test-Repository auszuprobieren.

- Die folgenden drei Kapitel – »GitHub«, »GitLab« sowie »Azure DevOps, Bitbucket, Gitea und Gitolite« – stellen die wichtigsten Git-Plattformen vor. Gerade für komplexe Projekte bieten diese Plattformen nützliche Zusatzfunktionen, z. B. um automatische Tests durchzuführen oder um Continuous Integration zu implementieren.

  Selbstverständlich berücksichtigen wir auch den Fall, dass Sie Ihr Git-Repository selbst hosten möchten. Mit GitLab, Gitea oder Gitolite lässt sich dieser Wunsch relativ leicht realisieren.

- Damit wenden wir uns von den Grundlagen der Praxis zu: Im Kapitel »Workflows« zeigen wir populäre Muster, wie Sie die Arbeit vieler Entwicklerinnen und Entwickler mit Git in geordnete Bahnen (*Branches*) leiten.

  Im Kapitel »Arbeitstechniken« stehen fortgeschrittene Git-Funktionen im Vordergrund, z. B. Hooks, Submodule oder Subtrees.

»Git in der Praxis« zeigt, wie Sie auf Linux-Systemen Konfigurationsdateien (*Dotfiles*) oder das ganze `/etc`-Verzeichnis mit Git versionieren, wie Sie ein Projekt von SVN auf Git umstellen oder wie Sie eine simple Website schnell und einfach mit Git und Hugo realisieren.

»Gängige Probleme und ihre Lösungen« hilft Ihnen bei schwer verständlichen Fehlermeldungen aus der Sackgasse. Hier finden Sie auch Anleitungen, wie Sie Sonderwünsche realisieren – z. B. wie Sie einen Merge-Vorgang nur für eine ausgewählte Datei durchführen.

- Die »Kommandoreferenz« fasst in aller Kürze die wichtigsten Git-Kommandos und deren Optionen zusammen. Dabei haben wir uns vom Motto »weniger ist mehr« leiten lassen. Unser Ziel war nicht eine vollständige Referenz, sondern eine Art »Essenz von Git«.

Beispiel-Repositories

Einige Beispiele aus dem Buch stellen wir Ihnen auf GitHub zur Verfügung:

https://github.com/git-compendium

Liebe Leserin, lieber Leser!

Uns ist bewusst, dass Sie vielleicht nicht mit großer Freude die Lektüre dieses Buchs beginnen: Sie wollen oder müssen für ein Projekt Git verwenden. Aber Ihr Ziel ist nicht Git an sich, vielmehr wollen Sie Code produzieren, Ihr Projekt vorantreiben. Sie haben eigentlich weder Zeit noch Lust, sich mit Git zu beschäftigen – Sie wollen gerade so viel wissen, dass Sie Git fehlerfrei anwenden können.

Wir haben dafür Verständnis. Trotzdem empfehlen wir Ihnen dringend, ein paar Stunden mehr als geplant zu investieren, um Git systematisch kennenzulernen.

Wir versprechen Ihnen: Sie gewinnen diese Zeit später zurück! Zu wenig Git-Verständnis bedeutet zwangsläufig, dass Sie immer wieder im Internet nach der Lösung für ein gerade aufgetretenes Problem suchen müssen (oft unter Zeitdruck).

Auch wenn Sie aktuell primär Ihr Projekt im Fokus haben: Git-Kenntnisse sind langfristig eine Kernkompetenz, die Sie als Entwicklerin oder Entwickler in vielen zukünftigen Projekten brauchen werden! In diesem Sinne wünschen wir Ihnen viel Erfolg mit Git!

Michael Kofler (*https://kofler.info*)
Bernd Öggl (*https://webman.at*)

Kapitel 4
Datenanalyse im Git-Repository

In diesem Kapitel geht es darum, ein Repository gezielt nach Daten zu durchsuchen: Welche Dateien sind unter Versionskontrolle? In welchen Commits wurde eine Datei zuletzt geändert? Welche Änderungen wurden dabei durchgeführt? In welchen Commits kommt ein bestimmter Begriff in der Commit-Message vor?

Wie im vorigen Kapitel konzentrieren wir uns dabei auf den Einsatz des Kommandos `git` und gehen nur am Rande auf andere Tools ein:

- Commits durchsuchen: `git log`, `git reflog`, `git tag`, `git shortlog`
- Dateien durchsuchen: `git show`, `git diff`, `git grep`, `git blame`
- Fehler suchen: `git bisect`
- Statistik und Visualisierung: `git shortlog`, `gitstats`, Gitgraph.js

Wir wollen aber nicht unerwähnt lassen, dass Entwicklungsumgebungen, Editoren, Weboberflächen von Git-Plattformen oder spezielle (oft kommerzielle) Programme wie GitKraken beim Durchsuchen des Git-Repositorys mehr Komfort bieten. Auf unseren persönlichen Favoriten, das Programm VSCode in Kombination mit der Erweiterung GitLens, haben wir ja schon mehrfach hingewiesen.

Aber wie so oft gilt auch hier: Wenn Sie einmal verstanden haben, wie Git intern funktioniert und welche Funktionen es auf Kommandoebene gibt, fällt Ihnen die Anwendung solcher Tools umso leichter. Außerdem stößt jedes Tool früher an die Grenzen als das Kommando `git`!

4.1 Commits durchsuchen (git log)

Das Kommando `git log` zeigt, ausgehend vom aktuellen Commit, die vorangegangenen Commits an. Das ist möglich, weil zusammen mit jedem Commit auch eine Referenz auf den Parent-Commit gespeichert wird. (Bei Merge-Commits gibt es entsprechend mindestens zwei Parents.)

Standardmäßig zeigt `git log` zu jedem Commit alle Metadaten (Datum, Autor, Zweig etc.) sowie die jeweilige Commit-Message an (siehe Abbildung 4.1). Wenn es mehr

Commits gibt, als im Terminalfenster Platz finden, können Sie mit den Cursortasten durch die Commit-Abfolge scrollen. Q beendet die Anzeige.

Abbildung 4.1 Commits des Linux-Kernels in einem Terminalfenster

Spielwiese Linux-Kernel

Wenn Sie sich gerade in Git einarbeiten, haben Sie vielleicht noch kein großes eigenes Git-Repository. Verwenden Sie einfach den Linux-Kernel! Mit ca. 1,4 Millionen Commits von unzähligen Entwicklern und fast 900 mit Tags gekennzeichneten Releases (Stand Mitte 2025) finden Sie eine wunderbare Spielwiese vor – und sehen außerdem, wie schnell Git selbst bei riesigen Repositories funktioniert. Der einzige Nachteil: Mit mehr als 7,5 GByte ist der Platzbedarf auf Ihrer Festplatte/SSD erheblich.

```
git clone https://github.com/torvalds/linux.git
```

Intern wird die Ausgabe von `git log` durch einen sogenannte *Pager* geleitet, wobei üblicherweise das Kommando `less` zum Einsatz kommt. Dementsprechend gelten die bei `less` üblichen Tastenkürzel (siehe Tabelle 4.1). Besonders praktisch ist die Suchfunktion, die Sie mit / starten.

Tastenkürzel	Funktion
Cursortasten	Scrollen durch den Text.
G	Springt an den Beginn des Texts.
⇧+G	Springt an das Ende des Texts.
/ abc ↵	Sucht vorwärts.
? abc ↵	Sucht rückwärts.
N	Wiederholt die letzte Suche (vorwärts).
⇧+N	Wiederholt die letzte Suche (rückwärts).
Q	Beendet `less`.
H	Zeigt die Onlinehilfe an.

Tabelle 4.1 »less«-Tastenkürzel

Wenn `git` die Buchstaben ä, ö, ü und ß, andere internationale Zeichen oder Emojis fehlerhaft anzeigt, liegt dies oft am fehlerhaften Zusammenspiel zwischen `git` und dem Textanzeigekommando `less`. Abhilfe schafft vorübergehend die Option `--no-pager`, dauerhaft das folgende Kommando:

```
git config --global core.pager 'less --raw-control-chars'
```

Übersichtliches Logging

Häufig zeigt `git log` mehr Details an, als Sie eigentlich brauchen. Abhilfe schaffen die folgenden zwei Optionen:

- `--graph`: visualisiert Zweige (ASCII-Art)
- `--oneline`: fasst Metadaten und Commit-Message in einer Zeile zusammen

Umgekehrt fehlt im Logging vielleicht genau die Information, nach der Sie suchen:

- `--all`: zeigt auch Commits anderer Zweige an
- `--decorate`: zeigt auch Tags an
- `--name-only`: listet die veränderten Dateien auf
- `--name-status`: listet die Art der Änderungen pro Datei auf (z. B. `M` für *modified*, `D` für *deleted*, `A` für *added*)
- `--pretty=online|short|medium|full|fuller|...`: vordefinierte Ausgabeformate für die Metadaten und die Commit-Message
- `--numstat`: listet die Anzahl der geänderten Zeilen pro Datei auf
- `--stat`: listet den Umfang der Änderungen pro Datei als Balkendiagramm auf

Es ist eine gute Idee, die Wirkung der Optionen einfach einmal auszuprobieren. Die meisten Optionen können miteinander kombiniert werden. Abbildung 4.2 zeigt nochmals die Commits des Linux-Kernels, diesmal mit den Optionen `--graph --oneline`. Eine detailliertere Beschreibung der Syntax von `git log` folgt in Kapitel 12, »Kommandoreferenz«.

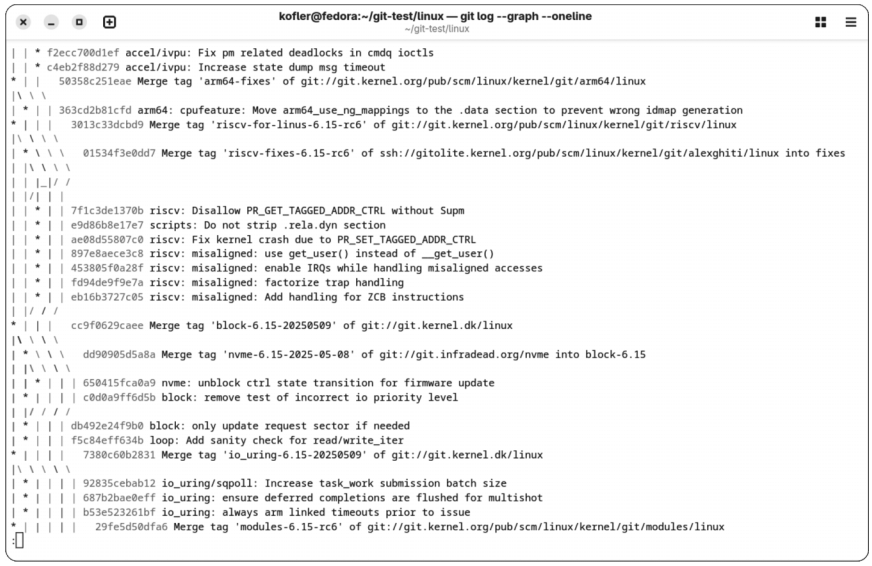


Abbildung 4.2 Kompakte Commit-Darstellung mit Zweigvisualisierung

Eigene Formatierung (Pretty-Syntax)

Wenn Sie mit den vorgegebenen Formaten nicht zufrieden sind, können Sie die Ausgabe der Commits durch die Option `--pretty=format'<fmt>'` selbst formatieren. Dabei setzt sich `<fmt>` aus `printf`-ähnlichen Codes zusammen. Unzählige weitere Codes dokumentiert `man git-log`. Das Format für die Ausgabe von Datum und Uhrzeit können Sie zusätzlich durch die Option `--date=iso|local|short|...` beeinflussen.

Im folgenden Beispiel sollen nur der siebenstellige Commit-Code, die ersten 20 Zeichen des Entwicklernamens sowie die erste Zeile der Commit-Message angezeigt werden:

```
git log --pretty=format:'%h %<(20)%an %s'
  3587Oe2  Michael Kofler      bugfix y
  ebdb53f  Bernd Öggl          added validation
  9ae3fb8  Michael Kofler      feature x
```

Um den Autorennamen rot anzuzeigen, muss die Formatzeichenkette wie folgt umgebaut werden:

```
git log --pretty=format:'%h %>(20)%Cred%an%Creset %s'
```

Die wichtigsten Codes listet Tabelle 4.2 auf.

Code	Bedeutung
`%H`	vollständiger Hashcode
`%h`	siebenstelliger Hashcode
`%ad`	Author Date
`%cd`	Commit Date
`%an`	Name des Entwicklers (*Author*)
`%ae`	E-Mail-Adresse des Entwicklers
`%s`	erste Zeile der Commit-Message (*Subject*)
`%b`	Rest der Commit-Message (*Body*)
`%n`	neue Zeile
`%<(20)`	nächste Spalte 20 Zeichen linksbündig
`%>(20)`	nächste Spalte 20 Zeichen rechtsbündig
`%Cred`	ab hier Ausdruck rot darstellen
`%Cgreen`	ab hier Ausdruck grün darstellen
`%C...`	diverse weitere Farben
`%Creset`	Farbe zurücksetzen

Tabelle 4.2 Pretty-Format-Codes

Commit-Messages durchsuchen

Mit der Option `--grep 'pattern'` zeigt `git log` nur die Commits an, in deren Message der Suchbegriff vorkommt. Dabei wird auch die Groß- und Kleinschreibung berücksichtigt. Wenn Sie das nicht wollen, geben Sie zusätzlich die Option `-i` an.

Das folgende Kommando sucht in *allen* Commits (nicht nur in denen des aktuellen Zweigs) nach dem Suchbegriff »CVE« in beliebiger Groß- und Kleinschreibung:

```
git log --all -i --grep CVE
```

Leider werden die gefundenen Suchbegriffe nicht farblich hervorgehoben. Das können Sie erreichen, indem Sie zuerst `git log` ohne die Option `--grep` ausführen, den resultierenden Ergebnistext dann mit dem Kommando `grep` filtern und schließlich durch `less` leiten. Diese Vorgehensweise ist allerdings nicht besonders effizient und bietet weniger Optionen bei der Darstellung der Commits. (Die `grep`-Option `-5` bewirkt, dass außer der gefundenen Zeile jeweils die fünf Zeilen oberhalb und unterhalb dargestellt werden. Die `less`-Option `-R` ist notwendig, damit die von `grep` weitergeleiteten Farbcodes korrekt verarbeitet werden.)

```
git log --all | grep -i -5 --color=always CVE | less -R
```

Commits suchen, die bestimmte Dateien verändern

Oft sind Sie nicht an *allen* Commits interessiert, sondern nur an Commits, in denen eine bestimmte Datei oder irgendeine Datei aus einem bestimmten Verzeichnis verändert wird. Dazu übergeben Sie an `git log` den Datei- oder Verzeichnisnamen. Falls es eine Namensgleichheit mit Tags, Branches etc. gibt, müssen Sie `--` voranstellen.

Das folgende Kommando filtert die Commits des Linux-Kernels heraus, in denen Dateien des ext4-Treibers (im Verzeichnis `fs/ext4`) verändert wurden. Dank der Option `--stat` werden auch gleich die Namen der geänderten Dateien und der Umfang der Änderungen angezeigt.

```
git log --oneline --stat -- fs/ext4
  94824ac9a8aa ext4: fix off-by-one error in do_split
   fs/ext4/namei.c | 2 +-
   1 file changed, 1 insertion(+), 1 deletion(-)
  ccad447a3d33 ext4: make block validity check resistent to sb bh
   fs/ext4/block_validity.c | 5 ++---
   fs/ext4/inode.c          | 7 ++++---
   2 files changed, 6 insertions(+), 6 deletions(-)
  7e50bbb134ab ext4: avoid -Wflex-array-member-not-at-end warning
   fs/ext4/mballoc.c | 18 ++++++++----------
   1 file changed, 8 insertions(+), 10 deletions(-)
  642335f3ea2b ext4: don't treat fhandle lookup of ea_inode as FS
   fs/ext4/inode.c | 68 +++++++++++++++++++++++++-------------
   1 file changed, 48 insertions(+), 20 deletions(-)
   ...
```

Umbenannte Dateien verfolgen

`git log -- <file>` kommt nicht mit dem Fall zurecht, dass sich der Name einer Datei ändert. In solchen Fällen müssen Sie die zusätzliche Option `--follow` verwenden, also `git log --follow -- <file>`.

Commits eines bestimmten Entwicklers suchen

Mit der Option `--author <name>` oder `--author <email>` filtern Sie die Commits eines bestimmten Entwicklers heraus. Wie bei `--grep` werden `<name>` und `<email>` als Muster interpretiert.

Mit dem folgenden Beispiel bleiben wir beim Dateisystem-Code des Linux-Kernels und suchen nach Commits von *Theodore Ts'o*. Der Apostroph im Namen macht die Suche nicht einfacher. Geben Sie stattdessen einfach einen Punkt an. (Der Punkt wird gemäß der Syntax für reguläre Ausdrücke als Platzhalter für ein beliebiges Zeichen interpretiert.)

```
git log --oneline  --author 'Theodore Ts.o'
```

Das zweite Beispiel sucht nach E-Mail-Adressen, in denen *ibm.com* vorkommt:

```
git log   --author 'ibm\.com'
```

Commit-Bereich einschränken (Range-Syntax)

Normalerweise liefert `git log [<branch>]` alle Commits des aktuellen bzw. des angegebenen Zweigs bis zurück zum Anfang der Commit-Abfolge, also in der Regel bis hin zum ersten Commit des Repositorys. Das ist nicht immer sinnvoll. Oft sind Sie nur an Commits interessiert, die spezifisch für einen Branch oder mehrere Branches gelten, nicht aber an der gemeinsamen Basis. In solchen Fällen können Sie die Range-Syntax `<branch1>..<branch2>` oder `<branch1>...<branch2>` verwenden. Anstelle von Zweignamen können Sie auch Hashcodes oder andere Revisionsangaben verwenden (siehe auch Abschnitt 3.12, »Referenzen auf Commits«).

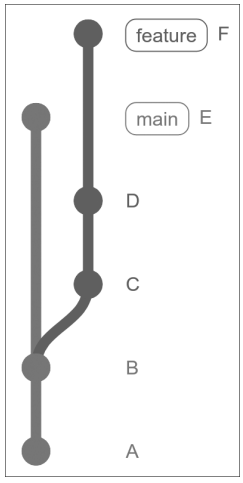


Abbildung 4.3 Commits in zwei Zweigen

Als Ausgangspunkt für die folgenden Beispiele gilt die in Abbildung 4.3 dargestellte Commit-Abfolge, wobei die Commit-Messages einfach `A`, `B`, `C` usw. lauten. Momentan ist der Zweig `main` aktiv. Ohne Range-Syntax werden jeweils alle Commits bis zurück zum initialen Commit `A` angezeigt:

```
git checkout main
git log --oneline

  ebdb53f (HEAD -> main) E
  c9bb505 B
  45c6cd4 A

git log --oneline feature

  35870e2 (feature) F
  9ae3fb8 D
  b115d39 C
  c9bb505 B
  45c6cd4 A
```

`git log main..feature` zeigt nur die nicht mit `main` zusammengeführten Commits des Feature-Zweigs. Die gemeinsame Basis fällt weg (hier also die Commits `A` und `B`). Anstelle von `main..feature` gibt es zwei alternative Schreibweisen, die eigentlich syntaktisch klarer sind, in der Praxis aber selten vorkommen:

```
git log --oneline main..feature
git log --oneline feature --not main  (gleichwertig)
git log --oneline feature ^main       (auch gleichwertig)

  35870e2 (feature) F
  9ae3fb8 D
  b115d39 C
```

`git log main...feature` mit drei Punkten funktioniert wie das obige Kommando, berücksichtigt aber zusätzlich die seit der Trennung der Zweige in `main` durchgeführten Commits. (Zum selben Ergebnis kommen Sie übrigens auch, wenn Sie die Branch-Namen vertauschen.)

```
git log --oneline main...feature
git log --oneline feature...main      (gleichwertig)

  35870e2 (feature) F
  ebdb53f (HEAD -> main) E
  9ae3fb8 D
  b115d39 C
```

Commits zeitlich eingrenzen

Anstelle der im vorigen Abschnitt vorgestellten Range-Syntax, die den Commit-Bereich anhand logischer Kriterien einschränkt, können Sie die von `git log` gelieferten Commits mit Optionen auch zeitlich eingrenzen:

- `--since <date>` oder `--after <date>` zeigt nur Commits, die nach `<date>` entstanden sind.
- `--until <date>` oder `--before <date>` zeigt nur Commits, die vor/bis `<date>` durchgeführt wurden.

Wenn Sie die im Januar 2025 entstandenen Commits ansehen möchten, führen Sie das folgende Kommando aus:

```
git log  --after 2025-01-01 --until 2025-01-31
```

Commits sortieren

Standardmäßig werden Commits durch `git log` zeitlich sortiert, der neueste Commit zuerst. Das ändert sich allerdings, sobald Sie die Option `--graph` hinzufügen. `git log` bündelt jetzt zusammengehörende Commits. Wenn Sie die Commits trotz `--graph` im zeitlichen Ablauf ordnen wollen, verwenden Sie die Zusatzoption `--date-order`. Umgekehrt können Sie die Gruppierung der Commits nach Zweigen auch ohne `--graph` mit `--topo-order` erreichen.

Die folgenden Beispiele beziehen sich wieder auf Abbildung 4.3. Allerdings wurden die Zweige mit Merge verbunden:

```
git checkout main
git merge feature
```

Normalerweise ordnet `git log` die Commits streng chronologisch. (Die Option `--pretty` ermöglicht hier eine einzeilige Darstellung samt Commit-Datum. Zur Verbesserung der Übersicht haben wir die Originalausgaben ein wenig umformatiert und Wochentage und Jahreszahlen entfernt.)

```
git log --pretty=format:"%h %cd %s" --date=local

  52003e9  Jan 13 07:06:25     Merge branch 'feature'
  35870e2  Jan 10 10:32:56     F
  ebdb53f  Jan 10 10:32:38     E
  9ae3fb8  Jan 10 10:32:04     D
  b115d39  Jan 10 10:30:36     C
  c9bb505  Jan 10 10:29:24     B
  45c6cd4  Jan 10 10:29:16     A
```

Mit der Option `--graph` werden die Commits `C`, `D` und `F` gruppiert:

```
git log --pretty=format:"%h %cd %s" --date=local --graph

  *    52003e9  Jan 13 07:06:25    Merge branch 'feature'
  |\
  | *  35870e2  Jan 10 10:32:56    F
  | *  9ae3fb8  Jan 10 10:32:04    D
  | *  b115d39  Jan 10 10:30:36    C
  * |  ebdb53f  Jan 10 10:32:38    E
  |/
  *    c9bb505  Jan 10 10:29:24    B
  *    45c6cd4  Jan 10 10:29:16    A
```

Die Option `--date-order` stellt trotz Zweigdarstellung die ursprüngliche Ordnung wieder her:

```
git log --pretty=format:"%h %cd %s" --date=local --graph \
        --date-order

  *    52003e9  Jan 13 07:06:25    Merge branch 'feature'
  |\
  | *  35870e2  Jan 10 10:32:56    F
  * |  ebdb53f  Jan 10 10:32:38    E
  | *  9ae3fb8  Jan 10 10:32:04    D
  | *  b115d39  Jan 10 10:30:36    C
  |/
  *    c9bb505  Jan 10 10:29:24    B
  *    45c6cd4  Jan 10 10:29:16    A
```

Author Date versus Commit Date

Zusammen mit jedem Commit werden *zwei* Zeitangaben gespeichert, das *Author Date* und das *Commit Date*. Normalerweise stimmen beide Zeitangaben überein. Bei Commits, die durch Rebasing verändert wurden, ist das aber nicht der Fall: Dann gibt das Author Date den Zeitpunkt an, zu dem der ursprüngliche Commit entstanden ist. Das Commit Date verweist auf den Zeitpunkt des Rebasings.

Wenn Sie beim Sortieren der Commits das Author Date berücksichtigen wollen, verwenden Sie die Option `--author-date-order`. Die Commits werden nun wie bei `--topo-order` gruppiert, innerhalb der Zweige (von denen es dank Rebasing üblicherweise weniger oder gar keine gibt) wird aber das Author Date als Sortierkriterium verwenden.

Markierte Commits (git tag)

`git tag` liefert eine Liste aller Tags. `git tag <pattern>` schränkt das Ergebnis auf Tags ein, die dem Suchmuster entsprechen. Sobald Sie das gewünschte Tag ermittelt haben, können Sie sich mit `git log <tagname>` die Commits ansehen, die zu diesem Release geführt haben.

Alternativ können Sie mit `git log --simplify-by-decoration` nur solche Commits anzeigen, die Tags enthalten oder auf die ein Zweig verweist. In großen Repositories ist das aber vergleichsweise langsam.

`git log` zeigt normalerweise keine Tags an. Wenn Sie diese Zusatzinformation wünschen, übergeben Sie an `git log` die Option `--decorate`. Wenn Sie dennoch eine kompakte Anzeige wünschen, können Sie `--decorate` wie bisher mit `--oneline` kombinieren.

Referenz-Log (git reflog)

Immer wenn von der Commit-Abfolge (also dem *Commit-Log*) die Rede ist, müssen wir auch auf das Referenz-Log hinweisen: Es enthält alle lokal durchgeführten Kommandos, die den globalen HEAD oder den Head eines Zweiges verändert haben. Das Kommando `git reflog` listet diese Aktionen samt den Hashcodes der Commits auf:

```
git reflog

  ebdb53f (HEAD -> main) HEAD@{0}: checkout: moving from
                                   feature to main
  35870e2 (feature) HEAD@{1}: commit: F
  9ae3fb8 HEAD@{2}: checkout: moving from main to feature
  ebdb53f (HEAD -> main) HEAD@{3}: commit: E
  c9bb505 HEAD@{4}: checkout: moving from feature to main
  9ae3fb8 HEAD@{5}: commit: D
```

Wenn Sie die detaillierte Ausgabe von `git log` wünschen, aber gleichzeitig genau die Commits sehen möchten, die `git reflog` liefert, führen Sie `git log` mit der Option `--walk-reflog` aus:

```
git log --walk-reflogs

  commit ebdb53f0db624c6dd4d754940903c3be905a9be (HEAD -> main)
  Reflog: HEAD@{0} (Michael Kofler <...>)
  Reflog message: checkout: moving from feature to main
  Author: Michael Kofler <...>
  Date:   Jan 10 10:32:38 2025 +0200

      E
```

```
commit 35870e24fb49bb77622e17f5844cfaeb515c0a00 (feature)
Reflog: HEAD@{1} (Michael Kofler <...>)
Reflog message: commit: F
Author: Michael Kofler <...>
Date:   Jan 10 10:32:56 2025 +0200

    F
```

Anstelle von `--walk-reflog` können Sie auch die Option `--reflog` verwenden. Damit wird jeder Commit nur einmal angezeigt. (Bei `--walk-reflog` kann der gleiche Commit mehrfach auftauchen, z. B. immer dann, wenn Sie zuvor mit `git checkout` den Zweig gewechselt haben.)

4.2 Dateien durchsuchen

Während wir uns Abschnitt 4.1, »Commits durchsuchen (git log)«, darauf konzentriert haben, die Metadaten eines Repositorys zu durchsuchen, ist nun der Inhalt an der Reihe: Welchen Inhalt hatte eine bestimmte Datei zu einem früheren Zeitpunkt? Was hat sich seither geändert? Und wer ist dafür verantwortlich? Bei der Beantwortung dieser und weiterer Fragen hilft ein ganzes Bündel von Kommandos, unter anderem `git show`, `git diff` und `git blame`.

Alte Versionen einer Datei ansehen (git show)

Das Kommando `git show <revision>:<file>` haben wir in Abschnitt 3.4, »Commit-Undo«, schon vorgestellt: Es gibt die Datei `<file>` in dem Zustand aus, den sie hatte, als der Commit `<revision>` aktuell war. Wenn Sie also Version 2.0 Ihres Programms mit dem Tag `v2.0` gekennzeichnet haben und wissen wollen, wie die Datei `index.php` damals aussah, führen Sie das folgende Kommando aus:

```
git show v2.0:index.php
```

Natürlich können Sie die Ausgabe auch in eine andere Datei umleiten, damit Sie beide Versionen (die aktuelle und die alte) parallel zur Verfügung haben:

```
git show v2.0:index.php > old_index.php
```

Unterschiede zwischen Dateien ansehen (git diff)

Wenn Sie wissen möchten, was sich zwischen der aktuellen Version und einer alten Version einer Datei geändert hat, verwenden Sie `git diff`. Das folgende Programm zeigt an, wie sich die Datei `index.php` seit der Version 2.0 geändert hat. Die Ausgabe besteht aus mehreren Blöcken, die mit `@@` eingeleitet werden und die Position

angeben. Zur Orientierung helfen einige Zeilen Code, den Kontext herzustellen. Anschließend folgen die geänderten Zeilen, denen - oder + vorangestellt ist, je nachdem, ob sie gelöscht oder hinzugefügt wurden. (Im Terminal sind die gelöschten Zeilen rot und die hinzugefügten Zeilen grün hervorgehoben, was in diesem Buch leider nicht dargestellt werden kann.)

```
git diff v2.0 index.php
```

```
  diff --git a/index.php b/index.php
  index a41783c..d1e3af2 100644
  --- a/index.php
  +++ b/index.php
  @@ -10,9 +10,9 @@ try {
     exit();
   }
  -try {
  -  $ctl->checkAccess();
  -} catch (Exception $e) {
  +if ($ctl->checkAccess() === TRUE) {
  +  $ctl->showRequestedPage();
  +} else {
     if ($ctl->isJSONRequest()) {
       $data = new stdClass();
       $data->error = true;
  @@ -29,4 +29,3 @@ try {
       exit();
     }
   }
  -$ctl->showRequestedPage();
```

Wenn Sie nur am Umfang der Änderungen interessiert sind, übergeben Sie zusätzlich die Option `--compact-summary`:

```
git diff --compact-summary v2.0 index.php
 index.php | 7 +++----
 1 file changed, 3 insertions(+), 4 deletions(-)
```

Der Befehl `git diff <revision1>..<revision2> <file>` zeigt die Änderungen zwischen zwei alten Versionen an:

```
git diff --compact-summary v1.0..v2.0 index.php
```

Die Option `--word-diff` hilft dabei, Änderungen in einem mehrzeiligen Fließtext zu verfolgen. Die Granularität von `git diff` ändert sich damit von Zeilen (per Default) zu Wörtern.

```
git diff --word-diff readme.txt
```

Natürlich können Sie an `git diff` anstelle von Tags oder Versionen auch die Hashcodes von Commits, die Namen von Zweigen oder sonstige Referenzen übergeben (siehe Abschnitt 3.12, »Referenzen auf Commits«). Beachten Sie, dass die ausgesprochen praktische Schreibweise `HEAD@{2.weeks.ago}` zur zeitlichen Einordnung nur für lokal durchgeführte Commits funktioniert, also nur für Aktionen, die im Reflog gespeichert sind. Davon abgesehen gibt es keine Möglichkeiten, den Vergleichs-Commit zeitlich festzulegen. Gegebenenfalls müssen Sie zuerst mit `git log` einen zeitlich passenden Commit suchen und dessen Hashcode dann an `git diff` übergeben.

Range-Syntax mit drei Punkten

Die Variante `git diff <rev1>...<rev2>` ist vor allem dann zweckmäßig, wenn es sich bei den Revisionen um Zweige handelt. In diesem Fall ermittelt `git diff` zuerst die letzte gemeinsame Basis beider Zweige und zeigt dann an, was sich in `<rev2>` im Vergleich zum letzten gemeinsamen Commit verändert hat. Anders als bei `<rev1>..<rev2>` werden aber alle Änderungen ignoriert, die seither in `<rev1>` passiert sind.

git-diff-Resultate schöner anzeigen

Mit dem Zusatzprogramm `delta` können Sie die Ausgaben von `git diff` übersichtlicher anzeigen. Wir stellen Ihnen `delta` in Abschnitt 9.5, »git diff mit delta lesbarer darstellen«, näher vor.

Unterschiede zwischen Commits ansehen

Wenn Sie bei `git diff` auf die Angabe einer Datei verzichten, zeigt es *alle* geänderten Dateien seit der angegebenen Version bzw. zwischen zwei Versionen/Commits an. Wiederum ist die Option `--compact-summary` hilfreich, wenn Sie vorerst nur einen Überblick gewinnen möchten.

Bei umfangreichen Änderungen fehlt der Platz, um für jede geänderte Zeile ein + oder ein - auszugeben. Stattdessen wird nach | die Gesamtanzahl der geänderten Zeilen angegeben. Die Anzahl der Plus- und Minuszeichen ist relativ zu der Datei mit den größten Änderungen. Je länger der Balken aus den Zeichen ist, desto umfangreicher sind die Änderungen ausgefallen.

```
git diff --compact-summary v1.0..v2.0 index.php

  css/autocompleteList.css                  |   225 +-
  css/editproject.css (new)                 |    13 +
```

```
 css/iprot.css                                  |   648 ++++-
 css/iprot/jquery-ui-1.8.13.custom.css          |     2 +-
 css/mobile.css (new)                           |    17 +
 ...
 269 files changed, 22819 insertions(+), 12792 deletions(-)
```

Selten sind Sie einfach an allen Änderungen interessiert. Zwei Optionen helfen dabei, das Ergebnis gezielt einzuschränken:

- Mit `-G <pattern>` geben Sie ein Suchmuster (einen regulären Ausdruck) an. `git diff` liefert dann nur die Textdateien, deren Änderungen den Suchausdruck enthalten, wobei die Groß- und Kleinschreibung exakt übereinstimmen muss.
- `--diff-filter=A|C|D|M|R` filtert jene Dateien heraus, die hinzugefügt (*added*), kopiert (*copied*), gelöscht (*deleted*), verändert (*modified*) oder umbenannt (*renamed*) wurden.

Das folgende Kommando liefert die Dateien, die zwischen Version 1.0 und 2.0 verändert wurden und in deren Code der Suchtext *PDF* vorkommt.

```
git diff  -G PDF --diff-filter=M --compact-summary v1.0..v2.0
```

Änderungen seit dem letzten Commit

Bevor Sie `git commit` ausführen, ist es oft eine gute Idee, sich einen Überblick über die Änderungen in allen für den Commit vorgemerkten Dateien zu verschaffen. Genau das macht `git diff --staged`.

Sollten Sie `git add` noch nicht ausgeführt haben oder vorhaben, `git commit -a` zu verwenden, zeigt `git diff` ohne irgendwelche weiteren Parameter alle zuletzt durchgeführten Änderungen an. (Nicht berücksichtigt werden neue Dateien, die noch nicht unter Versionskontrolle stehen.)

Dateien durchsuchen (git grep)

An welchen Stellen in den zahlreichen Dateien aus Ihrem riesigen Projekt wird die Funktion *X* aufgerufen oder ein Objekt der Klasse *Y* erzeugt? Antwort auf derartige Fragen gibt `git grep <pattern>`. Standardmäßig berücksichtigt das Kommando alle Dateien im Projektverzeichnis und listet die Zeilen auf, in denen der Suchausdruck in exakter Groß- und Kleinschreibung auftritt. (Wenn Sie nicht zwischen Groß- und Kleinschreibung differenzieren wollen, geben Sie zusätzlich die Option `-i` an.)

```
git grep SKAction
  ios-pacman/Maze.swift:   let setGlitter = SKAction.setTextur...
  ios-pacman/Maze.swift:   let setStandard  = SKAction.setText...
  ios-pacman/Maze.swift:   let waitShort = SKAction.wait(forDu...
  ...
```

Ein kompakteres Suchergebnis erhalten Sie mit `--count`. In diesem Fall zeigt `git grep` nur an, wie oft der Suchausdruck in den jeweiligen Dateien vorkommt:

```
git grep --count CGSize
  ios-pacman/CGOperators.swift:6
  ios-pacman/Global.swift:1
  ios-pacman/Maze.swift:4
  ...
```

Durch die Angabe von Dateien oder Verzeichnissen können Sie die Suche einschränken. Das folgende Kommando durchsucht die Dateien im Verzeichnis `css` nach dem Schlüsselwort `margin`. Wegen der Option `-n` wird zu jeder Fundstelle auch die Zeilennummer angegeben.

```
git grep -n margin css/
  css/config.json:100:    "@form-group-margin-bottom": "15px",
  css/config.json:144:    "@navbar-margin-bottom": "@line-heig...
  css/editglobal.css:25:  margin-top: 1px;
  css/editglobal.css:29:  margin-top: 0px;
  ...
```

Natürlich können Sie auch alte Versionen Ihres Codes durchsuchen, indem Sie die gewünschte Revision vor den Dateinamen oder Verzeichnissen angeben. Wenn der Suchausdruck wie im folgenden Beispiel Sonder- oder Leerzeichen enthält, müssen Sie ihn zwischen Apostrophe stellen. Das folgende Beispiel sucht in Version 2.0 des Programms nach `UPDATE`-Kommandos, die die Tabelle `person` verändern:

```
git grep 'UPDATE person' v2.0
  v2.0:lib/delete.php:         $sql = "UPDATE person SET sta...
  v2.0:lib/person.php:         $sql = sprintf("UPDATE person...
  v2.0:lib/personengruppe.php: $sql = sprintf("UPDATE person...
  ...
```

Schwierig ist die Anwendung von `git grep`, wenn Sie nicht wissen, in welchem Commit Sie suchen sollen, oder wenn es sich um Änderungen handelt, die nur vorübergehend durchgeführt und später wieder aus der Codebasis entfernt wurden. In solchen Fällen können Sie mit `git rev-list v1.0..v2.0` eine Liste mit den Hashcodes aller Commits für den fraglichen Zeitraum erstellen. Diese Liste verarbeiten Sie dann mit `git grep`.

Beispielsweise zählt das folgende Kommando, wie oft das SQL-Schlüsselwort `UPDATE` in diversen Versionen der Datei `lib/kapitel.php` vorkommt. Wie bei `git log` wird der neueste Commit zuerst berücksichtigt. Die Zeichen `--` trennen die durch `git rev-list` erzeugte Hashcode-Liste vom Dateinamen.

```
git grep -c 'UPDATE'  $(git rev-list v1.0..v2.0) -- user.php
  262d67fed686cda939092e7b0cb337bbc1e2dbe9:user.php:5
  96d0a06d389784ec93f252a097185ee3678a2c1c:user.php:5
  c07c2f0ce5682bea898ba3a65a15bf5230dd23dc:user.php:4
  ...
```

Urheberschaft von Code herausfinden (git blame)

Wenn Sie mit den hier beschriebenen Kommandos die Datei gefunden haben, die Sie eigentlich interessiert, ist die nächste Frage natürlich: Wer ist für den dort enthaltenen Code verantwortlich? Ein großartiges Hilfsmittel ist in diesem Fall `git blame <file>`. Ohne weitere Optionen zeigt es die betreffende Datei zeilenweise an und gibt bei jeder Zeile an, in welchem Commit von welchem Autor zu welchem Datum diese Zeile verändert wurde (siehe Abbildung 4.4).

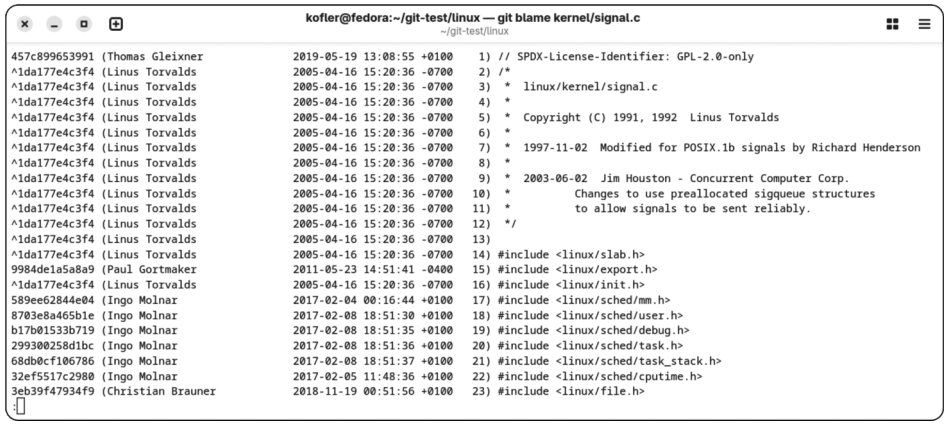


Abbildung 4.4 Urheberschaft der Datei »signal.c« des Linux-Kernels

Mit der Option `-L 100,200` berücksichtigen Sie nur die Zeilennummern 100 bis 200. Eine große Hilfe beim Lesen der Ausgaben sind die beiden folgenden Optionen:

- `--color-lines` stellt Fortsetzungszeilen aus dem gleichen Commit in blauer Farbe dar.
- `--color-by-age` kennzeichnet frisch geänderten Code rot (Änderungen im letzten Monat) und mäßig neuen Code weiß (Änderungen im letzten Jahr).

Eine noch übersichtlichere Darstellung der Blame-Ergebnisse bieten die Websites von GitLab, GitHub und Co.

Außerdem können Sie dort sich per Mausklick direkt den betreffenden Commit ansehen.

Boundary Commits

Wenn im lokalen Repository nicht alle Commits enthalten sind, kommt es vor, dass einzelnen Hashcodes das Zeichen ^ (*Caret* oder *Circumflex*) vorangestellt wird, z. B. `^1da177e4c3f4`. Es weist auf einen *Boundary Commit* hin, also auf den letzten im Repository verfügbaren Commit.

4.3 Fehler suchen (git bisect)

Stellen Sie sich vor, Sie bemerken, dass in einem Feature Ihres Programms ein Fehler auftritt, aber es gelingt Ihnen nicht, dessen Ursache zu finden oder auch nur einzugrenzen. Vermutlich handelt es sich um eine Wechselwirkung, die erst durch Änderungen in mehreren Dateien entstanden ist.

Sie sind sich sicher, dass der Fehler früher nicht aufgetreten ist. Mit `git checkout v1.5` sind Sie vorübergehend zur Version 1.5 zurückgekehrt und haben sie nochmals getestet. Dort ist die Welt noch in Ordnung. Seither gab es 357 Commits. (Das Kommando `git rev-list` ist eine einfachere Variante zu `git log`, das normalerweise anstelle von Commit-Messages einfach nur die Hashcodes der betreffenden Commits liefert. Mit der Option `--count` zählt es die Commits zwischen zwei Punkten eines Zweigs.)

```
git rev-list v1.5..HEAD --count
  357
```

Um herauszufinden, was den Fehler verursacht, müssen Sie den ersten Commit finden, in dem der Fehler auftritt. Das klingt nach der sprichwörtlichen Suche nach einer Nadel im Heuhaufen.

Glücklicherweise unterstützt Sie `git bisect` bei dem Unterfangen. Die Idee von `git bisect` besteht darin, dass Sie zuerst den letzten bekannten »guten« und »schlechten« Commit angeben – in diesem Beispiel den Commit mit dem Tag `v1.5` sowie den aktuellen Commit (also HEAD). `git bisect` führt nun einen Checkout in der Mitte des Commit-Bereichs aus, halbiert also den Suchbereich. (Damit liegt der Fall eines Detached HEADs vor, d. h., HEAD verweist nicht auf das Ende eines Zweigs, sondern auf irgendeinen Commit in der Vergangenheit.)

```
git bisect start
git bisect bad HEAD
git bisect good v1.5
  Bisecting: 178 revisions left to test after this
    (roughly 8 steps)
  [e84fd83319c1280bcef38400299fd55925ea25e6] Merge branch ...
```

Jetzt liegt es an Ihnen, zu testen, ob der Fehler bei diesem Commit noch immer auftritt. *Wie* Sie diesen Test durchführen, hängt ganz von der Art des Codes ab. Eventuell müssen Sie Ihr Programm kompilieren, um es zu testen. Bei einer Webapplikation reicht dagegen ein Test im Browser. Je nachdem, wie der Test ausfällt, melden Sie das Ergebnis mit `git bisect bad` oder mit `git bisect good`:

```
git bisect bad
  Bisecting: 89 revisions left to test after this
    (roughly 7 steps)
  [cea22541893ded6e6e9f6a9d40bf6d0c2ec806d8] bugfix xy ...
```

Abhängig von Ihrer Antwort weiß `git bisect` nun, ob es in der oberen oder unteren Hälfte des Commit-Bereichs weitersuchen soll. Das Kommando führt einen weiteren Checkout in der Mitte des verbleibenden Suchbereichs aus. Der Suchbereich wurde damit auf ca. ein Viertel reduziert.

Abermals müssen Sie nun den Test wiederholen, ob der Fehler noch auftritt oder nicht, und diese Information an `git` weiterleiten. Auf diese Weise fahren Sie fort, bis `git bisect` schließlich meldet:

```
git bisect good
  4127d9d06ecbae0d4d9babaaa8aacebc0c8853cb is the first bad
  commit ...
```

Damit wissen Sie, zu welchem Zeitpunkt in der Vergangenheit der Fehler erstmals aufgetreten ist. Die Suche nach der Ursache des Fehlers steht jetzt noch aus – aber eigentlich sollte `git diff HEAD^`, also die Zusammenfassung der Änderungen im Vergleich zu vorigen Commit, Sie auf die richtige Spur bringen.

Mit `git bisect reset` beenden Sie schließlich `git bisect` und kehren zurück zum Head des Zweiges, in dem Sie sich zu Beginn der Suche befanden. Dort versuchen Sie nun, den jetzt eingegrenzten Fehler endgültig zu beheben.

```
git bisect reset
  Previous HEAD position was ef81d5c fix: getLink for csv ...
  Switched to branch 'develop'
```

4.4 Statistik und Visualisierung

Bei großen Repositories sieht man oft den Wald vor lauter Bäumen (in unserem Fall eigentlich: vor lauter Zweigen) nicht mehr. In diesem Abschnitt stellen wir Ihnen `git`-Kommandos sowie diverse Werkzeuge vor, mit denen Sie wieder den Durchblick erlangen.

Einfache Zahlenspiele (git shortlog)

Ein praktisches Kommando, um einen ersten Überblick zu erhalten, ist `git shortlog`. In seiner einfachsten Form liefert es eine alphabetisch geordnete Liste aller Commit-Autoren, wobei zu jedem Autor die Anzahl der Commits sowie jeweils die erste Zeile jeder Commit-Message angegeben werden.

Durch diverse Optionen können Sie die Ausgabe weiter verkürzen. Das folgende Kommando liefert eine Liste der Entwickler und Entwicklerinnen des Linux-Kernels, die seit Anfang 2021 die meisten Commits aufzuweisen haben, wobei Merge-Commits nicht gerechnet werden:

```
git shortlog --summary --numbered --email --no-merges \
             --since 2021-01-01

  3996  Krzysztof Kozlowski <...@linaro.org>
  3976  Uwe Kleine-König <...@pengutronix.de>
  3233  Christoph Hellwig <...@lst.de>
  2859  Andy Shevchenko <...@linux.intel.com>
  2609  Kent Overstreet <...@linux.dev>
  2392  Sean Christopherson <...@google.com>
  2344  Dmitry Baryshkov <...@kernel.org>
  2179  Ville Syrjälä <...@linux.intel.com>
   ...
```

Die Gesamtanzahl aller Commits (über alle Zweige) ermitteln Sie mit `git rev-list`:

```
git rev-list --all --count
  1352809
```

Die Anzahl der Dateien im aktuellen Zweig ermitteln Sie, indem Sie die Ausgabe von `git ls-files` an `wc` (*word count*) weiterleiten:

```
git ls-files | wc -l
  88835
```

Analog können Sie auch die Anzahl der Branches und Tags herausfinden:

```
git branch -a | wc -l
  3

git tag | wc -l
  883
```

Den Umfang der Änderungen zwischen zwei Versionen/Zweigen/Revisionen Ihres Projekts ermitteln Sie mit `git diff --shortstat`:

```
git diff --shortstat v6.13..v6.14
   9960 files changed
 443249 insertions(+)
 147714 deletions(-)
```

Statistiktools und -Scripts

Das Internet ist voll von Scripts und Programmen, die aus einem Git-Repository mehr Details als die obigen Kommandos herausholen können. Einen ersten Startpunkt bietet der folgende Stack-Overflow-Artikel:

https://stackoverflow.com/questions/1828874

Beliebt und einfach anzuwenden ist das Go-Programm `gitstats`, das die Weiterentwicklung eines veralteten, gleichnamigen Python-Scripts ist. Sie können Binarys für Linux, Windows und macOS von der folgenden Seite herunterladen:

https://github.com/nielskrijger/gitstat

Im nächsten Schritt wenden Sie das Programm auf eines Ihrer lokalen Repositorys an:

```
path/to/gitstat path/to/repo
```

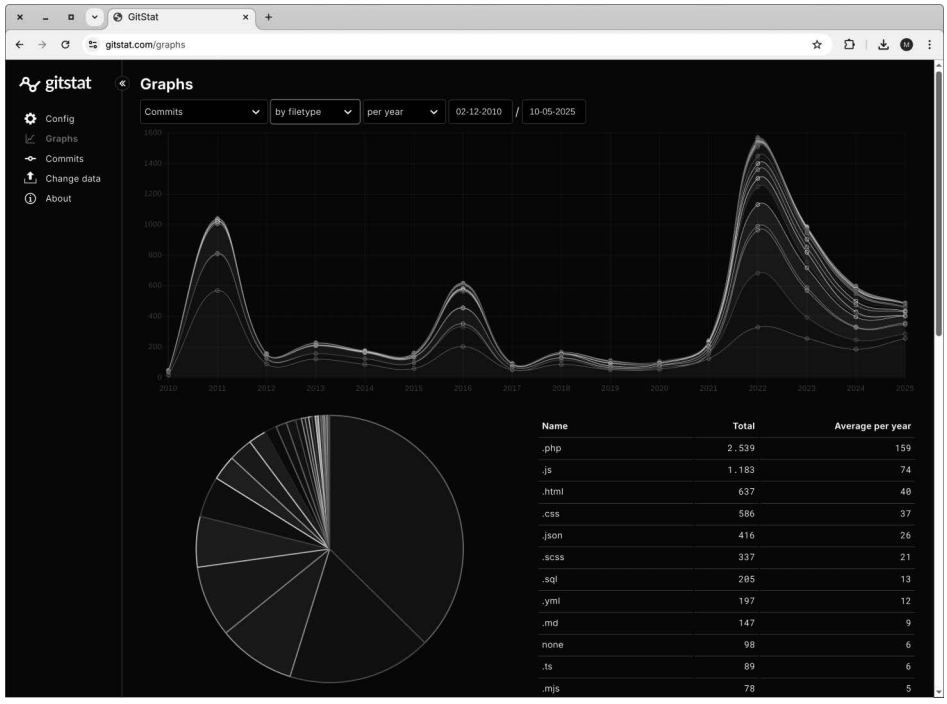


Abbildung 4.5 Visualisierung eines Repositorys mit »gitstat«

Jetzt ist etwas Geduld erforderlich. Bei unseren Tests dauerte die Analyse eines ca. 600 MByte großen Repositorys gut eine halbe Stunde. Das Ergebnis ist schließlich die Datei `<repo>_result.yaml`. Sie wird im gerade aktuellen Verzeichnis gespeichert.

Diese Datei können Sie nun auf der Seite *https://gitstat.com* visualisieren. Sie müssen keine Datenschutzbedenken haben. Die Datei wird nicht auf einen externen Server hochgeladen, vielmehr kümmert sich lokaler JScript-Code um die grafische Darstellung. Unter anderem können Sie sich ansehen, welche Personen über den zeitlichen Verlauf des Projekts wie viele Commits durchgeführt und wie viele Zeile Code geändert haben. Alternativ können Sie das Projekt auch nach den Dateitypen aufschlüsseln (siehe Abbildung 4.5).

Zweige visualisieren

Gerade in Git-Schulungen oder bei dem Versuch, Kollegen die Funktionsweise von Git zu verdeutlichen, besteht der Wunsch, die über mehrere Zweige verteilten Commits »ordentlich« zu visualisieren. Die Ergebnisse von `git log --graph` sind dazu aber ungeeignet.

Schon ein wenig besser ist die Darstellung durch das auf vielen Rechnern installierte Programm `gitk` (siehe Abbildung 4.6). Es wird üblicherweise aus dem Terminal heraus gestartet und zeigt die Commit-Abfolge für den gerade aktuellen Zweig.

Abbildung 4.6 Visualisierung von Zweigen durch »gitk«

Falls Sie Wert auf eine übersichtlichere Darstellung von Zweigen legen, haben wir ein paar Vorschläge für Sie:

- Das kommerzielle Programm *GitKraken* zeigt nicht nur die Commit-Abfolge in einer ansprechenden Form an, sondern bietet eine Menge weiterer Funktionen, die bei der Administration von Git-Repositories helfen. Die kostenlose Version kann nur für öffentliche Repositories verwendet werden.
- Auch manche Git-Plattformen enthalten Visualisierungsfunktionen. Beispielsweise zeigt GitLab auf der Teilseite Repository • Graph eine übersichtliche Darstellung des Commit-Verlaufs (siehe Abbildung 4.7).
- GitHub-Anwender, die diesbezüglich weniger verwöhnt sind, sollten sich das kommerzielle Projekt *GFC* (*Git Flow Chart*) ansehen: Die Website *https://gfc.io* kann die Commit-Abfolge von Repositories auf GitHub und Bitbucket visualisieren. Die Grundfunktionen stehen für öffentliche Repositories kostenlos zur Verfügung. Wenn Sie GFC für private Repositories oder in Kombination mit den GitHub-Teamfunktionen einsetzen wollen, müssen Sie einen monatlichen Obolus leisten.

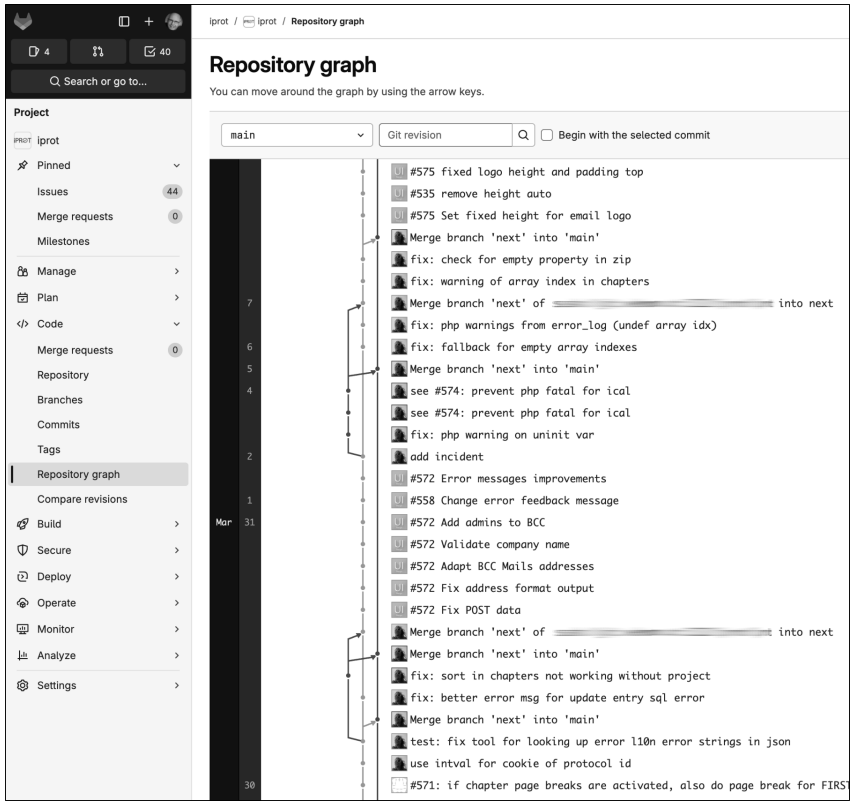


Abbildung 4.7 Darstellung von Zweigen in GitLab

GitGraph.js

Vielleicht ist es Ihnen aufgefallen: Alle Abbildungen in diesem Buch, die die Commit-Abfolge mehrerer Zweige zeigen, haben ein einheitliches Aussehen. Das ist natürlich kein Zufall. Mit der Open-Source-Bibliothek *GitGraph.js* und wenigen Zeilen eigenem JavaScript-Code lassen sich viele Visualisierungswünsche erfüllen. Das Ergebnis ist dann im Webbrowser zu bewundern. Werfen Sie einen Blick auf die Projektwebsite *https://www.nicoespeon.com/gitgraph.js*, die elementare Arbeitstechniken in Form einer Präsentation zusammenfasst!

Leider ist GitGraph.js nicht in der Lage, echte Commits aus einem Repository zu zeichnen. Sie müssen die Commit-Struktur also durch entsprechende JavaScript-Anweisungen zusammensetzen, was mit einigem Aufwand verbunden ist (den wir für dieses Buch natürlich nicht gescheut haben).

Die folgenden Zeilen zeigen den Code für Abbildung 4.3. `graphContainer` verweist auf die Stelle im HTML-Code, wo das Diagramm dargestellt werden soll. `mytemplate` enthält einige Optionen, Commits ohne Autoren und Hashcodes, Zweige aber mit ihren Namen anzuzeigen. `createGitGraph` erzeugt die vorerst leere Commit-Abfolge. Mit `branch` und `commit` werden dann Commits und Zweige hinzugefügt.

```
<!doctype html>
<html><head>
<script src="https://cdn.jsdelivr.net/npm/@gitgraph/js">
</script>
</head>
<body>
<div id="graph"></div>
<script>
const graphContainer = document.getElementById("graph");
const mytemplate =  GitgraphJS.templateExtend(
   GitgraphJS.TemplateName.Metro, {
        commit: {  message: { displayAuthor: false,
                              displayHash: false    } },
        branch: {  label:   { display: true } }
      });
const gitgraph = GitgraphJS.createGitgraph(
  graphContainer,
  { author: " ", template: mytemplate } );
// hier beginnt der Code für die Commits/Zweige
const main  = gitgraph.branch("main).commit("A").commit("B")
const develop = main.branch("feature").commit("C").commit("D")
main.commit("E")
develop.commit("F")
</script>
</body></html>
```

Mermaid

Eine interessante Alternative zu GitGraph.js ist das Online-Zeichenprogramm Mermaid. Es erstellt aus YAML-ähnlichen Texten alle erdenklichen Arten von Diagrammen, unter anderem auch Commit-Verläufe. Dabei gibt es eine Menge Gestaltungsoptionen. Mermaid wird direkt im Webbrowser bedient (siehe Abbildung 4.8).

https://mermaid.js.org/syntax/gitgraph.html

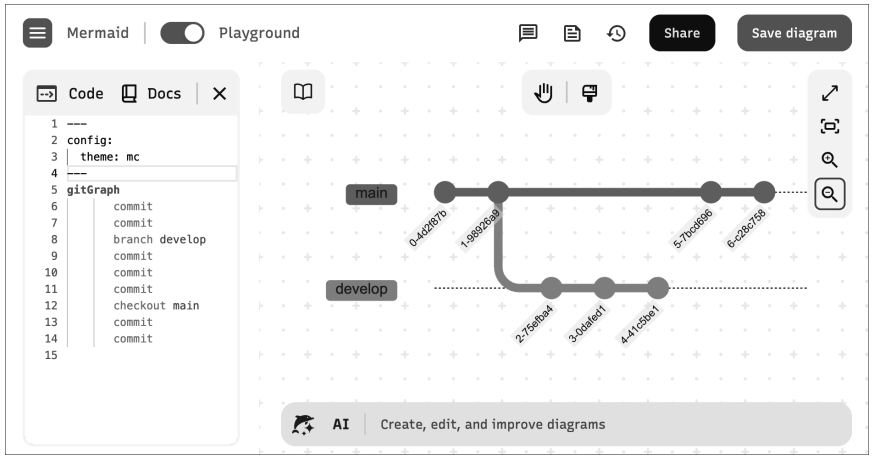


Abbildung 4.8 Mit »Mermaid« können Sie im Webbrowser Commit-Abläufe und Zweige visualisieren.

Was im Vergleich zu GitHub oder GitLab fehlt, sind ein Wiki und ein Issue-System. Aber, wie Sie vielleicht schon vermutet haben: Atlassian bietet die Verwendung der anderen Softwareprodukte aus dem eigenen Hause mit sehr guter Integration an. Sie können beides für 7 Tage testen, dann verlangt Atlassian aber Geld dafür. Die Kosten starten bei 10 US$ pro Monat (weniger als 10 Benutzer), wenn Sie Ihre Daten auf Atlassian-Servern speichern. Dafür erhalten Sie eine sehr gut funktionierende Integration von Issue-Tracker und Git-Hosting (siehe Abbildung 7.7).

7.3 Gitea

Die Git-Hosting-Lösungen, die wir Ihnen bisher vorgestellt haben, sind alles Schwergewichte: Aufwendige Weboberflächen, Caches und Datenbanken, die von vornherein viel Ressourcen verschlingen. Gitea geht einen anderen Weg: Das relativ junge Projekt (erstes Release 2016) wurde in der Programmiersprache Go entwickelt und zeichnet sich durch hohe Performance und eine sehr aufgeräumte Weboberfläche aus (siehe Abbildung 7.8).

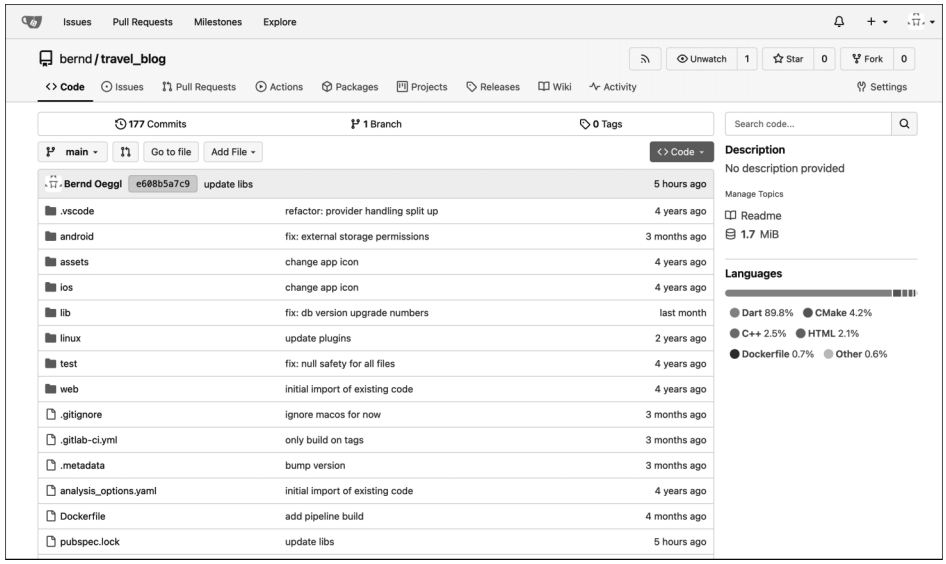


Abbildung 7.8 Die Gitea-Weboberfläche mit einem Beispielprojekt. Die Ähnlichkeiten mit GitHub sind nicht zu verleugnen.

Gitea ausprobieren

Eine Probeinstallation kann sehr einfach vonstattengehen: Sie laden das Binary für Ihre Plattform von GitHub oder über Gitea (*https://dl.gitea.com*) und starten es auf Ihrem Computer. Wenig später können Sie die Adresse *http://localhost:3000* im

Browser öffnen und auf Ihre Gitea-Instanz zugreifen. In der Erstkonfiguration, die vollständig über die Weboberfläche geschehen kann, werden Sie nach den Zugangsdaten zu einem Datenbankserver gefragt (siehe Abbildung 7.9). Für einen Testlauf können Sie das dateibasierte SQLite3-Format wählen und die Einrichtung ohne weitere Änderungen abschließen.

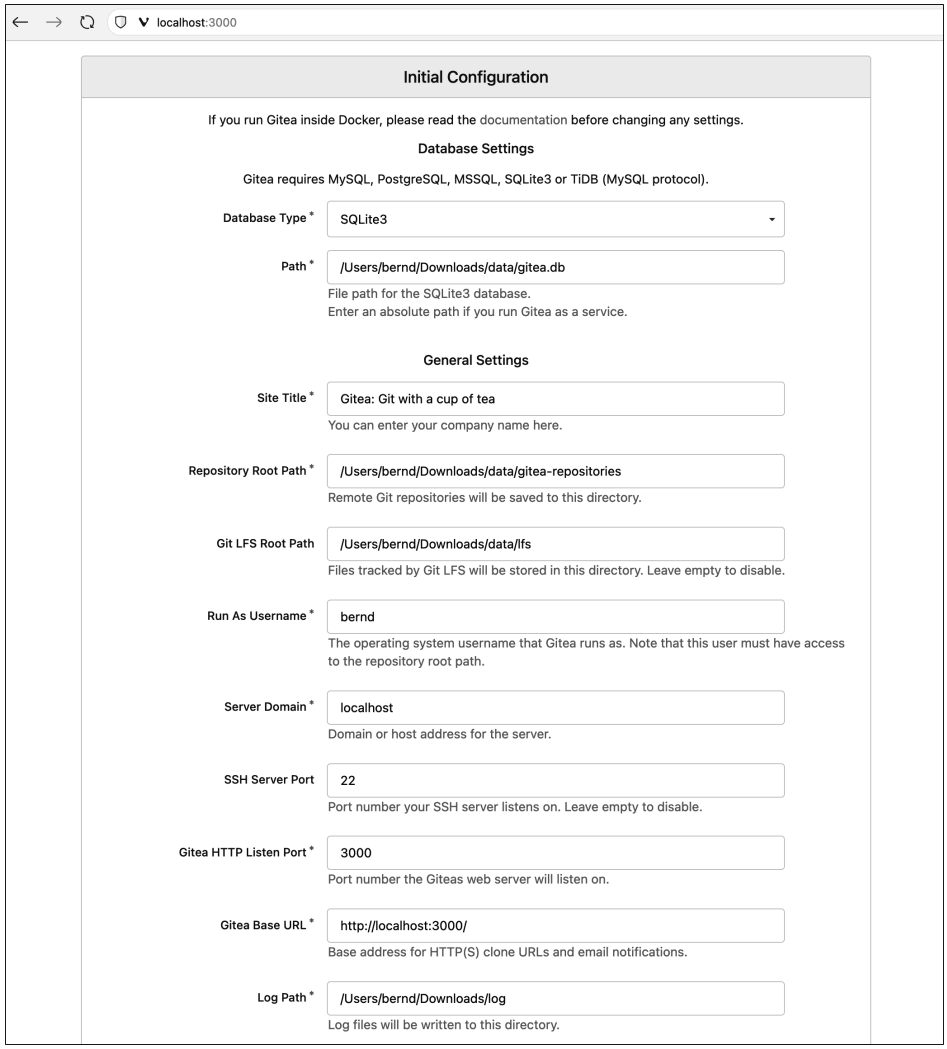


Abbildung 7.9 Die Konfiguration einer lokalen Gitea-Instanz

Unterschätzen Sie Gitea ob des einfachen Setups nicht! Sie haben soeben eine Webapplikation mit einem Ticketsystem, einem Wiki und der Möglichkeit von Pull-Requests installiert. Zwei-Faktor-Authentifizierung mit Einmalpasswörtern oder Hardwareschlüsseln funktioniert ohne weiteren Konfigurationsaufwand. Die Bedie-

nung der Weboberfläche erinnert stark an GitHub, was einen möglichen Umstieg sehr leicht macht.

Forgejo, ein Fork von Gitea

Nachdem im Herbst 2022 die Domainnamen des Gitea-Projekts von einer kommerziellen Firma übernommen wurden, entschlossen sich einige Entwickler, das Projekt zu forken, um eine unabhängige Entwicklung sicherzustellen. Seit 2024 sind die beiden Repositories nicht mehr miteinander verbunden, da die Auffassungen der Teams zu unterschiedlich waren. Forgejo, so der Name des Forks, wird heute unter dem Dach des Vereins *Codeberg e.V.* mit Sitz in Berlin weiterentwickelt. Der Verein bietet auch den Git-Hosting-Service Codeberg (*https://codeberg.org*), der mit Forgejo läuft, an.

Gitea wird inzwischen auch als Cloud-Service angeboten, der Quellcode steht aber nach wie vor auf GitHub unter der freien MIT-Lizenz zur Verfügung.

Serverinstallation mit Docker

In diesem Abschnitt gehen wir davon aus, dass Sie schon einmal mit Docker gearbeitet haben. Das folgende Setup verwendet außerdem `docker compose`, eine Komponente, die Ihnen als Docker-Anwender sicherlich vertraut ist.

Die Entwickler von Gitea unterstützen die Installation mit Docker durch aktuelle Images auf Docker-Hub und die Möglichkeit, den Applikationsserver über Umgebungsvariablen zu konfigurieren. Für uns bedeutet das, dass wir mit einer `compose.yml`-Datei ein fertiges System konfigurieren können, das mit einem Kommando im Produktivbetrieb läuft.

```
# Datei: gitea/docker/compose.yaml
services:
  server:
    image: gitea/gitea:1.23.8
    environment:
      - USER_UID=1000
      - USER_GID=1000
      - DB_TYPE=mysql
      - DB_HOST=db:3306
      - DB_NAME=gitea
      - DB_USER=gitea
      - DB_PASSWD=quaequo5eN6b
      - DISABLE_REGISTRATION=true
      - SSH_PORT=2221
      - SSH_DOMAIN=gitea.git-compendium.info
```

```
    restart: always
    volumes:
      - gitea:/data
    ports:
      - "2221:2221"
      - "3000:3000"
    depends_on:
      - db
  db:
    image: mariadb:10
    restart: always
    environment:
      - MYSQL_ROOT_PASSWORD=aGh3beexOeit
      - MYSQL_USER=gitea
      - MYSQL_PASSWORD=quaequo5eN6b
      - MYSQL_DATABASE=gitea
    volumes:
      - mariavol:/var/lib/mysql
volumes:
  gitea:
  mariavol:
```

In unserem Beispiel verwenden wir die Version `1.23.8` von Gitea für den ersten Dienst, `server`, was Sie in der Zeile `image: gitea/gitea:1.23.8` sehen. Für einen Testbetrieb können Sie hier auch `gitea/gitea:latest` einstellen, wodurch Sie die letzte Entwicklerversion testen.

Die Umgebungsvariablen legen zunächst fest, unter welchem Benutzer und welcher Gruppe der Applikationsserver im Container läuft. Die mit `DB_*` gekennzeichneten Variablen legen die Verbindung zur Datenbank (in diesem Fall MariaDB) fest. `DISABLE_REGISTRATION` unterbindet die Registrierung für alle Benutzer, und mit `SSH_PORT` wird ein abweichender Port für den internen SSH-Server festgelegt, da auf dem Serversystem vermutlich schon ein solcher Dienst läuft. Der `SSH_DOMAIN`-Eintrag wird gebraucht, damit die Links zum Klonen via SSH die korrekte Adresse anzeigen.

Wesentlich bei diesem Setup ist, dass Sie den Ordner `/data` im Container einem Volume zuweisen. Nur so ist sichergestellt, dass Sie bei einem Upgrade Ihre Daten nicht verlieren. Wir verwenden hier ein *Named Volume*, das natürlich in einer automatischen Sicherung eingebunden werden muss. Hier liegen die Git-Repositories, die SSH-Schlüssel und die Konfiguration der Applikation.

Der zweite Dienst, `db`, startet eine Instanz von MariaDB in der Version 10. Die Datenbank-Daten werden ebenfalls in einem Named Volume gespeichert.

Mit dieser Konfiguration läuft Ihr Webserver auf Port `3000` und der SSH-Server auf Port `2221`. Im Docker-Umfeld kommt für verschlüsselte HTTP-Verbindungen oft ein

Reverse Proxy zum Einsatz. Darunter versteht man einen vorgeschalteten Webserver, der die notwendigen Zertifikate verwaltet und den verschlüsselten Verkehr terminiert. Damit diese Konfiguration funktioniert, müssen Sie das Setup mit Ihrem Hostnamen um den Eintrag `ROOT_URL=ihr.hostname.com` im `environment`-Abschnitt des `server`-Services erweitern.

Serverinstallation unter Ubuntu 24.04

Leider gibt es für Gitea keine fertigen Pakete für Linux-Distributionen wie Ubuntu oder Debian. Daher müssen Sie, wenn Sie nicht auf Docker setzen, einige manuelle Anpassungen für den Betrieb von Gitea vornehmen. Wir zeigen Ihnen hier die Vorgehensweise für Ubuntu 24.04.

Auf Ihrem Server muss Git installiert sein, und wenn Sie eine andere Datenbank als SQLite verwenden wollen, brauchen Sie die Zugangsdaten dafür. Aus Sicherheitsgründen empfiehlt es sich, den Applikationsserver nicht als `root`-Benutzer auszuführen. Am besten legen Sie dazu einen eigenen Benutzer an, der sonst keine Rechte am System hat, zum Beispiel den Benutzer `gitea`:

```
sudo adduser --system --shell /bin/bash --group \
  --disabled-password --home /home/gitea gitea
```

Als Speicherplatz für alle Dateien, die Gitea verwaltet, bietet sich unter Ubuntu der Ordner `/var/lib/gitea` an. Legen Sie dort die Unterordner `custom`, `data` und `log` an, und geben Sie dem `gitea`-User und der Gruppe die Rechte auf diese Ordner:

```
sudo mkdir -p /var/lib/gitea/custom /var/lib/gitea/log \
  /var/lib/gitea/data
sudo chown -R gitea:gitea /var/lib/gitea/
sudo chmod -R o-rwx /var/lib/gitea/
```

Die zentrale Gitea-Konfigurationsdatei sollte im Ordner `/etc/gitea` gespeichert werden. Legen Sie abschließend noch diesen Ordner an, und setzen Sie die Berechtigungen so, dass die Gruppe `gitea` dort Schreibrechte hat:

```
sudo mkdir /etc/gitea
sudo chown root:gitea /etc/gitea
sudo chmod o-rwx,ug+rwx /etc/gitea
```

Jetzt fehlen nur noch zwei Sachen: der Gitea-Server selbst und ein Startup-Script, damit der Server bei jedem Neustart automatisch gestartet wird. Laden Sie den Server in den Ordner `/usr/local/bin`, und kennzeichnen Sie diese Datei als ausführbar:

```
sudo wget -O /usr/local/bin/gitea \
  https://dl.gitea.com/gitea/1.23.8/gitea-1.23.8-linux-amd64
sudo chmod 755 /usr/local/bin/gitea
```

Unter Ubuntu kümmert sich `systemd` um das Starten und Beenden von Diensten. Eine minimale Konfigurationsdatei (`gitea.service`) für den Gitea-Dienst sieht zum Beispiel so aus:

```
[Unit]
Description=Gitea
After=syslog.target
After=network.target
[Service]
RestartSec=2s
Type=simple
User=gitea
Group=gitea
WorkingDirectory=/var/lib/gitea/
ExecStart=/usr/local/bin/gitea web --config /etc/gitea/app.ini
Restart=always
Environment=USER=gitea HOME=/home/gitea GITEA_WORK_DIR=/var/lib/
gitea
[Install]
WantedBy=multi-user.target
```

Kopieren Sie die Datei mit dem Namen `gitea.service` in den Ordner `/etc/systemd/system/` auf Ihrem Linux-System, und aktivieren Sie den Dienst mit:

```
sudo systemctl enable gitea --now
```

Ihr Gitea-Server läuft jetzt auf Port `3000`, und sobald Sie auf Anmelden oder Registrieren klicken, werden Sie auf die Installationsseite geleitet. Wenn Sie auf diesem Server keine anderen Webdienste aktiviert haben, können Sie Gitea auch auf dem Standard-Port für sicheres HTTP laufen lassen, und Gitea kann sogar selbst Zertifikate bei *Let's Encrypt* für Sie erstellen.

Damit HTTPS auf dem dafür vorgesehenen Standard-Port `443` mit den Zertifikaten funktioniert, müssen Sie noch zwei kleine Änderungen in den Dateien vornehmen: In der Servicedatei für `systemd` erteilen Sie dem Benutzer `gitea` die Berechtigungen, die Ports `80` und `443` zu verwenden. Dazu fügen Sie folgende zwei Zeilen in der `[Service]`-Sektion ein:

```
CapabilityBoundingSet=CAP_NET_BIND_SERVICE
AmbientCapabilities=CAP_NET_BIND_SERVICE
```

Anschließend müssen Sie den `systemd`-Prozess mit dem Kommando `systemctl daemon-reload` neu starten. In der Konfiguration des Applikationsservers fehlen nun noch die Einträge für HTTPS und Let's Encrypt. Fügen Sie folgende Zeilen am Anfang der Datei `/etc/gitea/app.ini` ein:

```
[server]
PROTOCOL=https
DOMAIN=tea.git-compendium.info
HTTP_PORT = 443
ENABLE_LETSENCRYPT=true
LETSENCRYPT_ACCEPTTOS=true
LETSENCRYPT_DIRECTORY=https
LETSENCRYPT_EMAIL=root@git-compendium.info
```

`DOMAIN` und `LETSENCRYPT_EMAIL` müssen Sie natürlich an den wirklichen Domainnamen Ihres Servers anpassen. Wenn Sie jetzt noch den Gitea-Prozess neu starten (`systemctl restart gitea.service`), wird sich Gitea um die Zertifikate kümmern, und nach kurzer Zeit können Sie mit HTTPS auf Ihren Server zugreifen.

Wenn Sie Gitea produktiv einsetzen möchten, empfehlen wir Ihnen ein anderes Datenbanksystem als SQLite. Um MariaDB mit dieser Installation unter Ubuntu 24.04 zu verwenden, reichen folgende Kommandos:

```
apt install mariadb-server
mysqladmin create gitea
mysql gitea -e "GRANT ALL PRIVILEGES ON gitea.* TO \
  gitea@localhost IDENTIFIED BY 'einohD8ith3I'"
```

Sie können jetzt in der Weboberfläche bei der Datenbankeinstellung die voreingestellte Auswahl für MySQL belassen und als Passwort `einohD8ith3I` beziehungsweise die von Ihnen gewählte Zeichenkette eintragen.

Ein erstes Beispiel mit Gitea

Um ein paar der Funktionen von Gitea zu demonstrieren, werden wir ein bereits bestehendes Projekt in Gitea importieren. Anders als GitHub oder GitLab bietet Gitea keinen Importer für Repositories der Konkurrenten an. Ein bestehendes Git-Repository können wir aber auf jeden Fall importieren.

Wir legen dazu ein neues Projekt mit dem Namen `pictures` über die Weboberfläche auf dem Gitea Server an (siehe Abbildung 7.10). Es wird die Node.js-Applikation enthalten, die wir schon in Abschnitt 5.4, »Automatische Sicherheits-Scans«, verwendet haben. Anschließend klonen wir das bestehende GitHub-Repository auf den lokalen Computer mit dem `--mirror`-Flag.

```
git clone --mirror https://gitlab.com/git-compendium/pictures.git

  Cloning into bare repository 'pictures.git'...
  ...
```

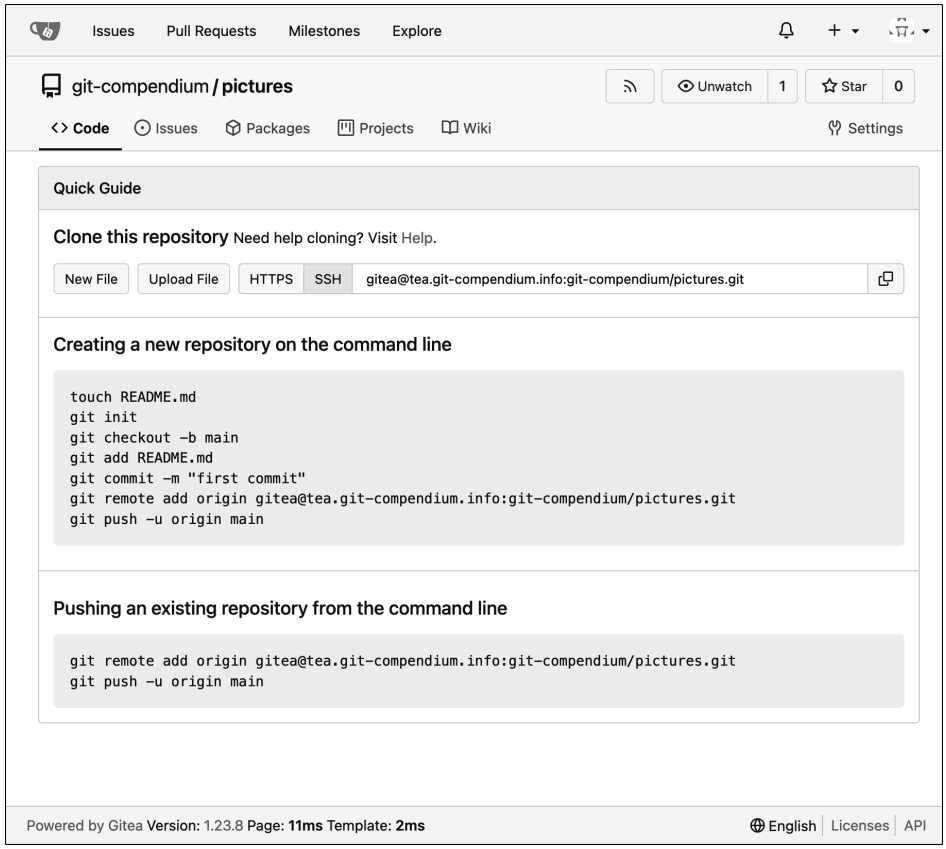


Abbildung 7.10 Das neue Gitea-Repository für das Bilddatenbank-Beispiel

Wir haben jetzt ein `bare`-Repository in dem Ordner `pictures.git` erzeugt, das alle Referenzen wie Tags und Remote Tracking-Branches enthält. Dieses Repository werden wir jetzt in das auf Gitea neu erzeugte Projekt kopieren. Wechseln Sie dazu in den neuen Ordner, und rufen Sie `git push` mit der Mirror-Option auf:

```
git push --mirror gitea@tea.git-compendium.info:git-compendium/pi
ctures.git
  Enumerating objects: 229, done.
  ...
  To tea.git-compendium.info:git-compendium/pictures.git
   * [new branch]      main -> main
```

Jetzt können wir mit Gitea loslegen und erstellen als Erstes einen ersten Eintrag (*Issue*) im Ticketsystem. Der Vorschlag zur Verbesserung (*Feature-Request*) lautet, dass das `Dockerfile` in dem Projekt so umgebaut wird, dass es Multi-Stage-Builds für Entwicklung und den Produktivbetrieb unterstützt.

Wie Ihnen sicher bereits aufgefallen ist, unterstützt Gitea in der Weboberfläche mehrere Sprachen. Wenn Ihr Betriebssystem auf die deutsche Sprache eingestellt ist, sehen Sie auch die Menüs in Gitea auf Deutsch – ein Luxus, den sich weder GitHub noch GitLab bislang leisten.

Um im Ticketsystem den Überblick nicht zu verlieren, können Sie einem Eintrag ein oder mehrere *Labels* und einen Milestone zuweisen. Labels müssen bei neuen Projekten entweder manuell erstellt werden, oder Sie importieren ein vorgegebenes Label-Set, das aus sieben praktischen Labels wie *bug*, *duplicate* oder *wontfix* besteht.

Wir entwickeln jetzt lokal die gewünschten Änderungen auf dem Git-Branch `multistage-dev`.

```
git clone gitea@tea.git-compendium.info:git-compendium/pictures
  Cloning into 'pictures'...
  ...
  Resolving deltas: 100% (102/102), done.

cd pictures

git checkout -b multistage-dev
  Switched to a new branch 'multistage-dev'
```

Nachdem wir die Änderungen in der Datei `Dockerfile` gemacht haben, committen und pushen wir den Feature-Branch. Beachten Sie dabei den Verweis in der Commit-Message auf den Issue im Ticketsystem (`see #1`):

```
git commit -am "feat: multistage build for Docker image (see #1)"
  [multistage-dev fe4f316] feat: multistage build for Docker i...
   1 file changed, 13 insertions(+), 3 deletions(-)

git push --set-upstream origin multistage-dev
  Enumerating objects: 5, done.
  Counting objects: 100% (5/5), done.
  Delta compression using up to 12 threads
  Compressing objects: 100% (3/3), done.
  Writing objects: 100% (3/3), 339 bytes | 339.00 KiB/s, done.
  Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
  remote:
  remote: Create a new pull request for 'multistage-dev':
  remote:   https://tea.git-compendium.info/git-compendium/pic...
  remote:
  remote: . Processing 1 references
  remote: Processed 1 references in total
  To tea.git-compendium.info:git-compendium/pictures
   * [new branch]      multistage-dev -> multistage-dev
  branch 'multistage-dev' set up to track 'origin/multistage-d...
```

Öffnen Sie nun den Link zum Erzeugen eines neuen Pull-Requests, der in der Konsole nach dem Push-Kommando ausgegeben wurde. In der Weboberfläche sehen Sie die Änderungen und einen grünen Button, mit dem Sie den Pull-Request wirklich erstellen (siehe Abbildung 7.11).

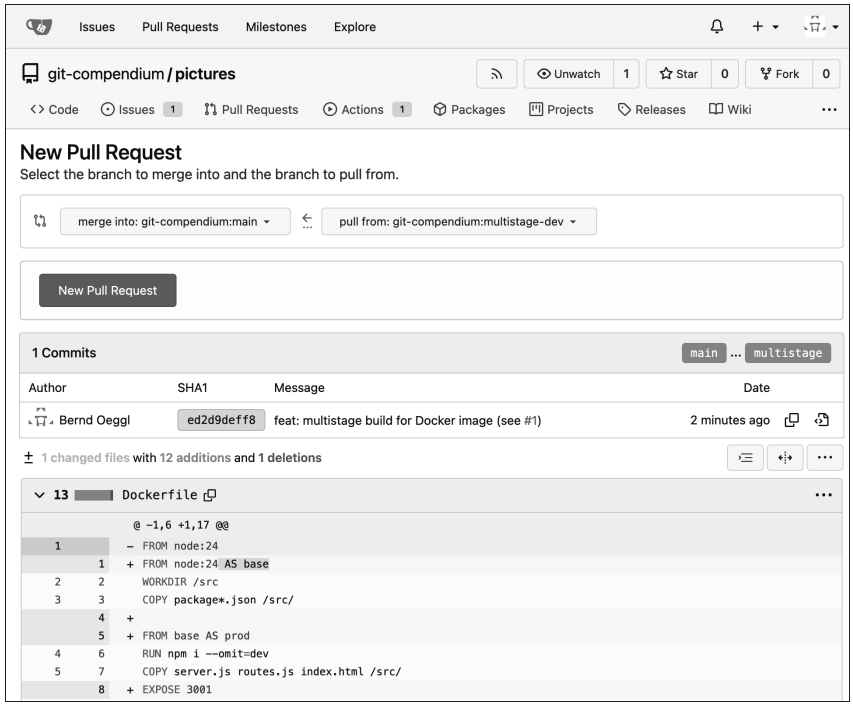


Abbildung 7.11 Der neu angelegte Pull-Request in Gitea

Die Aufgabe, den Pull-Request zu begutachten und zu akzeptieren oder abzulehnen, kommt in einem Team normalerweise einer anderen Person zu. In unserem Beispiel machen wir das aber gleich selbst. Wir akzeptieren den Pull-Request mit der Voreinstellung Pull-Request zusammenführen. Dadurch wird ein neuer Commit erzeugt, der den Merge des Feature-Branches anzeigt. Hätten wir die Option Rebase und Merge aus dem Dropdown gewählt, würde dieser Eintrag in der Commit-History nicht erscheinen. Nach dem Zusammenführen können wir den Branch `multistage-dev` direkt mit dem entsprechenden Button löschen (siehe Abbildung 7.12).

Der Issue im Ticketsystem enthält Referenzen zu allen Vorgängen, die mit dem Pull-Request in Zusammenhang standen, da die Commit-Message den Verweis auf den Issue enthält (siehe Abbildung 7.13).

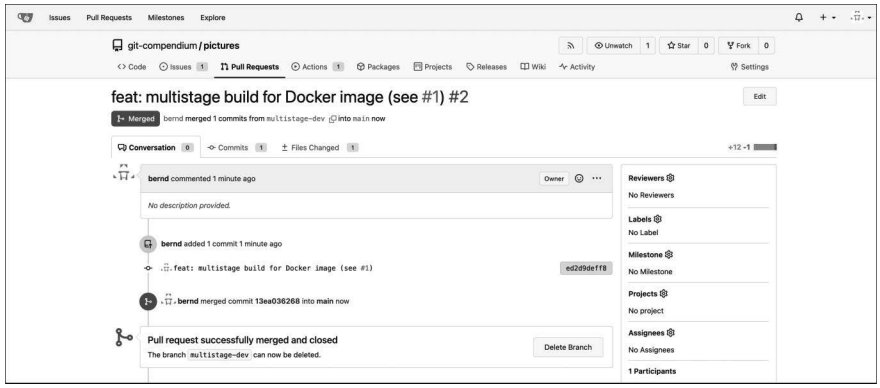


Abbildung 7.12 Ein erfolgreich durchgeführter Pull-Request in Gitea

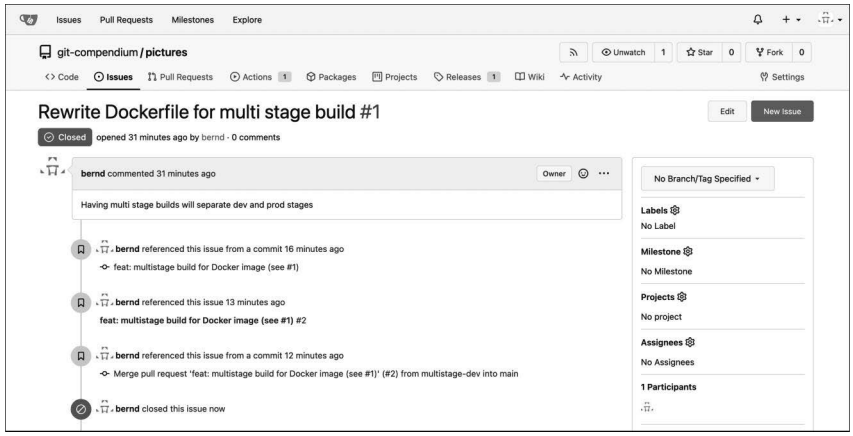


Abbildung 7.13 Der geschlossene Issue #1 im Ticketsystem mit der gesamten Merge-History

Zum Abschluss dieses kurzen Beispiels werden wir noch ein Release von unserer Software erstellen. Wir klicken dazu auf RELEASES • NEW RELEASE. Als Tag-Name verwenden wir `v1.0.0` und als Titel `Docker multistage`. Gitea erzeugt darauf das gewünschte Tag und zwei komprimierte Dateien, einmal das vor allem unter Windows übliche `.zip`-Format und eine `.tag.gz`-Datei, die unter Linux und macOS gebräuchlicher ist (siehe Abbildung 7.14).

Wenn wir auf unserem lokalen Computer ein `git pull` ausführen, wird das neue Tag heruntergeladen:

```
git pull

  remote: Enumerating objects: 1, done.
  ...
   * [new tag]         v1.0.0     -> v1.0.0
```

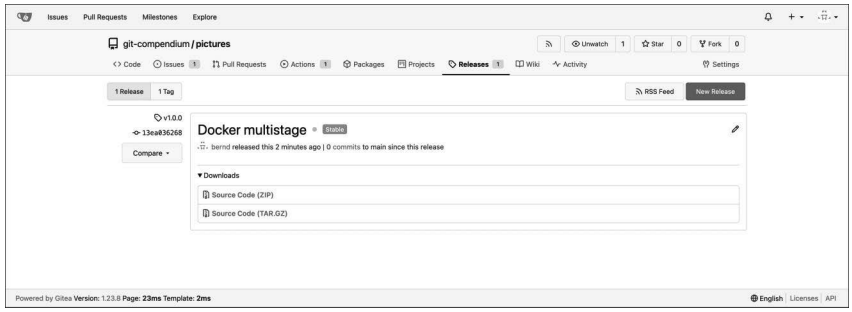


Abbildung 7.14 Das Release mit dem Tag »v1.0.0« in Gitea

7.4 Gitolite

Mit Gitolite stellen wir Ihnen das zurzeit schlankeste Programm für ein eigenes Git-Hosting vor. Bei diesem Programm spielt sich alles auf der Kommandozeile ab. Es gibt keine Weboberfläche und natürlich auch kein Ticketsystem, Wiki oder Ähnliches. Dafür sind die Softwareanforderungen minimal:

- OpenSSH Server
- Git
- Perl (mit dem JSON-Modul, sofern Sie diese Ausgabe wünschen)

Da das System auf eine Datenbank oder einen Applikationsserver verzichtet, sind die Hardwareanforderungen ebenfalls sehr gering: Solange der SSH-Server läuft und der Zugriff auf den permanenten Speicher flott ist, werden Sie für Ihre privaten Repositories keine Performanceprobleme haben. Daher können Sie auch Miniserver wie einen Raspberry Pi gut als Gitolite-Server verwenden.

Installation

Anders als die bisher vorgestellten Git-Hosting-Programme wickelt Gitolite den Remote-Zugriff auf Git-Repositories ausschließlich über SSH ab. Alle SSH-Zugriffe erfolgen dabei über einen Benutzeraccount auf dem Linux-System, und dieser verwaltet die Rechte für die Projekte. Das machen übrigens die anderen vorgestellten Hosting-Programme auch so. Sie erkennen das an der Adresse des Remote Git-Repositorys: Mit der Adresse `git@gitolite.git-compendium.info:gitolite-first` verbinden Sie sich als Benutzer `git` mit dem Server `gitolite.git-compendium.info` und verwenden das Repository `gitolite-first`.

Für eine möglichst unkomplizierte Installation legen Sie diesen Benutzer neu auf Ihrem Server an, und überlassen Sie Gitolite die Initialisierung der SSH-Einstellungen. Sollten Sie mit Docker vertraut sein, werfen Sie einen Blick in das Repository *https://github.com/git-compendium/gitolite-docker*; Sie finden dort ein Docker-Setup, das

den Gitolite-Server mit einem Kommando startet. Auf einem Ubuntu- oder Debian-Server können Sie die manuelle Einrichtung zum Beispiel so bewerkstelligen:

```
sudo useradd -m git
sudo su - git
git clone https://github.com/sitaramc/gitolite
mkdir /home/git/bin
/home/git/gitolite/install -ln
```

Damit sind Sie schon fast fertig mit der Installation. Es fehlt noch der Administrator-Benutzer für Ihre Git-Repositories. Erzeugen Sie sich dazu einen SSH-Schlüssel auf Ihrer Workstation/Laptop (oder verwenden Sie einen bestehenden).

```
ssh-keygen -f ~/.ssh/gitoliteroot -N ''
scp ~/.ssh/gitoliteroot.pub gitolite.git-compendium.info:/tmp
```

Kopieren Sie anschließend den öffentlichen Teil des Schlüssels auf den Server, und schließen Sie die Installation ab (immer noch als Benutzer `git`):

```
/home/git/bin/gitolite setup -pk /tmp/gitoliteroot.pub

  Initialized empty Git repository in
    /home/git/repositories/gitolite-admin.git/
  Initialized empty Git repository in
    /home/git/repositories/testing.git/
  WARNING: /home/git/.ssh missing; creating a new one
      (this is normal on a brand new install)
  WARNING: /home/git/.ssh/authorized_keys missing; creating a ...
      (this is normal on a brand new install)
```

Anwendung

Der Gitolite-Server ist jetzt bereit. Wie Sie im oben stehenden Listing sehen, wurden bereits zwei Git-Repositories angelegt: `gitolite-admin` und `testing`. Das Anlegen von Benutzern oder neuen Git-Repositories geschieht durch die Veränderung des `gitolite-admin`-Repositorys. Klonen Sie das Repository auf Ihrem Laptop oder der Workstation, auf der Sie zuvor die SSH-Schlüssel erzeugt haben:

```
git clone git@gitolite.git-compendium.info:gitolite-admin

  Cloning into 'gitolite-admin'...
  ...
```

In dem neuen Verzeichnis befinden sich die Ordner `conf` und `keydir`, wobei in letzterem bereits der öffentliche SSH-Schlüssel für `gitoliteroot` liegt, den Sie beim Setup importiert haben.

```
tree --charset=ascii
  .
  |-- conf
  |   `-- gitolite.conf
  `-- keydir
      `-- gitoliteroot.pub
```

Um neue Benutzer anzulegen, kopieren Sie einfach deren öffentliche SSH-Schlüssel in den `keydir`-Ordner. Beachten Sie, dass der Name der Datei dem Benutzernamen entspricht. Zum Anlegen neuer Git-Repositories editieren Sie die Datei `conf/gitolite.conf` und fügen einen neuen `repo`-Eintrag hinzu:

```
repo gitolite-first
    RW+     =  gitoliteroot
```

Damit wird das Repository `gitolite-first` angelegt, der Benutzer `gitoliteroot` bekommt Lese- und Schreibrechte. Committen und pushen Sie die Änderungen:

```
git commit -a -m "add gitolite-first repo"

  [master 0fcc024] add gitolite-first repo
   1 file changed, 3 insertions(+)

git push

  Enumerating objects: 7, done.
  Counting objects: 100% (7/7), done.
  Delta compression using up to 8 threads
  Compressing objects: 100% (3/3), done.
  Writing objects: 100% (4/4), 393 bytes | 393.00 KiB/s, done.
  Total 4 (delta 0), reused 0 (delta 0)
  remote: Initialized empty Git repository in
    /home/git/repositories/gitolite-first.git/
  To gitolite:gitolite-admin
     7b33ac2..bb8b85f  master -> master
```

In der Ausgabe von `git push` erkennen Sie, dass Gitolite ein neues Git-Repository für Sie angelegt hat. Beachten Sie, dass in der aktuellen Version von Gitolite der Branch mit dem Namen `master` verwendet werden muss. An vielen Stellen im Perl-Quellcode des Programms wird dieser Name referenziert.

Fazit

Obwohl beide Autoren Freunde der Kommandozeile sind, haben wir Gitolite doch als *sehr* spartanisch empfunden. Die Weboberflächen von GitHub, GitLab oder Gitea bieten hier deutlich mehr Komfort.

Inhalt

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

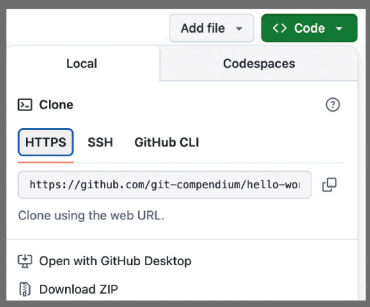
U

Git für Entwickler und DevOps-Teams

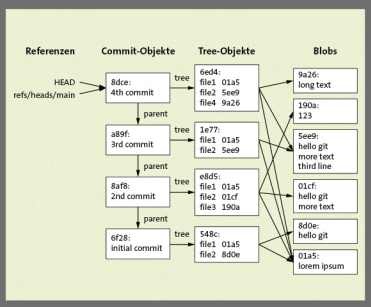
»Software-entwicklung im Team«

Git in der Praxis

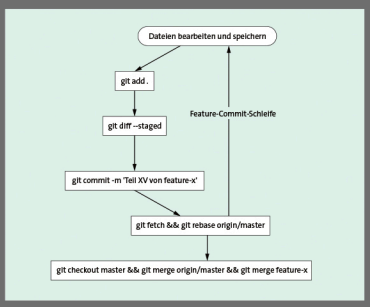
Git darf im Werkzeugkasten eines IT-Profis nicht fehlen. Ob Sie im Team komplexe Anwendungen entwickeln, Konfigurationsdateien versionieren oder an Open-Source-Projekten mitarbeiten: Dieses Buch zeigt Ihnen, wie Sie die Features von Git richtig einsetzen und mit den passenden Workflows den Überblick behalten.



Git, GitHub und GitLab



Git verstehen und einsetzen



Tutorials und Praxistipps

Softwareprojekte verwalten

Auch wenn Ihr Projekt ein bisschen kleiner als der Linux-Kernel ist, werden Sie von einer Versionskontrolle Ihres Codes profitieren. Und wenn Sie im Team an einer großen Codebasis arbeiten, sind Feature-Branches, Pull-Requests & Co. erst recht unverzichtbar.

Workflows für den Arbeitsalltag

Sprechende Commit-Messages sind ein erster Schritt, aber es gibt noch mehr Tricks, mit denen Git Ihren Entwickleralltag bereichert. Mit git blame und git bisect kommen Sie Bugs auf die Spur und behalten auch in großen Projekten den Überblick.

Kommandoreferenz und Tutorials

Der Einstieg in Git ist leicht, denn schon mit wenigen Kommandos klonen Sie Repositories und verwalten Änderungen. Und was Git sonst noch alles kann, finden Sie hier auf einen Blick.

Alle Beispiele zum Download und Klonen auf GitHub

Bernd Öggl und **Michael Kofler** nutzen Versionskontrollsysteme für alle Projekte. Was ihnen an Git besonders gut gefällt, zeigen sie Ihnen in diesem Leitfaden mit vielen Beispielen und Tipps.

Aus dem Inhalt

Rheinwerk Computing

Gedruckt in Deutschland
Mineralölfreie Druckfarben
Zertifiziertes Papier

€ 39,90 [D] € 41,10 [A]

Softwareentwicklung
ISBN 978-3-367-10812-1