mitp

Sujeevan
Vijayakumaran

2. Auflage

Versionsverwaltung mit Git

Praxiseinstieg

Inhaltsverzeichnis

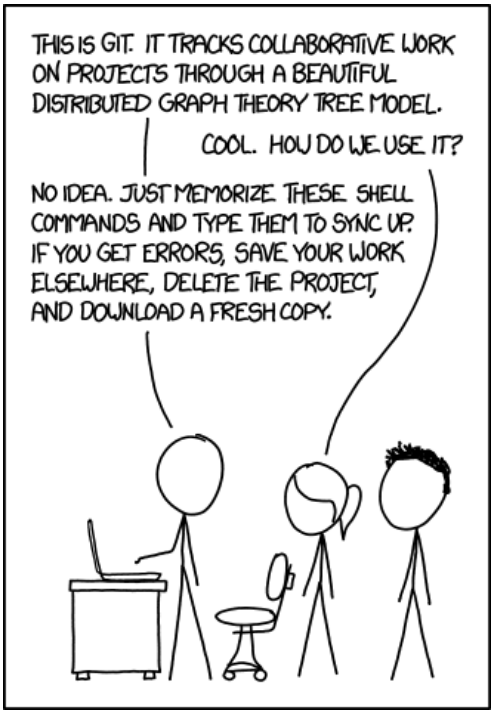
Danksagung

Mein Dank für die zweite Auflage geht diesmal an meine Eltern und an meine Schwester Shrani, die mich bei meinem beruflichen Werdegang stets unterstützt haben und verständnisvoll waren, wenn ich doch mehr Zeit für die Arbeit an diesem Buch brauchte.

Weiterhin gilt mein Dank meiner Lektorin Sabine und an mitp-Verlag für die stets gute Zusammenarbeit in den letzten vier Jahren und für die Möglichkeit, eine zweite Auflage zu veröffentlichen. Mein Dank gilt auch noch Philip Haas, der ein sehr schönes Coverbild gebastelt hat, und Dirk Deimeke, der mir auch in der zweiten Auflage weiterhin fachlich bestens beratend zur Seite stand. Zum Schluss gilt mein Dank noch allen Personen, die Tippfehler und auch kleinere inhaltliche Fehler gefunden und gemeldet haben, allen voran Peter Sicarevic, der den aktuellen High Score an gefundenen Fehlern hält.

Nicht zu vergessen ist ebenfalls die Unterstützung für die erste Auflage, denn an einem Buch arbeitet selten eine Person vollständig allein. An dieser Stelle möchte ich daher Thomas Schmidt und Sarah Blume danken, die sich die Zeit genommen haben, mir gutes Feedback zu meinem Manuskript zu geben, das in das Buch eingeflossen ist. Weiterhin gilt mein Dank auch Dirk Deimeke für die initiale Idee und Motivation, dieses Buch zu schreiben, sowie Dr. Veit Jahns von der otris software AG aus Dortmund, wo ich im Wesentlichen mein Wissen von Git erwerben, anwenden und neue Ideen einbringen konnte. Zum Schluss geht mein Dank noch an das Team von ubuntuusers.de und freiesMagazin.de, ohne die ich wohl nie ein Buch geschrieben hätte.

Einleitung



Er: »Das ist Git. Es bietet einen Überblick über die kollaborative Arbeit in Projekten durch die Nutzung eines wunderschönen Graphen-Theorie-Modells.«

Sie: »Cool. Aber wir nutzt man es?«

Er: »Keine Ahnung. Merke dir einfach all diese Befehle und tippe sie ein. Wenn du auf Fehler stößt, dann sichere deine Arbeit woanders hin, lösche das Projekt und lade eine frische Kopie herunter.«

»If that doesn't fix it, git.txt contains the phone number of a friend of mine who understands git. Just wait through a few minutes of 'It's really pretty simple, just think of branches as...' and eventually you'll learn the commands that will fix everything.«

Und wenn das auch nicht hilft, dann enthält git.txt die Telefonnummer von einem Freund, der sich mit Git auskennt. Warte einfach ein paar Minuten von »Es ist wirklich gar nicht so schwer, stell dir nur die Branches als ...« ab und irgendwann lernst du die Befehle, die jedes Problem fixen.

»xkcd: Git«, Copyright Randall Munroe (`https://xkcd.com/1597/`*) ist lizenziert unter der Creative Commons Lizenz CC BY-NC 2.5 (*`https://creativecommons.org/licenses/by-nc/2.5/`*)*

Versionskontrolle ist ein wichtiges Thema für Software-Entwickler. Jeder, der ohne jegliche Versionskontrollprogramme arbeitet, ist vermutlich schon einmal an den Punkt gestoßen, wo man sich ältere Stände ansehen wollte. Dabei fragt man sich gegebenenfalls, warum und wann man eine Funktion eingeführt hat, oder man möchte auf einen älteren Stand zurückspringen, wenn man etwas kaputt gemacht hat. Genau an dieser Stelle kommen Versionsverwaltungsprogramme ins Spiel. Git ist eines dieser Programme, die nicht nur die bereits genannten Probleme lösen. Es ist Kernbestandteil des Entwicklungsprozesses, um sowohl kollaborativ im Team als auch alleine an einem Projekt zu arbeiten. Dabei ist es gleichgültig, ob man Programmierer, System-Administrator oder gar Autor ist.

Randall Munroe beleuchtet in seinem Webcomic xkcd viele verschiedene Themen. Das hier abgedruckte xkcd-Comic zum Thema Git wurde während meiner Arbeit an diesem Buch veröffentlicht. Viele meiner Freunde und Bekannten aus dem Open-Source-Umfeld posteten das Comic in den verschiedenen sozialen Netzwerken und machten eins deutlich: Viele Leute nutzen Git zwar, wissen aber nur grob, was dort passiert. Wenn etwas nicht wie geplant funktioniert oder man zu einem fehlerhaften Zustand im Arbeitsprojekt kommt, dann weiß man erst mal nicht weiter und fragt seinen persönlichen Git-Experten, wie den einen Kollegen, der glücklicherweise ein Git-Buch geschrieben hat.

Das Ziel dieses Buches ist nicht nur, dass Sie die gängigen Befehle erlernen, die beim Arbeiten mit Git gebraucht werden, sondern ich lege auch großen Wert auf die Einbindung und Anpassung des Entwicklungsprozesses. Darüber hinaus sollten Sie Git als Ganzes verstehen und nicht nur die Grundlagen, damit Sie mit einem Programm arbeiten, das Sie verstehen und bei dem bei Konflikten keine Hürden vorhanden sind.

Aufbau des Buches

Dieses Buch besteht aus insgesamt zwölf Kapiteln, davon gehören die ersten vier Kapitel zu den Grundlagen und die übrigen acht zu den fortgeschrittenen Themen.

Das erste Kapitel führt in das Thema der Versionsverwaltung mit Git ein, um den Einsatzzweck und die Vorteile von Git zu verdeutlichen. Das zweite Kapitel behandelt die grundlegenden Git-Kommandos. Dies beinhaltet die Basis-Befehle, die für das Arbeiten mit Git notwendig sind. Im anschließenden dritten Kapitel geht es um die Nutzung von Branches, eines der elementaren Features von Git. So lernen Sie, mit Branches parallele Entwicklungslinien zu erstellen, zwischen diesen verschiedenen Branches hin und her zu wechseln und sie wieder zusammenzuführen. Der Grundlagenteil endet mit dem vierten Kapitel, bei dem es um den Einsatz von verteilten Repositorys geht, die es ermöglichen mit Repositorys zu arbeiten, die auf entfernten Servern, wie etwa GitHub, liegen.

Bei den fortgeschrittenen Themen liegt der Fokus besonders auf dem Einsatz von Git in Software-Entwicklungsteams. Wichtig ist dabei, über eine gute Möglichkeit zu verfügen, Git-Repositorys hosten zu können, damit man kollaborativ in einem Team an Projekten arbeiten kann. Während die wohl gängigste, bekannteste und einfachste Hosting-Möglichkeit GitHub ist, gibt es auch einige Open-Source-Alternativen, wie zum Beispiel GitLab, die sich ebenfalls gut für den Einsatz in Firmen oder anderen Projektgruppen eignen. Das ist das Thema im fünften Kapitel, in dem auch der Workflow bei GitHub und GitLab thematisiert wird. Im anschließenden sechsten Kapitel geht es um die verschiedenen existierenden Workflows. Um die Features von Git sinnvoll einzusetzen, sollten Sie einen Workflow nutzen, der sowohl praktikabel ist als auch nicht zu viel Overhead im Projekt führt. Die Art und Weise, mit Git zu arbeiten, unterscheidet sich vor allem bei der Anzahl der Personen, Branches und Repositorys. Im siebten Kapitel geht es im Anschluss darum, Git-Hooks zu verwenden, um mehr aus dem Projekt herauszuholen oder simple Fehler automatisiert zu überprüfen und somit zu vermeiden. So lernen Sie, was Hooks sind, wie sie programmiert werden und damit zu automatisieren. Generell ist dieses Kapitel für den Git-Nutzer kein alltägliches Thema. Hooks werden im Alltag eher unregelmäßig programmiert.

Die weiteren drei Kapitel befassen sich mit dem Umstieg von Subversion nach Git, wobei sowohl die Übernahme des Quellcodes inklusive der Historie als auch die Anpassung des Workflows thematisiert wird. Das neunte Kapitel ist eine Sammlung vieler verschiedener nützlicher Tipps, die zwar nicht zwangsläufig täglich gebraucht werden, aber trotzdem sehr nützlich sein können. Im zehnten Kapitel folgt dann noch ein Kapitel mit einem Überblick über die grafischen Git-Programme unter den verschiedenen Betriebssystemen Windows, Mac OS X und Linux.

Komplett neu in der zweiten Auflage sind Kapitel 11 und 12. Hier werden zum einen nützliche Hilfestellungen gegeben, um eine möglichst nachvollziehbare Git-Historie zu erzeugen, und zum anderen werden häufige Probleme von Anfängern und Erfahrenen beleuchtet und die dazugehörigen Lösungen aufgezeigt.

Um den Einsatz von Git und die einzelnen Funktionen sinnvoll nachvollziehen zu können, werden alle Git-Kommandos anhand eines realen Beispiels erläutert. Über die Kapitel des Buches hinweg entsteht eine kleine statische Webseite, an der die Funktionen verdeutlicht werden. Denn was bringt es, die Kommandos von Git ohne den Bezug zu realen Projekten und dessen Einsatzzwecke zu kennen? Eine kleine Webseite hat insbesondere den Vorteil, dass Sie nicht nur Unterschiede im Quellcode nachvollziehen, sondern auch sehr einfach die optischen Unterschiede auf einer Webseite erkennen können.

Konvention

In diesem Buch finden Sie zahlreiche Terminal-Ausgaben abgedruckt. Diese sind größtenteils vollständig, einige mussten aus Platz- und Relevanz-Gründen jedoch gekürzt werden. Eingaben in der Kommandozeile fangen immer mit dem »$« an. Dahinter folgt dann der eigentliche Befehl. Das Dollarzeichen ist der Prompt, der in der Shell dargestellt wird, und es muss daher nicht eingetippt werden. Zeilen, die kein solches Zeichen besitzen, sind Ausgaben der Befehle. Das sieht dann etwa so aus:

```
$ git log
commit 9534d7866972d07c97ad284ba38fe84893376e20
[...]
```

Zeilen, die nicht relevant sind oder verkürzt wurden, sind als »`[...]`« dargestellt.

Hinweise und Tipps

Die einzelnen Kapitel bauen zwar aufeinander auf, doch ist es nicht immer möglich, alle Themen an Ort und Stelle ausführlich zu behandeln. Zudem werden wohl eher wenige Leser das Buch von vorne bis hinten durcharbeiten. Das Buch beinhaltet daher einige Hinweise und Tipps. Teilweise sind es Hinweise auf nähere Details in anderen Teilen des Buches, teilweise Tipps und Warnungen für die Nutzung von Git. Dies sind häufig nützliche Inhalte, die sich auf das gerade behandelte Thema beziehen, hin und wieder aber auch Querverweise zu näheren Erläuterungen in anderen Kapiteln.

Feedback

Als Autor habe ich sehr wohl den Anspruch, dass Sie als Leser das, was in diesem Buch behandelt wird, sowohl richtig verstehen als auch anwenden können. Ich bin daher offen für Feedback und Verbesserungsvorschläge – entweder per Mail an `mail@svij.org` oder Kurzes gerne auch via Twitter an `@svijee` (`https://twitter.com/svijee`).

Kapitel 2

Die Grundlagen

Dieses Kapitel beinhaltet die grundlegenden Funktionen und Kommandos von Git. So gut wie alle in diesem Kapitel behandelten Befehle dürften beim täglichen Arbeiten mit Git zum Einsatz kommen. Damit Sie den Sinn und Zweck einzelner Befehle und Funktionen von Git sowohl nutzen als auch nachvollziehen können, arbeitet dieses Kapitel hauptsächlich an einem Beispielprojekt, das die Nutzung und Arbeitsweise von Git verdeutlicht.

Das Beispiel-Projekt ist eine kleine Webseite, die nach und nach aufgebaut wird. HTML-Kenntnisse sind prinzipiell nicht notwendig, können aber natürlich auch nicht schaden. Damit es nicht ganz so trocken und langweilig ist, empfehle ich Ihnen, die Beispiele auf dem eigenen Rechner nachzumachen. An der ein oder anderen Stelle bietet es sich auch an, etwas herumzuexperimentieren, denn nur durch Praxis wird Ihnen das Arbeiten mit Git klar und Sie haben hinterher in echten Projekten keine großen Probleme.

Als Beispiel wird eine kleine persönliche Webseite mit dem HTML5-Framework »Bootstrap« erstellt. Auf die genaue Funktionsweise des Frameworks gehe ich nicht näher ein, da es sich hier ja um Git und nicht um HTML und CSS dreht.

Git ist traditionell ein Kommandozeilenprogramm, weshalb der Fokus auf der Arbeit mit Git in der Kommandozeile liegt. Einschübe mit grafischen Git-Programmen gibt es dennoch. Grafischen Git-Programmen ist mit Kapitel 10, »Grafische Clients« ein vollständiges Kapitel gewidmet.

2.1 Installation

Bevor Sie losgelegen, muss Git installiert werden. Git gibt es für die Betriebssysteme Windows, Mac OS X und Linux. Die gängigen Linux-Distributionen stellen Git unter dem Paketnamen »git« in der Paketverwaltung zur Verfügung. Nutzer von Windows und Mac OS X können sich Git von der Projektwebseite `https://git-scm.com/downloads` herunterladen.Während der Arbeit an diesem Buch ist die Git-Version 2.21 die neueste Version. Große Unterschiede zu den vorherigen Versionen seit 2.0 existieren hingegen nicht, es sind vielmehr zahlreiche Kleinigkeiten, die über die Zeit eingeflossen sind. Bei Bedarf werden neue Funktionen aus vergleichsweise neuen Versionen hervorgehoben. Gleiches gilt für möglicher-

weise ältere Versionen von Git, die sich noch in den Paketverwaltungen älterer Linux-Distributionen finden. Es gibt einige kleinere Unterschiede bei der Benutzung der Git-Versionen vor und nach Version 2.0. An den Stellen im Buch, an denen es zu Abweichungen bei verschiedenen Git-Versionen kommt, habe ich zusätzlich hervorgehoben, was Sie beachten müssen und wo die Unterschiede liegen.

Seit der Version 2.5 hat Git unter Windows keinen Preview-Status mehr, sondern ist als vollwertige stabile Version verfügbar.

Die Installation unter Windows ist größtenteils selbsterklärend. Ein paar Kleinigkeiten gibt es aber doch zu beachten. Ein Punkt bei der Installation ist die Abfrage des genutzten Editors. In früheren Versionen wurde automatisch der Konsolen-Texteditor »vim« installiert und konfiguriert. Dieser ist vor allem für gängige Windows-Nutzer ohne Erfahrung in der Nutzung von »vim« eine schwierige Wahl. Eine Konfiguration eines anderen Editors war zuvor erst nachträglich möglich.

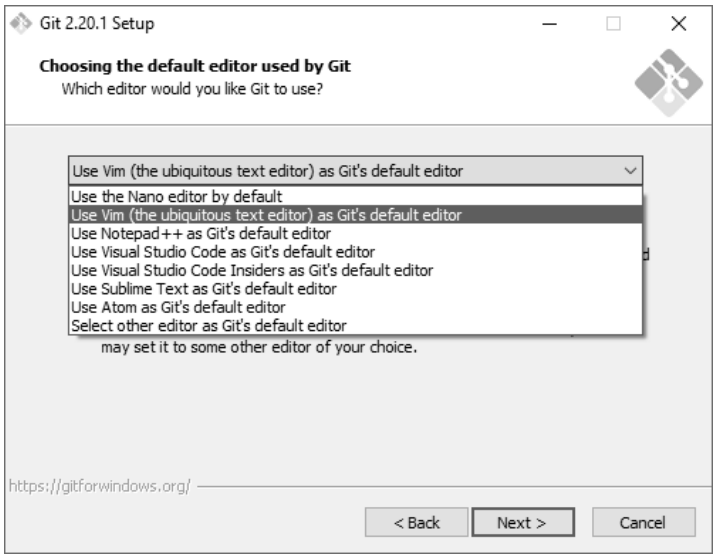


Abb. 2.1: Die Installation erlaubt die Auswahl verschiedener Editoren.

Mittlerweile können verschiedene Editoren ausgewählt werden. Eine Auswahl davon ist vorgegeben, aber auch andere Editoren lassen sich bereits bei der Installation konfigurieren.

Ein weiterer Punkt sind die Unterschiede bei der Nutzung der Shell. Die Shell ist das Fenster, in dem die Kommandozeilenbefehle eingetippt werden. Git lässt sich sowohl in der Windows-Cmd nutzen als auch in der »Git-Bash« verwenden.

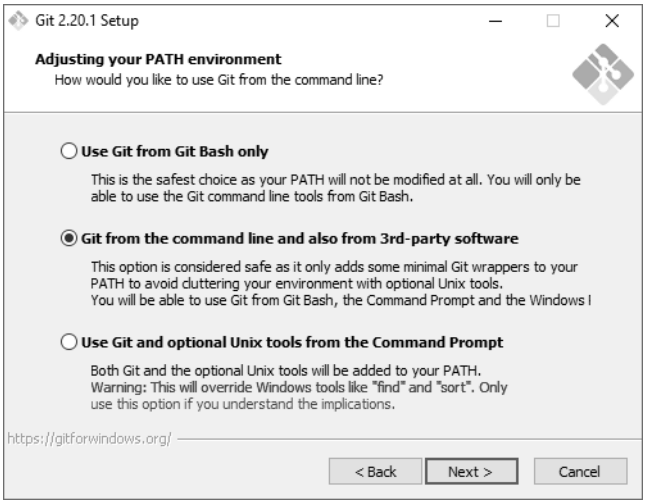


Abb. 2.2: Im Standard ist Git sowohl in der Bash als auch in der Windows-Cmd nutzbar.

Wenn Git zusätzlich in der Windows-Cmd genutzt werden soll, muss es in der PATH-Variablen eingetragen werden. Dies ist der Standard, wenn Git installiert wird.

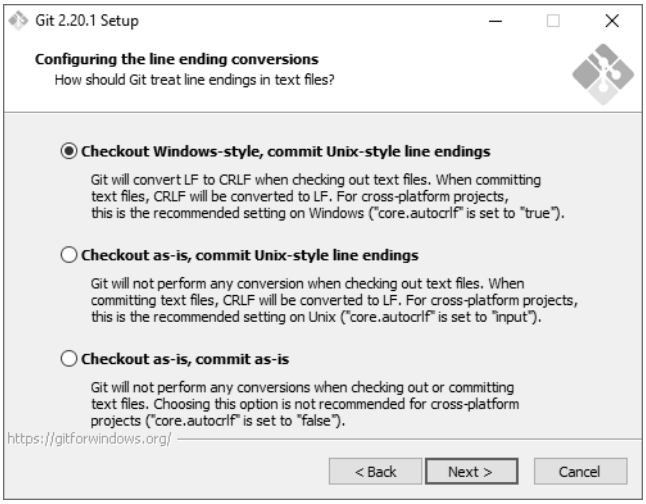


Abb. 2.3: Konfiguration des Verhaltens bezüglich des Zeilenendes

Eine weitere Konfiguration, die während der Installation abgefragt wird, ist die Einstellung bezüglich des Verhaltens des Zeilenendes. Der Standard ist, dass beim Auschecken von Dateien das Zeilenende von LF zu CRLF umgewandelt wird und

beim Committen in das Repository wieder in LF. Sofern Sie nicht wissen, was dann genau passiert, sollten Sie die anderen Optionen ausdrücklich nicht auswählen.

Bei der Nutzung von Git präferiere ich die Git-Bash, da sie im Defaultzustand einige zusätzliche Funktionen bietet, wie die Anzeige des Namens des aktuellen Branches. Außerdem können die gängigen Unix-Kommandos verwendet werden. Alle Befehle in diesem Buch lassen sich problemlos in einer Shell unter Mac OS X und Linux bzw. der Git-Bash unter Windows ausführen. Die Windows-Cmd kann zwar auch verwendet werden, allerdings nenne ich Windows-Cmd-Kommandos nicht noch einmal explizit. Dies ist unter anderem dann relevant, wenn etwa Ordner angelegt oder Dateien verschoben werden.

Abb. 2.4: Git-Bash unter Windows

2.2 Das erste Repository

Da Git ein verteiltes Versionsverwaltungsprogramm ist, lassen sich alle Operationen an einem Repository vollständig lokal ausführen. Alle Git-Funktionen, die in diesem Kapitel erläutert werden, sind ohne Ausnahme lokale Befehle auf dem eigenen Rechner. Verbindungen zu externen Git-Servern werden also nicht aufgenommen.

Bevor die ersten Dateien in ein Repository geschoben werden können, müssen sie lokal angelegt werden. Da Git ein Kommandozeilenprogramm ist, müssen Sie jetzt unter einem Linux- oder Mac-Betriebssystem das Terminal öffnen. Windows-Nutzer rufen an dieser Stelle die Git-Bash auf.

Im Defaultzustand landet man dann im Home-Verzeichnis des Nutzerkontos. Dies ist etwa `/home/svij` unter Linux, `C:\Users\svij` unter Windows oder `/Users/svij` unter Mac OS X. In diesem oder in einem anderen beliebigen Verzeichnis müssen Sie nun einen Unterordner namens `meineWebseite` anlegen, in dem das Repository und dessen Daten liegen sollen. Anschließend wechseln Sie mit dem `cd`-Befehl in das Verzeichnis.

```
$ mkdir meineWebseite
$ cd meineWebseite
```

In diesem Verzeichnis soll nun das Git-Repository angelegt werden. Dazu reicht ein einfaches Ausführen des folgenden Befehls:

```
$ git init
Leeres Git-Repository in /home/sujee/Repositorys/meineWebseite/.git/ initialisiert
```

Die Ausgabe des Befehls verrät schon, was passiert ist. Git hat innerhalb des Projekt-Ordners ein `.git`-Verzeichnis angelegt, in dem das leere Git-Repository liegt. Es liegen zwar Dateien und Ordner im `.git`-Verzeichnis, doch ist das Repository prinzipiell leer, da noch keine Daten und keine Revisionen hinterlegt sind. Das Verzeichnis ist auf allen Betriebssystemen versteckt. Das Gedächtnis des Git-Repositorys liegt vollständig im `.git`-Unterverzeichnis. Falls man das Verzeichnis löscht, sind auch alle gespeicherten Informationen des Repositorys gelöscht. Zum jetzigen Zeitpunkt wäre das natürlich nicht so tragisch, da es noch leer ist.

An dieser Stelle lohnt sich schon ein kleiner Blick in dieses Verzeichnis:

```
$ ls -l .git
insgesamt 12
drwxr-xr-x 1 sujee sujee   0 30. Dez 20:10 branches
-rw-r--r-- 1 sujee sujee  92 30. Dez 20:10 config
-rw-r--r-- 1 sujee sujee  73 30. Dez 20:10 description
-rw-r--r-- 1 sujee sujee  23 30. Dez 20:10 HEAD
drwxr-xr-x 1 sujee sujee 414 30. Dez 20:10 hooks
drwxr-xr-x 1 sujee sujee  14 30. Dez 20:10 info
drwxr-xr-x 1 sujee sujee  16 30. Dez 20:10 objects
drwxr-xr-x 1 sujee sujee  18 30. Dez 20:10 refs
```

Wie man sieht, liegen in dem `.git`-Verzeichnis einige Ordner und Dateien. Was genau darin passiert, ist an dieser Stelle zunächst irrelevant. Händisch muss in diesem Verzeichnis in der Regel zunächst nichts unternommen werden, außer wenn Sie Hooks – für das Ausführen von Skripten bei diversen Aktionen – oder die Konfiguration anpassen möchten. Allerdings sollten Sie die Dateien nur anfassen, wenn Ihnen bekannt ist, zu welchen Auswirkungen es führt, denn das Repository kann sonst kaputt gemacht werden!

2.3 Git-Konfiguration

Da Sie bereits ein leeres Repository angelegt haben, können Sie nun ein Commit hinzufügen. Was genau ein Commit ist und wie es getätigt werden kann, wird später genau erläutert. Denn zunächst müssen Sie noch die Git-Installation konfigurieren.

Vorerst werden allerdings nur zwei Dinge konfiguriert: der eigene Entwicklername und die dazugehörige E-Mail-Adresse.

Mit den folgenden Befehlen können Sie den eigenen Namen und die eigene E-Mail-Adresse setzen:

```
$ git config --global user.name "Sujeevan Vijayakumaran"
$ git config --global user.email mail@svij.org
```

An dieser Stelle sollten Sie natürlich dann Ihren Namen und E-Mail-Adresse eintragen und nicht meine Daten.

Mit diesen beiden Befehlen wird die Datei `.gitconfig` im Home-Verzeichnis angelegt. Der Inhalt der Datei in `~/.gitconfig` sieht anschließend so aus:

```
$ cat ~/.gitconfig
[user]
        name = Sujeevan Vijayakumaran
        email = mail@svij.org
```

Mit dem Befehl `git config -l` lässt sich die Konfiguration ebenfalls über die Kommandozeile ansehen.

Beachten Sie, dass bei den oben genannten Befehlen die Git-Identität global für das Benutzerkonto des Betriebssystems gesetzt wird. Wenn Sie für einzelne Git-Repositorys spezifische Einstellungen setzen möchten, reicht es, den Aufruf-Parameter `--global` wegzulassen. Die Konfiguration wird dann in die Datei `.git/config` im Projektordner gespeichert. Dies ist häufig dann sinnvoll, wenn Sie verschiedene E-Mail-Adressen für verschiedene Projekte nutzen. Das trifft beispielsweise dann zu, wenn Sie für die Erwerbsarbeit eine E-Mail-Adresse verwenden und für private Repositorys eine andere. Die angegebenen Informationen zu einem Entwickler sind für alle Personen einsehbar, die mindestens Lese-Rechte im Repository besitzen, sofern der Entwickler mindestens einen Commit getätigt hat.

2.4 Der erste Commit

An dieser Stelle startet das »echte« Arbeiten mit dem Git-Repository. Wie bereits vorher erwähnt, sind sowohl das Arbeitsverzeichnis als auch das Repository leer. Sie müssen daher einige Dateien in das Arbeitsverzeichnis schieben.

Zuvor lohnt sich noch ein Ausführen des Git-Kommandos `git status`:

```
$ git status
Auf Branch master
Noch keine Commits
nichts zu committen (erstellen/kopieren Sie Dateien und benutzen Sie "git
add" zum Versionieren)
```

Dieser Befehl gibt immer sinnvolle und praktische Informationen aus, die für das Arbeitsverzeichnis-Repository zu der entsprechenden Zeit hilfreich sind. Zum jet-

zigen Zeitpunkt teilt es mit, dass man sich auf dem Branch `master` befindet, noch keine Commits vorhanden sind und es noch nichts zu committen gibt. Es handelt sich demnach um ein noch leeres Repository.

Jetzt ist es an der Zeit, die ersten Dateien hinzuzufügen und den ersten Commit zu tätigen. Da in diesem Beispielprojekt das HTML5-Framework Bootstrap verwendet wird, müssen Sie dieses zunächst herunterladen und entpacken. Dies kann entweder händisch geschehen oder Sie führen folgende Befehle aus. Falls `curl` nicht installiert ist, was bei einigen Linux-Distributionen der Fall sein kann, können Sie es nachinstallieren, das Programm `wget` nutzen oder die Datei über den angegebenen Link händisch über den Browser herunterladen und entpacken.

```
$ curl -o bootstrap.zip -L ↵
https://github.com/twbs/bootstrap/releases/download/v4.2.1/ ↵
bootstrap-4.2.1-dist.zip
$ unzip bootstrap.zip
$ mv bootstrap-4.2.1-dist/* .
$ rmdir bootstrap-4.2.1-dist
$ rm bootstrap.zip
```

Einige der aufgeführten Befehle geben Text auf der Standard-Ausgabe aus, die ich hier aus Gründen der Übersichtlichkeit weggelassen habe. Über die Befehle wurde der Download getätigt, die Zip-Datei entpackt und somit ihr Inhalt in das Projektverzeichnis geschoben. Anschließend liegen im Projektverzeichnis dann zwei Unterverzeichnisse: `css` und `js`.

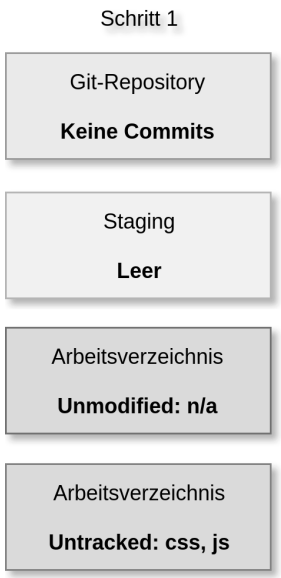


Abb. 2.5: Das Arbeitsverzeichnis ist gefüllt, Repository und Staging sind leer.

Obwohl die Dateien und Ordner im Projektordner liegen, sind die Dateien noch nicht im Repository. Sie müssen Git immer explizit mitteilen, dass Dateien in das Repository geschoben werden sollen.

Generell ist es durchaus häufig sinnvoll, den aktuellen Status im Arbeitsverzeichnis zu prüfen, deshalb lohnt sich jetzt ein Blick auf die Ausgabe von `git status`:

```
$ git status
Auf Branch master
Noch keine Commits
Unversionierte Dateien:
  (benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)
        css/
        js/
nichts zum Commit vorgemerkt, aber es gibt unversionierte Dateien (benutzen
Sie "git add" zum Versionieren)
```

Wie zu lesen ist, zeigt Git an, dass unbeobachtete Dateien bzw. Verzeichnisse vorhanden sind. Bei unbeobachteten Dateien handelt es sich um Dateien, die nicht von Git verwaltet werden und somit noch unbekannt sind. Eine Versionierung der Dateien findet noch nicht statt. Mit dem Befehl `git add` können Sie nun Dateien und Ordner zu dem sogenannten Staging-Bereich hinzufügen. Das ist der Bereich, in dem die Dateien hinzugefügt werden, um sie für einen Commit vorzumerken. Es ist dadurch eine Zwischenstufe zu einem Commit und wird häufig auch einfach kurz »Staging« oder »Index« genannt. Wenn im Git-Kontext von »Index« oder »Staging« die Rede ist, dann ist es genau dasselbe. Um die Dateien für einen Commit vorzumerken, müssen Sie den Ordner `css` zum Staging-Bereich hinzufügen:

```
$ git add css
```

Eine Ausgabe erfolgt an dieser Stelle nicht. Mit `git status` lässt sich erneut nachvollziehen, was geschehen ist, dies sieht dann wie folgt aus:

```
$ git status
Auf Branch master
Noch keine Commits
Zum Commit vorgemerkte Änderungen:
  (benutzen Sie "git rm --cached <Datei>..." zum Entfernen aus der Staging-Area)
        neue Datei:     css/bootstrap-grid.css
        neue Datei:     css/bootstrap-grid.css.map
        neue Datei:     css/bootstrap-grid.min.css
        neue Datei:     css/bootstrap-grid.min.css.map
        neue Datei:     css/bootstrap-reboot.css
```

```
        neue Datei:     css/bootstrap-reboot.css.map
        neue Datei:     css/bootstrap-reboot.min.css
        neue Datei:     css/bootstrap-reboot.min.css.map
        neue Datei:     css/bootstrap.css
        neue Datei:     css/bootstrap.css.map
        neue Datei:     css/bootstrap.min.css
        neue Datei:     css/bootstrap.min.css.map

Unversionierte Dateien:
  (benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)
        js/
```

Die Status-Ausgabe gibt Ordner und Dateien unterschiedlich aus. Der Ordner `js` wird mit abschließendem Schrägstrich / dargestellt. Dadurch ist auf den ersten Blick zu erkennen, dass es sich um ein Verzeichnis und nicht um eine normale Datei handelt. Beim Hinzufügen zum Staging-Bereich müssen diese Schrägstriche nicht mit angegeben werden.

Durch das Hinzufügen des Ordners `css` werden seine einzelnen Dateien für den nächsten Commit vorgemerkt, da sie sich nun im Staging-Bereich befinden. Die obige Abbildung visualisiert den Stand im Projekt mit drei Schichten: Arbeitsverzeichnis, Staging-Bereich und Git-Repository. Der Befehl `git add` hat also nicht viel mehr gemacht, als den Staging-Bereich zu füllen, indem es die zuvor noch nicht bekannten Daten dem Repository erstmals bekannt gemacht hat. An dieser Stelle ist allerdings immer noch kein Commit getätigt worden. Wenn Sie statt eines ganzen Ordners mit allen Dateien lieber nur einzelne Dateien hinzufügen möchten, geht das natürlich auch:

```
$ git add js/bootstrap.bundle.js
```

Auch hier erfolgt erneut keine Ausgabe, wenn die Ausführung erfolgreich war. An dieser Stelle bietet sich ein erneuter Blick auf den Status an:

```
$ git status
Auf Branch master
Noch keine Commits
Zum Commit vorgemerkte Änderungen:
  (benutzen Sie "git rm --cached <Datei>..." zum Entfernen aus der Staging-Area)
        neue Datei:     css/bootstrap-grid.css
        neue Datei:     css/bootstrap-grid.css.map
        neue Datei:     css/bootstrap-grid.min.css
        neue Datei:     css/bootstrap-grid.min.css.map
        neue Datei:     css/bootstrap-reboot.css
        neue Datei:     css/bootstrap-reboot.css.map
```

```
        neue Datei:     css/bootstrap-reboot.min.css
        neue Datei:     css/bootstrap-reboot.min.css.map
        neue Datei:     css/bootstrap.css
        neue Datei:     css/bootstrap.css.map
        neue Datei:     css/bootstrap.min.css
        neue Datei:     css/bootstrap.min.css.map
        neue Datei:     js/bootstrap.bundle.js
Unversionierte Dateien:
  (benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)
        js/bootstrap.bundle.js.map
        js/bootstrap.bundle.min.js
        js/bootstrap.bundle.min.js.map
        js/bootstrap.js
        js/bootstrap.js.map
        js/bootstrap.min.js
        js/bootstrap.min.js.map
```

Der vorherige Befehl hat das komplette `css`-Verzeichnis zum Staging-Bereich hinzugefügt. Mit dem Hinzufügen einer einzelnen Datei wird von `git status` nicht mehr der Ordner allgemein, sondern explizit alle Dateien aufgelistet.

Tipp

Wer eine kleinere und kürzere Status-Ausgabe haben möchte, kann den Parameter `-s` anhängen.

```
$ git status -s
A  css/bootstrap-grid.css
A  css/bootstrap-grid.css.map
A  css/bootstrap-grid.min.css
A  css/bootstrap-grid.min.css.map
A  css/bootstrap-reboot.css
A  css/bootstrap-reboot.css.map
A  css/bootstrap-reboot.min.css
A  css/bootstrap-reboot.min.css.map
A  css/bootstrap.css
A  css/bootstrap.css.map
A  css/bootstrap.min.css
A  css/bootstrap.min.css.map
A  js/bootstrap.bundle.js
?? js/bootstrap.bundle.js.map
?? js/bootstrap.bundle.min.js
?? js/bootstrap.bundle.min.js.map
?? js/bootstrap.js
?? js/bootstrap.js.map
?? js/bootstrap.min.js
?? js/bootstrap.min.js.map
```

Der Befehl mit diesem Parameter gibt eine kürzere Ausgabe aus und beschränkt sich auf die nötigsten Ausgaben. Vor den Dateien stehen Buchstaben, die den Status angeben. »A« steht für »Added«, also wenn sich die Dateien im Staging-Bereich befinden. »??« ist für Dateien, die bisher noch nicht beobachtet werden. Weiterhin existiert noch ein » M« und ein »M «, zu beachten ist das Leerzeichen jeweils vor bzw. hinter dem »M«. Bei Ersterem handelt es sich um eine veränderte Datei, die sich im Staging-Bereich befindet, bei Letzterem wiederum handelt es sich um eine Datei, die sich nicht im Staging-Bereich befindet.

Der nächste Befehl fügt alle restlichen Dateien, die noch nicht beobachtet werden, hinzu:

```
$ git add js
```

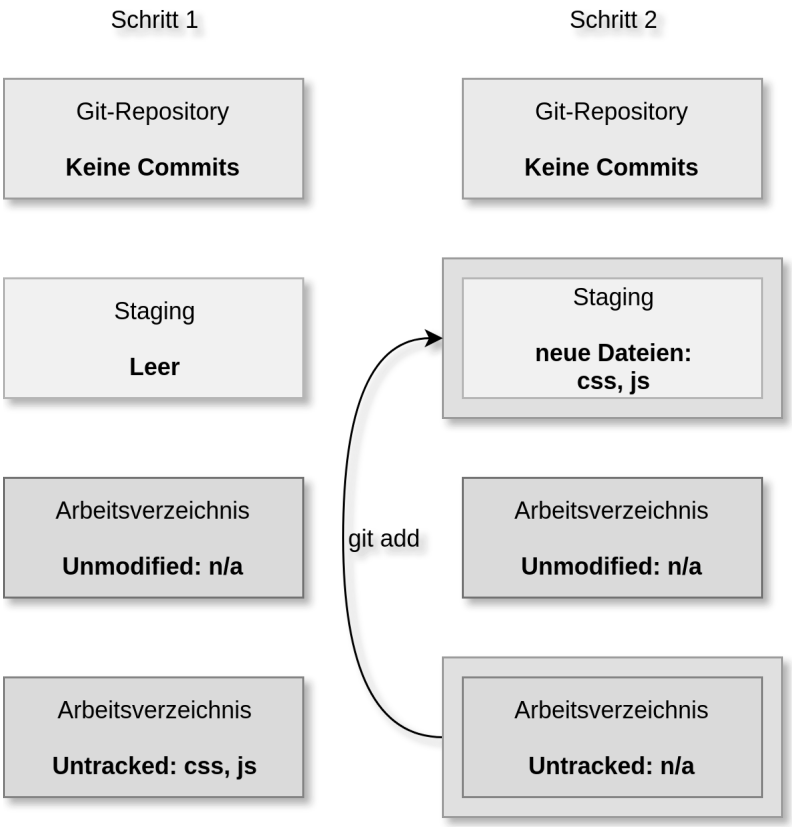


Abb. 2.6: Staging ist gefüllt, das Repository ist noch immer leer.

Wie an der Abbildung zu sehen ist, hat dieser Schritt den Staging-Bereich vollständig gefüllt. Es befinden sich alle Ordner des Arbeitsverzeichnisses im Staging-Bereich und sind bereit, in den ersten Commit und somit in das Repository geschoben zu werden.

Statt die Ordner einzeln hinzuzufügen, können Sie auch `git add -A` ausführen, um generell alle unbeobachteten und gegebenenfalls veränderten Dateien dem Staging-Bereich hinzuzufügen. Allerdings sollten Sie dabei stets aufpassen. Wenn Sie sichergestellt haben, dass keine anderen temporären Dateien vorhanden sind, können Sie den Befehl problemlos ausführen. Das ist in diesem Beispiel der Fall. Im Laufe von Entwicklungsarbeiten liegen aber häufig temporäre Dateien im Projektverzeichnis, die nicht in das Repository sollen.

Falls doch die ein oder andere Datei unbeabsichtigt hinzugefügt wurde und diese gar nicht mit in den nächsten Commit soll, dann kann man sie mit `git rm --cached $DATEINAME` einfach wieder herausnehmen. Der Parameter `--cached` ist hierbei wichtig, denn sonst wird die Datei gelöscht!

Nach einem erneuten Ausführen von `git status` werden alle dem Staging-Bereich hinzugefügten Dateien aus den zwei Unterordnern aufgelistet. Es bietet sich nicht nur für Anfänger an, jedes Mal zu überprüfen, ob die richtigen Dateien hinzugefügt wurden, bevor der Commit getätigt wird.

```
$ git status
Auf Branch master
Noch keine Commits
Zum Commit vorgemerkte Änderungen:
  (benutzen Sie "git rm --cached <Datei>..." zum Entfernen aus der Staging-Area)
        neue Datei:     css/bootstrap-grid.css
        neue Datei:     css/bootstrap-grid.css.map
        neue Datei:     css/bootstrap-grid.min.css
        neue Datei:     css/bootstrap-grid.min.css.map
        neue Datei:     css/bootstrap-reboot.css
        neue Datei:     css/bootstrap-reboot.css.map
        neue Datei:     css/bootstrap-reboot.min.css
        neue Datei:     css/bootstrap-reboot.min.css.map
        neue Datei:     css/bootstrap.css
        neue Datei:     css/bootstrap.css.map
        neue Datei:     css/bootstrap.min.css
        neue Datei:     css/bootstrap.min.css.map
        neue Datei:     js/bootstrap.bundle.js
```

```
        neue Datei:     js/bootstrap.bundle.js.map
        neue Datei:     js/bootstrap.bundle.min.js
        neue Datei:     js/bootstrap.bundle.min.js.map
        neue Datei:     js/bootstrap.js
        neue Datei:     js/bootstrap.js.map
        neue Datei:     js/bootstrap.min.js
        neue Datei:     js/bootstrap.min.js.map
```

Hiermit können Sie verifizieren, dass wirklich alle neuen Dateien hinzugefügt und keine vergessen wurden. Nach der Überprüfung können Sie den ersten Commit erzeugen. Er enthält dann genau die Dateien, die Sie zuvor mit `git add` zum Staging-Bereich hinzugefügt haben. Dateien, die gegebenenfalls ausgelassen wurden, bleiben unangetastet.

Mit dem folgenden Befehl wird der erste Commit erzeugt:

```
$ git commit -m "Füge Bootstrap Dateien hinzu"
[master (Basis-Commit) 8089134] Füge Bootstrap Dateien hinzu
 20 files changed, 25038 insertions(+)
 create mode 100644 css/bootstrap-grid.css
 create mode 100644 css/bootstrap-grid.css.map
 create mode 100644 css/bootstrap-grid.min.css
 create mode 100644 css/bootstrap-grid.min.css.map
 create mode 100644 css/bootstrap-reboot.css
 create mode 100644 css/bootstrap-reboot.css.map
 create mode 100644 css/bootstrap-reboot.min.css
 create mode 100644 css/bootstrap-reboot.min.css.map
 create mode 100644 css/bootstrap.css
 create mode 100644 css/bootstrap.css.map
 create mode 100644 css/bootstrap.min.css
 create mode 100644 css/bootstrap.min.css.map
 create mode 100644 js/bootstrap.bundle.js
 create mode 100644 js/bootstrap.bundle.js.map
 create mode 100644 js/bootstrap.bundle.min.js
 create mode 100644 js/bootstrap.bundle.min.js.map
 create mode 100644 js/bootstrap.js
 create mode 100644 js/bootstrap.js.map
 create mode 100644 js/bootstrap.min.js
 create mode 100644 js/bootstrap.min.js.map
```

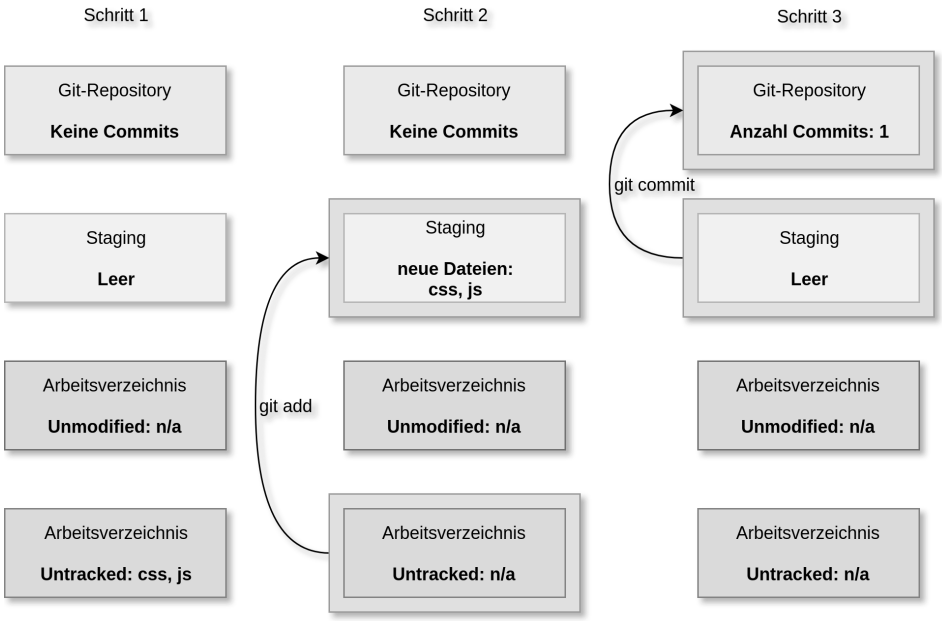


Abb. 2.7: Der erste Commit ist getätigt, Staging ist anschließend leer.

Der Befehl `git commit` speichert die Dateien aus dem aktuellen Staging-Bereich in eine Commit. Mit dem Parameter `-m` können Sie eine Commit-Nachricht direkt übergeben. Diese fasst in der Regel die aktuellen Änderungen zusammen, sodass sowohl Sie als auch andere Mitarbeiter mit Zugang zu dem Repository die Änderungen in dem Commit möglichst schnell und einfach nachvollziehen können. Wichtig ist vor allem, dass beschrieben wird, warum etwas getätigt wurde und nicht was. Was geändert wurde, ist im Commit selbst ersichtlich. In diesem Beispielprojekt ist das hingegen noch nicht sehr relevant.

Wie in der Abbildung verdeutlicht wird, ist der Staging-Bereich nach dem Commit leer. Der Prozess beim Committen sieht also prinzipiell so aus, dass die Änderungen – dazu zählen auch neue Dateien – zwischen die »Schichten« geschoben werden. Wenn Sie Dateien mit `git add` hinzufügen und mit `git commit` den Commit erzeugen, gehen die Dateien jeweils eine Schicht nach oben. Mit `git reset` können Sie die Änderungen auf verschiedene Arten wieder auf die unteren Schichten holen. Dazu folgen allerdings an späterer Stelle mehr Informationen. Zum Schluss sind vier statt drei Schichten dargestellt. Neben »Untracked« gibt es noch »Unmodified«. In »Untracked« liegen die unbeobachteten Dateien und in »Unmodified« nach dem ersten Commit die Dateien, die verändert werden könnten.

Hinweis

Wenn allerdings der Commit-Befehl ohne den Parameter `-m` ausführt wird, öffnet sich stattdessen der Standard-Editor, in dem dann die Commit-Nachricht eingetippt werden kann. Dies ist häufig der Konsoleneditor »vim«. Mit dem folgenden Befehl kann man den Editor auf den simplen Konsoleneditor »nano« ändern:

```
$ git config --global core.editor nano
```

Bei Windows-Nutzern wird bei der Installation von Git die Konfiguration des Standard-Editors abgefragt. Wer dies verpasst hat, kann auch unter Windows »nano« nutzen. Alternativ lässt sich auch der persönlich präferierte Editor nutzen, wozu man statt nano den Namen oder direkten Pfad zum Editor im obigen Befehl angeben muss. Für Notepad++ sähe das dann so aus, wenn es sich um eine 32-Bit-Windows-Installation handelt:

```
$ git config --global core.editor "'C:/Program Files (x86)/Notepad++/
notepad++.exe' -multiInst -nosession"
```

Bei einem 64-Bit-System ist der Pfad zur Exe-Datei ein wenig anders, weshalb der Befehl dementsprechend angepasst werden muss:

```
$ git config --global core.editor "'C:/Program Files/Notepad++/
notepad++.exe' -multiInst -nosession"
```

Nach dem ersten Commit können Sie noch einmal den `git status`-Befehl ausführen:

```
$ git status
Auf Branch master
nichts zu committen, Arbeitsverzeichnis unverändert
```

Der erste Commit ist also getätigt. Dieser wird häufig »initialer Commit« oder »Basis-Commit« genannt, weil er der erste Commit des Repositorys ist, auf dem die anderen Commits aufbauen. Kleiner Hinweis am Rande: Der erste Commit lässt sich im Nachhinein nicht mehr ändern. Das ist bei insgesamt einem Commit im Repository nicht so hinderlich. Alle anderen Commits lassen sich auch nach dem Committen anpassen, dazu aber später mehr. Alternativ lässt sich als initialer Commit auch ein komplett leerer Commit erzeugen, der keine Dateien oder Änderungen enthält. Dazu muss man nach der Initialisierung `git commit --allow-empty` ausführen.

Mit `git log` können Sie sich die Historie des Repositorys ansehen. Bei lediglich einem Commit ist sie in diesem Fall natürlich sehr kurz.

```
$ git log
commit 808913470b714257dcc836e23dcad0bc532e249e (HEAD -> master)
Author: Sujeevan Vijayakumaran <mail@svij.org>
Date:   Sun Dec 30 20:49:50 2018 +0100

    Füge Bootstrap Dateien hinzu
```

Abb. 2.8: Der erste Commit auf dem `master`-Branch

Jeder Commit besitzt eine eindeutige ID, über die er referenziert werden kann. Sie kann unter anderem genutzt werden, um etwa das Log zwischen zwei verschiedenen Revisionen darzustellen, explizit zu dem Stand des Commits zu wechseln oder die Änderungen des Commits rückgängig zu machen. Die ID ist genau genommen die SHA-1-Checksumme, die aus den Änderungen berechnet wird. Zusätzlich werden in einem Commit noch das Datum, der Autor, der Committer und eben die Commit-Nachricht vermerkt.

Der erste Commit ist an dieser Stelle also getätigt. Wirkliche Änderungen wurden im Projekt bislang noch nicht vorgenommen. Um das Beispiel-Projekt mit der Webseite fortzuführen, müssen Sie nun die `index.html` mit dem folgenden Inhalt anlegen:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no">
    <link rel="stylesheet" href="css/bootstrap.min.css"
crossorigin="anonymous">
    <title>Hello, world!</title>
  </head>
  <body>
    <h1>Hello, world!</h1>
    <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js"
integrity="sha384-KJ3o2DKtIkvYIK3UENzmM7KCkRr/rE9/Qpg6aAZGJwFDMVNA/
GpGFF93hXpG5KkN" crossorigin="anonymous"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/
umd/popper.min.js" integrity="sha384-ApNbgh9B+Y1QKtv3Rn7W3mgPxhU9K/
ScQsAP7hUibX39j7fakFPskvXusvfa0b4Q" crossorigin="anonymous"></script>
    <script src="js/bootstrap.min.js" crossorigin="anonymous"></script>
```

```
  </body>
</html>
```

Den Inhalt müssen Sie nicht abtippen. Eine leicht angepasste Version findet sich auch unter `https://getbootstrap.com/docs/4.0/getting-started/introduction/`.

Diese Datei können Sie mit einem beliebigen Browser öffnen. Sowohl der Seitentitel, der in der Tab-Leiste dargestellt wird, als auch der Inhalt der eigentlichen Webseite zeigen den Text »Hello, world!« an. HTML ist in diesem Buch natürlich nicht das Thema, allerdings ist es hilfreich, grob zu verstehen, wie der oben aufgeführte HTML-Code funktioniert. Er unterteilt sich in den Head- und den Body-Bereich, die durch die entsprechenden Tags in den spitzen Klammern gekennzeichnet sind. So wird darin unter anderem der Titel der Webseite spezifiziert und das Stylesheet geladen, das später Design-Elemente wie Buttons bereitstellt. Im Body wird eine Überschrift der ersten Ebene definiert, ebenfalls mit dem Text »Hello, world!«. Daneben wird noch JavaScript-Code geladen, der für dieses Buch nicht weiter interessant ist. Wie schon im vorherigen Kapitel erwähnt, ist es das Ziel, anhand dieser kleinen Webseite die verschiedenen Funktionen von Git exemplarisch an einem Projekt vorzustellen.

Die `index.html`-Datei ist zwar lokal auf dem Rechner verfügbar, sie befindet sich allerdings noch nicht im Repository. Beim ersten Commit wurden zwar alle Dateien aus dem Verzeichnis in das Repository geschoben, allerdings existierte die `index.html`-Datei zu dem Zeitpunkt noch nicht. Diese könen Sie, wie zuvor erklärt, mit einem einfachen `git add` in den Staging-Bereich überführen und mit `git commit` zum Repository hinzufügen.

```
$ git add index.html
$ git commit -m "Füge index.html hinzu"
[master 50d3b52] Füge index.html hinzu
 1 file changed, 15 insertions(+)
 create mode 100644 index.html
```

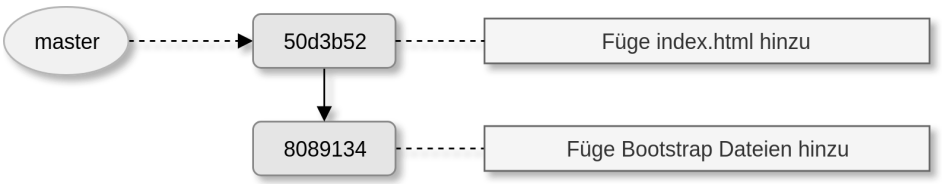


Abb. 2.9: Der zweite Commit auf dem Branch `master`

Somit wurde nun ein zweiter Commit erzeugt, der nur die neuen Änderungen beinhaltet. Dieser umfasst die neue Datei `index.html`, die 16 Zeilen enthält. Die

Ausgabe von `git commit` zeigt dankenswerterweise ein paar nützliche Informationen an. So wurde eine Datei verändert und in der darauf folgenden Zeile wird auch hervorgehoben, dass es sich um eine neue Datei handelt.

Weiterhin wird auch der Modus mit der Ziffernfolge »100644« ausgegeben. Diese gibt sowohl den Typ der Datei als auch die Berechtigungen an. Für Sie als Endanwender ist dieser Teil der Ausgabe wohl in der Regel irrelevant. Sinnvoll hingegen ist, dass Sie direkt an dieser Stelle schon erkennen können, ob nicht vielleicht zu viele oder zu wenige Dateien zum Commit hinzugefügt wurden.

Zurück zum Projekt: Das Grundgerüst steht an dieser Stelle. Die Startseite `index.html` können Sie nun nach Belieben anpassen. Um die Seite inhaltlich mehr in den Git-Kontext zu rücken, bietet es sich an, den Titel der Webseite zu verändern und statt der Welt einfach mal Git »Hallo« zu sagen. Der Titel steht in der siebten Zeile der `index.html`-Datei. Sie können ihn einfach durch folgende Zeile ersetzen:

```
<title>Meine Webseite mit Git!</title>
```

Anschließend müssen Sie noch die Überschrift ersetzen. Da Git gegrüßt werden will, kann die entsprechende Zeile wie folgt lauten:

```
<h1>Hallo Git!</h1>
```

Sie können die Datei mit einem beliebigen Editor bearbeiten. Nachdem beide Änderungen durchgeführt wurden, können Sie sich die Änderungen mittels des Kommandos `git diff` ansehen. Davor lohnt sich wieder ein Blick auf die Ausgabe von `git status`.

```
$ git status
Auf Branch master
Änderungen, die nicht zum Commit vorgemerkt sind:
  (benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)
  (benutzen Sie "git checkout -- <Datei>...", um die Änderungen im
Arbeitsverzeichnis zu verwerfen)
      geändert:       index.html
keine Änderungen zum Commit vorgemerkt (benutzen Sie "git add" und/oder "git
commit -a")
```

Beim Ausführen des Kommandos bemerkt Git, dass sich der Inhalt von `index.html` verändert hat. Die Status-Ausgabe gibt zusätzlich an, mit welchem Befehl Sie die Änderung zu einem Commit vormerken und wie sie wieder rückgängig gemacht werden kann. Zunächst bietet es sich an, die Änderungen anzuschauen. Diese sind zwar in diesem Beispiel trivial, dies ändert sich in echten Projekten allerdings sehr schnell.

```
$ git diff
diff --git a/index.html b/index.html
index ccb5aca..dcd955c 100644
--- a/index.html
+++ b/index.html
@@ -4,10 +4,10 @@
     <meta charset="utf-8">
     <meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no">
     <link rel="stylesheet" href="css/bootstrap.min.css"
crossorigin="anonymous">
-    <title>Hello, world!</title>
+    <title>Meine Webseite mit Git!</title>
   </head>
   <body>
-    <h1>Hello, world!</h1>
+    <h1>Hallo Git!</h1>
     <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js"
integrity="sha384-KJ3o2DKtIkvYIK3UENzmM7KCkRr/rE9/Qpg6aAZGJwFDMVNA/
GpGFF93hXpG5KkN" crossorigin="anonymous"></script>
     <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/
umd/popper.min.js" integrity="sha384-ApNbgh9B+Y1QKtv3Rn7W3mgPxhU9K/
ScQsAP7hUibX39j7fakFPskvXusvfa0b4Q" crossorigin="anonymous"></script>
     <script src="js/bootstrap.min.js" crossorigin="anonymous"></script>
```

Die Ausgabe von `git diff` zeigt letztendlich die Änderungen an der Datei an. Dabei wird immer ein Ausschnitt einer Datei dargestellt und die veränderten Zeilen mit einem »+« und »-« hervorgehoben. Wenig überraschend steht das Minus am Beginn einer Zeile für eine entfernte Zeile und das Plus für eine neue Zeile. Dies gilt sowohl für neue und entfernte Zeilen als auch für Veränderungen an einer Zeile.

Tipp

Zur besseren Übersicht über die Ausgaben von Git im Terminal bietet es sich an, die Farbausgabe zu aktivieren. Diese kann man global in der Konfiguration mit folgendem Befehl setzen:

```
$ git config --global color.ui true
```

Zusätzlich zu dem Plus- und Minus-Zeichen bei den Änderungen werden gelöschte Zeilen mit roter Schriftfarbe hervorgehoben und für hinzugefügte Zeilen ist es die grüne Farbe. Das erleichtert das Ansehen von Diffs ungemein. Bei den meisten Installationen dürfte diese Konfiguration bereits im Standard aktiviert sein.

Falls alle Änderungen korrekt sind, können Sie die geänderten Dateien wie gehabt zum Staging-Bereich hinzufügen und anschließend den Commit erzeugen. Dies ist dann der dritte Commit.

```
$ git status
$ git add index.html
$ git commit -m "Ändere Überschrift und Titel"
[master d9ce51e] Ändere Überschrift und Titel
 1 file changed, 2 insertions(+), 2 deletions(-)
```

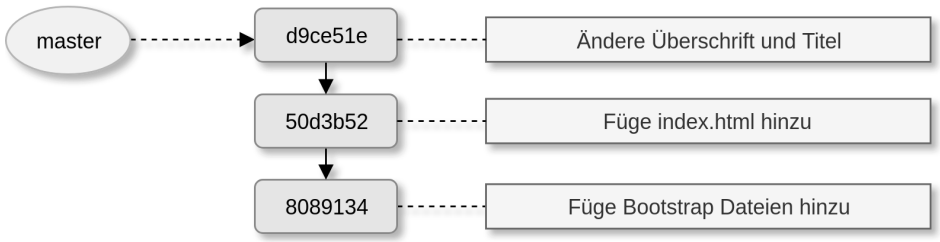


Abb. 2.10: Der dritte Commit auf dem `master`-Branch

Etwas konfus werden die Ausgaben von `git diff`, wenn sich etwa einzelne Änderungen im Staging-Bereich befinden, die allerdings noch nicht in einem Commit sind. Wenn Sie nach dem letzten Commit keine Änderungen vorgenommen haben, gibt es keine Ausgabe beim Ausführen von `git diff`. Nun sollten Sie eine zweite Überschrift unterhalb der ersten Überschrift setzen, um die verschiedenen Modi von `git diff` hervorzuheben. Fügen Sie dazu folgende Zeile ein:

```
<h2>Endlich lernen wir uns kennen.</h2>
```

Mit `git status` kann man wie gehabt nachvollziehen, dass `index.html` verändert wurde. Ein Blick auf `git diff` zeigt die hinzugefügte Zeile in der Datei an, so weit klar. Anders ist es allerdings, wenn die Datei zum Staging-Bereich hinzugefügt wird und man sich anschließend das Diff anschaut.

```
$ git add index.html
$ git diff
```

Hinweis

Es ist auch möglich, Änderungen häppchenweise zu stagen und zu committen. Dies geht mit dem Befehl `git add -p`. Er ist hilfreich, wenn nur Teile einer Änderung in einen Commit fließen sollen. Eine nähere Erläuterung dazu findet sich in Abschnitt 9.8.

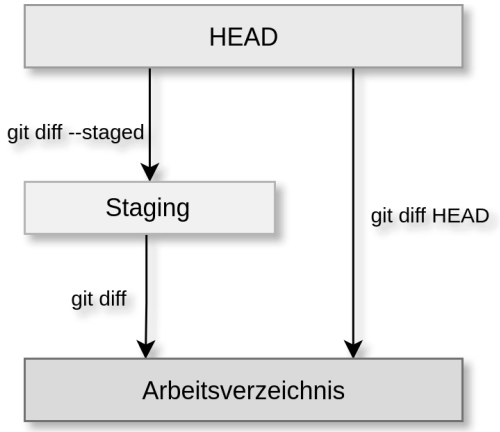


Abb. 2.11: Die verschiedenen Modi von `git diff` zeigen die Unterschiede der unterschiedlichen Schichten.

Wie Sie sehen, wird jetzt keine Änderung mehr angezeigt, obwohl Änderungen in der aktuellen Datei im Vergleich zum letzten Commit vorhanden sind. Das liegt daran, dass `git diff` immer nur die Änderungen zum letzten Commit anzeigt, die noch nicht im Staging-Bereich sind. Es zeigt also nur die Änderungen zwischen dem Arbeitsverzeichnis und dem Staging an, wie in der Grafik auch zu erkennen ist. Das hat den Vorteil, dass man sich bei größeren Änderungen immer wieder das Diff anschauen und nach und nach die Dateien zum Staging-Bereich hinzufügen kann und dabei eine gute Übersicht hat, welche Änderungen noch nicht im Staging-Bereich sind. Es gibt zusätzlich die Möglichkeit, die Unterschiede zwischen dem Staging-Bereich und HEAD anzuschauen. Der HEAD ist der aktuell ausgecheckte Commit. Dies entspricht in vielen Fällen dem aktuellsten Commit des ausgecheckten Branches. Um sich nun den Unterschied zwischen Staging-Bereich und HEAD anzuschauen, können Sie sowohl `git diff --staged` als auch `git diff --cached` verwenden. Unterschiede zwischen den beiden Befehlen gibt es keine, sodass sie synonym verwendet werden können. Auch eine Kombination aus `git diff` und `git diff --staged` ist möglich. `git diff HEAD` gibt alle Änderungen aus dem Arbeitsverzeichnis im Vergleich zum ausgecheckten Commit aus. Das sind dann auch alle Änderungen, die in einen Commit einfließen würden, wenn man `git commit -a` ausführt. Letzterer Befehl kombiniert `git add` und `git commit`, sodass alle geänderten Dateien direkt und ohne Umwege über den Staging-Bereich in einem neuen Commit gespeichert werden.

Aus diesen Änderungen soll allerdings kein Commit erzeugt werden. Weiter vorne habe ich schon kurz ohne Praxis-Beispiel beschrieben, wie die Änderung aus dem Staging-Bereich wieder ausgetragen werden kann. Der Befehl `git status` macht es dem Nutzer zum Glück einfach und schlägt direkt die nächsten Schritte vor. Mit

`git reset HEAD index.html` kann man die Änderung wieder aus dem Staging-Bereich nehmen.

```
$ git reset HEAD index.html
Nicht zum Commit vorgemerkte Änderungen nach Zurücksetzung:
M       index.html
```

Die Änderung ist nicht verloren gegangen, sodass sie wieder in den Staging-Bereich und anschließend in einen Commit fließen kann, wenn dies gewünscht ist. An dieser Stelle soll die Änderung allerdings wieder vollständig zurückgenommen werden. Die naive Herangehensweise wäre, die Datei händisch zu öffnen und die hinzugefügte Zeile zu löschen. Bei kleinen Änderungen ist das zwar möglich, einfacher geht es allerdings, wenn Sie sich die Ausgabe von `git status` erneut genauer anschauen. Auch hier gibt Git mal wieder eine nützliche Hilfestellung. Die Änderung kann nämlich mit dem folgenden Befehl wieder entfernt werden:

```
$ git checkout -- index.html
```

Der Checkout-Befehl wird im dritten Kapitel genauer behandelt, wenn es um das Arbeiten mit Branches geht. Wichtig sind allerdings die beiden Minus-Zeichen, nach denen dann der Pfad zu einer oder mehreren Dateien folgen kann. Eine Ausgabe erfolgt im Erfolgsfall nicht.

Interessant wird es, wenn mehrere Änderungen an einer Datei gemacht wurden, die man in einzelne Commits verpacken möchte. Die Größe von Commits sollte immer nur so klein wie möglich und so groß wie nötig sein. Konkret heißt das, dass ein Commit immer eine logische Änderung enthalten soll. Wenn Sie in Projekten wie dieser statischen Webseite arbeiten, dann sollten Sie zum Beispiel Inhalte auf Unterseiten nur dann in einem Commit zusammenfassen, wenn sie strikt zusammengehören.

Wird beispielsweise eine »Über mich«-Seite angelegt, dann sollte das eher nicht zusammen mit einer Seite über Naturfotografie in einem Commit enthalten sein. Das ist vor allem beim Arbeiten im Team hilfreich, denn jeder im Team kann dann anhand des Logs nachvollziehen, welche Änderungen von welcher Person aus welchem Grund bzw. mit welcher Notiz durchgeführt wurden.

Die Commit-Nachricht sollte daher auch immer eine sinnvolle Beschreibung der durchgeführten Änderung sein. Commit-Nachrichten à la »Aktueller Stand«, »Korrektur« oder »Fertig!« sollten Sie zum eigenen Wohl und zum Wohle des Teams vermeiden, da daraus überhaupt nicht ersichtlich wird, was sich hinter der Beschreibung verbirgt. In welchem Level die Verständlichkeit der Commit-Nachrichten forciert werden soll, hängt immer vom Team ab. Dazu folgt allerdings in Kapitel 6 des Buches noch eine genauere Betrachtung, wenn es um die Workflows geht.

2.4.1 Versionierte Dateien mit »git mv« verschieben

Dateien, die bereits versioniert sind, können Sie mit dem `git mv`-Kommando innerhalb des Projektordners verschieben. Es ist eine kürzere Form, als die Dateien händisch zu verschieben und einzeln zum Staging-Bereich zu schieben.

```
$ git mv alterDateiname neuerDateiname
```

Der obige Befehl tut prinzipiell dasselbe wie die folgenden Befehle:

```
$ mv alterDateiname neuerDateiname
$ git add neuerDateiname
$ git rm alterDateiname
```

Wie Sie sehen, gibt es auch `git rm`, womit Dateien aus dem Repository entfernt werden können. Git erkennt zwar eine Umbenennung der Datei, doch man hat bei einigen Gelegenheiten keine Chance, den Dateinamen herauszufinden, wenn etwa Änderungen über Patch-Dateien hinzugefügt werden. Dies wirkt sich auf die Historie einer einzelnen Datei aus. Diese zeigt defaultmäßig die Umbenennung nämlich nicht an. Die ältere Historie lässt sich nur über den `--follow`-Parameter abrufen. Konkret sähe es dann so aus:

```
$ git log --follow neuerDateiname
```

2.5 Änderungen rückgängig machen mit Reset und Revert

In den vorherigen Abschnitten ging es bereits kurz um das Rückgängigmachen von Änderungen. Änderungen können auf verschiedene Arten rückgängig gemacht werden. Diese hängen jeweils davon ab, in welchem Zustand sich die Änderung befindet. Es macht dadurch einen Unterschied, ob die Änderung in einem Commit steckt oder nur im Staging-Bereich ist. An dieser Stelle gehe ich zunächst nur auf die Grundlagen ein, da Branches und ihr Einsatz bis zu dieser Stelle noch nicht behandelt wurden.

2.5.1 Revert

Mit `git revert` können Sie einzelne Commits rückgängig machen. Dies betrifft daher nur Änderungen, die in Commits gespeichert werden. Am einfachsten lässt sich das Verhalten nachvollziehen, wenn Sie etwa am Ende der Datei `index.html` eine beliebige Zeile hinzufügen und daraus den Commit erzeugen, der im Anschluss zurückgenommen werden soll.

```
$ echo "Die Zeile für den Revert" >> index.html
$ git commit -am "Zeile hinzugefügt, um diese zu reverten"
[master c8184a3] Zeile hinzugefügt, um diese zu reverten
 1 file changed, 1 insertion(+)
```

Die Änderung ist an dieser Stelle nun im Repository. Mit `git revert` und der Commit-ID lässt sich die Änderung wieder zurücknehmen.

```
$ git revert c8184a3
```

Im Anschluss öffnet sich der Editor mit der folgenden Nachricht:

```
Revert "Zeile hinzugefügt, um diese zu reverten"
This reverts commit c8184a3831ba4b913a2fc406389c13cf30ea456e.
```

Der `revert`-Befehl erwartet also eine Commit-Nachricht, die nicht verändert werden sollte, da Revert-Commits, die nicht als solche ersichtlich sind, stark verwirren können, wenn sich jemand die Historie anschaut. Mit `revert` wird also ein zusätzlicher Commit erzeugt, der den angegebenen Commit rückgängig macht. Die Historie sieht dann so aus:

```
$ git log --pretty=oneline
5292ec9a8b27e80e10291b6e Revert "Zeile hinzugefügt, um diese zu reverten"
c8184a3831ba4b913a2fc4063 Zeile hinzugefügt, um diese zu reverten
```

Dies ist somit keine Option, die Sie nutzen sollten, wenn Änderungen komplett verschwinden sollen, da sie dadurch weiterhin in der Historie bleiben. Die Revert-Option bietet sich immer dann an, wenn der Branch mit anderen Personen geteilt wird und somit schon veröffentlicht wurde. Falls dies nicht der Fall ist und Sie möchten, dass die Überreste aus der Historie verschwinden, bietet sich ein Entfernen über das interaktive Rebasen an. Dies wird in Kapitel 6 thematisiert. Alternativ ist es auch möglich, `reset` zu nutzen.

2.5.2 Reset

Im vorherigen Schritt wurde zum Darlegen der Revert-Funktion ein Commit angelegt und wieder rückgängig gemacht, weshalb jetzt unnötige Commits in der Historie liegen. Diese können entweder über ein interaktives Rebasen entfernt werden oder man nutzt `reset`.

Es gibt verschiedene Modi bei der Nutzung von `reset`. Bereits in diesem Kapitel ging es darum, wie Änderungen wieder aus dem Staging-Bereich entfernt werden können. Dazu reicht die Ausführung von `git reset HEAD index.html`, wenn es sich um die `index.html`-Datei handelt. Die Änderungen selbst sind weiterhin

darin enthalten und können mit `git checkout -- index.html` rückgängig gemacht werden. So viel zur Wiederholung.

Reset besitzt drei verschiedene Modi: »soft«, »mixed« und »hard«. Jeder Modus hat eine andere Verhaltensweise, die Sie sich einprägen sollten, um das nicht jedes Mal nachgeschlagen zu müssen.

Ziel im nächsten Schritt ist das Rückgängigmachen der letzten beiden Commits, die beim Revert angelegt wurden.

```
$ git reset HEAD~2
$ git log --pretty=oneline -n 2
d9ce51e Ändere Überschrift und Titel
50d3b52 Füge index.html hinzu
```

Das war die einfachste Variante, die beiden Commits rückgängig zu machen. Mit `HEAD~2` wird angegeben, dass die letzten beiden Commits von `HEAD` rückgängig gemacht werden sollen. Im Arbeitsverzeichnis ist kein Unterschied zu erkennen, da die beiden Beispiel-Commits sich ja sowieso durch den vorherigen Revert gegenseitig aufgehoben haben. Anzumerken ist weiterhin, dass bei keiner Angabe eines Parameter immer defaultmäßig `--mixed` ausgeführt wird. Um das Verhalten besser nachvollziehen zu können, bietet sich ein weiteres Beispiel an, wozu Sie erneut eine Zeile in `index.html` hinzufügen.

```
$ echo "Die Zeile für den Reset" >> index.html
$ git commit -am "Füge Zeile für den Reset hinzu"
[master 556a3a5] "Füge Zeile für den Reset hinzu"
 1 file changed, 1 insertion(+)
```

Anschließend können Sie erneut `reset` ausführen, diesmal mit einem, statt zwei Commits.

```
$ git reset HEAD~1
$ git status -s
M index.html
```

Der Commit wurde rückgängig gemacht, die Änderung ist allerdings immer noch enthalten, so findet sich die zusätzliche Zeile weiterhin in der Datei. Anders sieht es aus, wenn der Commit zurückgenommen werden soll, die Änderungen aber im Staging-Bereich verbleiben sollen. Dazu können Sie wieder die Änderung zum Staging-Bereich hinzufügen und den Commit erzeugen.

```
$ git add index.html
$ git commit -m "Füge Zeile für den Reset hinzu"
```

```
$ git reset --soft HEAD~1
$ git status
[...]
zum Commit vorgemerkte Änderungen:
  (benutzen Sie "git reset HEAD <Datei>..." zum Entfernen aus der Staging-
Area)
          geändert:       index.html
```

Mit dem Parameter `--soft` ist es somit möglich, den letzten Schritt vor dem Commit rückgängig zu machen. Dies bietet sich immer dann an, wenn Sie feststellen, dass noch einige Änderungen in den letzten Commit hätten fließen sollen.

Die letzten beiden Varianten waren die »weichen« Varianten, da kein Verlust der Änderung droht. Anders sieht es aus, wenn ein harter Reset ausgeführt wird. Dazu müssen Sie wieder ein Commit erzeugen, der dann mit `reset --hard` rückgängig gemacht wird.

```
$ git commit -m "Füge Zeile für den Reset hinzu"
$ git reset --hard HEAD~1
```

In diesem Fall verschwindet nicht nur der Commit vollständig, sondern es ist auch die Änderung an sich verschwunden. Der Befehl sollte also nur dann ausgeführt werden, wenn die Änderungen wirklich nicht gebraucht werden.

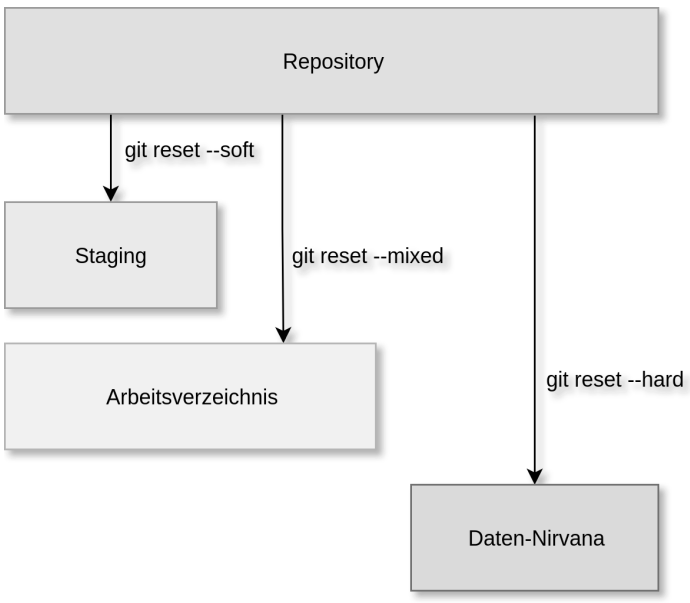


Abb. 2.12: Die Unterschiede zwischen den verschiedenen Modi

Deshalb gilt: Ein Soft-Reset wird immer dann durchgeführt, wenn der Commit rückgängig gemacht werden, die Änderung aber nicht verloren gehen soll. Ein Hard-Reset wird immer dann ausgeführt, wenn sowohl die Änderung als auch der Commit entfernt werden kann. Die Daten landen quasi im Daten-Nirvana und lassen sich nicht wiederherstellen. Das Hard-Reset wird häufiger im Zusammenhang mit dem Arbeiten mit Branches und Remote-Repositorys gebraucht, die in Kapitel 3 und 4 nochmals erläutert werden.

Wichtig

Nutzen Sie niemals `git reset` mit `--hard` und `--merge`, wenn veränderte Dateien im Arbeitsverzeichnis liegen, die nicht in einem Commit oder im Stash sind. Sonst gibt es Datenverlust!

Der Reset-Befehl kann mit verschiedenen Parametern aufgerufen werden. In den Beispielen wurde immer nur über `HEAD` auf die Commits referenziert. Dies ist bei kleinen Schritten problemlos möglich, aber die Commits können auch anders referenziert werden:

- `bf55e27b8f486c1e41754f6c3bebbd0e74c5a5fa` – vollständige Commit-ID
- `bf55e27` – kurze Commit-ID
- `HEAD~2` – vorletzter Commit von `HEAD` ausgehend
- `master~2` – vorletzter Commit auf `master`
- `master^^^` – drittletzter Commit auf `master`
- `master@{4}` – viertletzter Commit vom aktuellen `HEAD` von `master`

2.6 Git mit GUI

Das Buch fokussiert sich größtenteils auf die Kommandozeilen-Befehle von Git, nicht nur, weil es ein Kommandozeilen-Programm ist, sondern auch, weil damit die Funktionen besser erklärt werden können als mit teilweise überladenen und mehr oder weniger guten Git-GUI-Programmen. Nichtsdestotrotz sieht es in der Praxis eher so aus, dass die meisten Personen GUI-Programme verwenden, weshalb ich hin und wieder Einschübe mit Git-GUI-Programmen einfügen werde. Eine Übersicht über eine Auswahl von grafischen Git-Programmen erfolgt in Kapitel 10.

Die erste Anwendung, die ich Ihnen vorstellen möchte, ist das Programm »Git GUI«, das bei Git mitgeliefert wird. Es bietet eine einfache Übersicht über das Repository und dort können nicht nur Commits erstellt, sondern auch Branches angelegt und gemergt werden.

Nicht nur Anfänger können von dem Tool profitieren, insbesondere wenn es darum geht, Diffs anzuschauen und Commits zu tätigen. Das Tool kann aus der Konsole ganz einfach im Projektverzeichnis aufgerufen werden.

```
$ git gui
```

Wer das Programm in der auf Deutsch übersetzten Fassung präsentiert bekommt, der sollte die deutsche Übersetzung am besten abschalten. Die Übersetzung ist nämlich in keinster Weise empfehlenswert und stark verwirrend, denn es wurden fast alle Wörter übersetzt, so auch die Git-Befehle wie `commit` und `push`. »Commit« wird etwa mit »Eintragen« übersetzt, was nicht sonderlich verständlich und eher irreführend ist.

Eine einfache Methode, über das Programm selbst diese Übersetzung auszuschalten, gibt es leider nicht, daher kann man es etwa so aufrufen:

```
$ LANG=C git gui
```

Damit nicht jedes Mal `LANG=C` beim Aufruf übergeben werden muss, kann man zur Konfiguration die `~/.bashrc` anlegen. Der Inhalt kann dann so aussehen:

```
export LANG=C
```

Dies hätte nach einem Neustart der Git-Bash den Effekt, dass alle Programme, die man aus der Bash heraus startet, in der Standard-Sprache ausgeführt werden. Das Kommandozeilenprogramm von Git wäre dann auch vollständig auf Englisch. Eine andere Möglichkeit ist die Definition eines Bash-Alias, ebenfalls in der `~/.bashrc`-Datei:

```
alias git-gui='LANG=C git gui'
```

Der Alias bewirkt, dass beim Aufruf von `git-gui` der Befehl `LANG=C git gui` ausgeführt wird. Es ist also nicht viel mehr als eine Definition eines anderen Aufrufnamens. Wer Git komplett inklusive Git-GUI auf Englisch nutzen möchte, kann auch diesen Alias definieren:

```
alias git = 'LANG=C git'
```

Eine weitere eher unschöne Option, das Sprachproblem zu lösen, ist die Entfernung der Sprach-Datei, die im Installationsverzeichnis liegt. Dies ist allerdings nicht empfehlenswert, da das Löschen der Sprachdateien ungeahnte und unschöne Nebenwirkungen haben kann.

2.6.1 Commits mit Git GUI

Die Standard-Ansicht von Git-GUI ist recht aufgeräumt. Die Oberfläche teilt sich in zwei Teile auf, die untereinander noch einmal aufgeteilt sind. Auf der linken Seite finden sich in der oberen Hälfte alle neuen, unversionierten und geänderten

Dateien. Diese können in den Staging-Bereich geschoben werden. Der Staging-Bereich findet sich direkt darunter, wo alle zum Commit vorgemerkten Dateien aufgelistet sind. Wenn Sie auf eine der Dateien klicken, sind auf der rechten Seite die Unterschiede zur Vorgängerversion, also der Diff, der Datei zu sehen. Bei einer neuen Datei ist das folglich die vollständige Datei. Unterhalb des Teil-Fensters lassen sich einige Aktionen durchführen, etwa ein erneutes Einlesen des Arbeitsverzeichnisses, das Stagen aller geänderten und neuen Dateien, das Signieren des Commits sowie das Committen und Pushen. Rechts daneben können Sie die Commit-Nachricht eintippen, die dann durch den Klick auf COMMIT als Commit gespeichert wird.

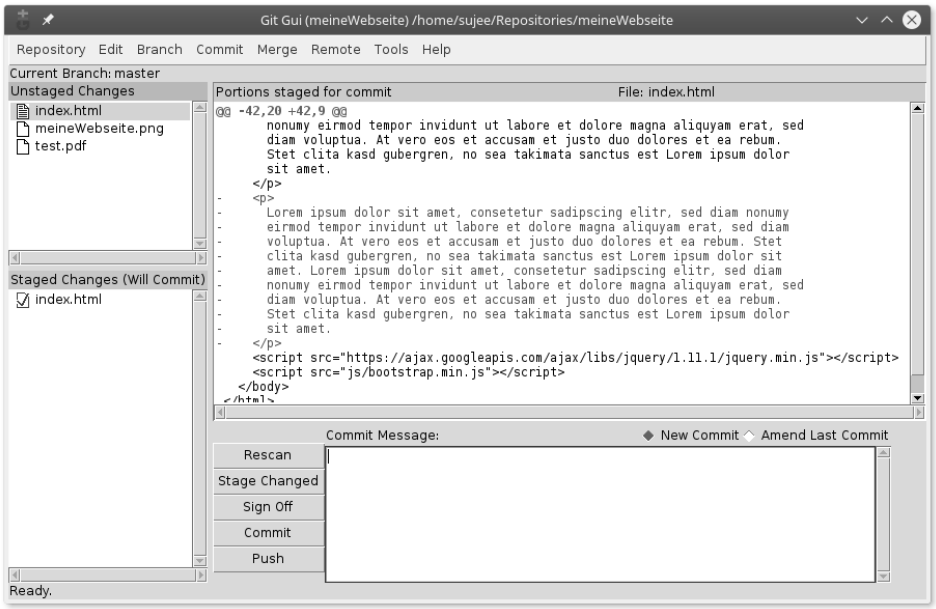


Abb. 2.13: Git GUI

Nicht ganz so ersichtlich ist, wie einzelne Dateien in den Staging-Bereich geschoben werden. Wenn Sie in der Übersicht der neuen und geänderten Dateien auf den Namen einer Datei klicken, wird ihr Inhalt angezeigt. Wenn sie allerdings in den Staging-Bereich geschoben werden soll, muss auf das kleine Datei-Icon links daneben geklickt werden. Das Herausnehmen aus dem Staging-Bereich klappt nach demselben Prinzip, nur dann eben in der Box darunter. Ganz intuitiv ist das Vorgehen leider nicht. Nichtsdestotrotz bietet dieses Tool eine einfache und übersichtliche Möglichkeit an, um sich Diffs anzuschauen und Commits zu tätigen. Dies dürfte für viele deutlich komfortabler sein, als sich mit der Kommandozeile abzumühen.

2.7 Wie Git arbeitet

In diesem ersten Praxiskapitel wurden zwar die ersten Commits getätigt, doch die Arbeit mit Git im Groben wurde noch nicht behandelt. Wenn man das Erzeugen von Commits verstanden hat, ist auch die Nutzung wesentlich einfacher, insbesondere wenn es später um das Erstellen und Mergen von Branches geht.

Im Gegensatz zu anderen Versionsverwaltungsprogrammen arbeitet Git sehr effizient, darunter fällt auch die Art und Weise, wie Git die Versionen einer Datei speichert. Prinzipiell sind viele verschiedene Arten möglich, eine Versionshistorie zu erstellen. Eine verschwenderische Möglichkeit ist es, bei jedem Commit alle Dateien des Repositorys erneut abzuspeichern. In der ersten Revision sind einmal alle Dateien zu dem damaligen Stand gespeichert und bei der zweiten Revision sind dann noch einmal alle Dateien zusätzlich gespeichert, darunter die veränderte Datei bzw. Dateien.

Aus Sicht der Speicherplatzbelegung ist dies der Worst Case, da durch mehrfach gespeicherte identische Dateien unglaublich viel Platz auf der Festplatte verbraucht wird. Die meisten Versionskontrollprogramme arbeiten auf Grundlage von Diffs, also Unterschieden zweier Dateien, die sie als Liste von Veränderungen sehen. Bei Git ist das allerdings etwas anders, da es quasi ein kleines Dateisystem ist. Jedes Mal, wenn ein Git-Commit erzeugt wird, speichert Git eine Art Snapshot, der die Dateien abbildet, und speichert eine Referenz dazu ab. Um Platz auf der Platte zu sparen, werden nur die veränderten Dateien erneut abgespeichert. Die alten Versionen bleiben natürlich weiterhin vorhanden, um Veränderungen darstellen zu können.

Die reine Speicherung der Diffs würde zwar weniger Speicherplatz auf der Festplatte belegen, wäre aber langsamer, etwa dann, wenn ein deutlich älterer Stand ausgecheckt wird. Wenn eine Datei 100-mal verändert wurde, wäre es beim Auschecken eines älteren Standes bis zu 100-mal notwendig, ein Diff anzuwenden, was durchaus langsam ist. Bei Git wird einfach die aktuelle Datei als Ganze ausgecheckt, ohne dass viel drum herum erledigt werden muss.

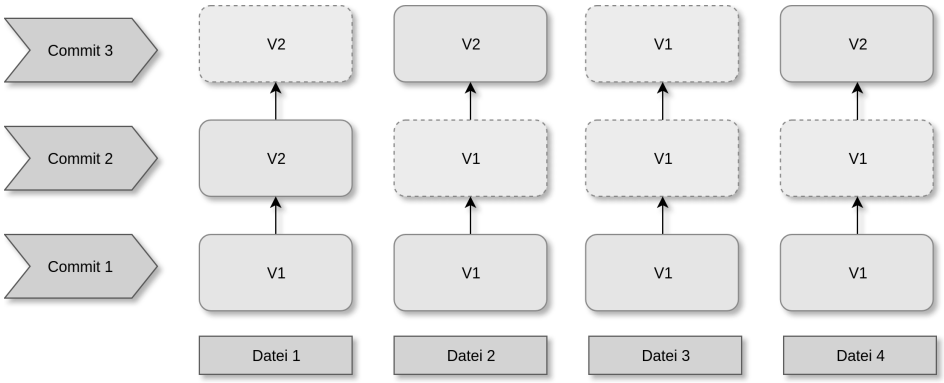


Abb. 2.14: Git speichert die Commits als Snapshots, nicht als Diffs.

Die Grafik soll das Verhalten nochmals verdeutlichen. Im ersten Commit wurden neue Dateien hinzugefügt. Diese liegen dementsprechend in Version 1 vor und alle Dateien wurden einmal gespeichert. Wenn im zweiten Commit nur `Datei 1` angepasst wird, wird die Datei als Ganzes in Version 2 gespeichert. Die übrigen Dateien bleiben unberührt. Im dritten Commit wurden `Datei 2` und `Datei 4` angepasst, die anderen bleiben unberührt.

Um auf das Beispiel-Projekt zurückzukommen, haben Sie im ersten Schritt alle benötigten Dateien des Bootstrap-Frameworks im Projektverzeichnis entpackt und in den ersten Commit geschoben. Da jede Datei vollständig neu für das Git-Repository war, wurden sie einmal komplett abgespeichert. Im zweiten Schritt haben Sie im zweiten Commit die `index.html`-Datei hinzugefügt, die als zusätzliche Datei dazugestoßen ist. Die anderen Dateien blieben für den zweiten Commit unberührt und wurden somit nicht noch ein weiteres Mal gespeichert. Die Dateien landen alle intern im `.git`-Verzeichnis.

Generell sollten Sie beim täglichen Arbeiten mit Git beachten, dass Git immer nur Dateien hinzufügt. Somit sind zwar gegebenenfalls Dateien in einer Revision gelöscht worden, sie sind aber immer noch in der Historie vorhanden. Wichtig ist zusätzlich, auch zu wissen, dass Git nur Dateien versionieren kann und keine Ordner. Leere Ordner lassen sich daher nicht in ein Repository committen. Dafür muss mindestens eine Datei enthalten sein.

Für diejenigen, die noch ein wenig tiefer einsteigen wollen, folgen nun ein paar genauere Erläuterungen, wie die Dateien von Git abgespeichert werden. Am besten lässt es sich nachvollziehen, wenn Sie ein neues Repository erstellen, in dem nur eine Datei und ein Commit erzeugt werden.

```
$ mkdir test-repo
$ git init
$ echo "Inhalt der Datei EINS" >> EINS
```

Mit diesen drei Befehlen wurde der Ordner des Repositorys erzeugt, gefolgt von dem leeren Repository selbst. Anschließend wurde noch die Datei `EINS` angelegt, die den Inhalt der Datei `EINS` trägt. Zu Beginn dieses Kapitels habe ich bereits erwähnt, dass das eigentliche Repository komplett im `.git`-Unterverzeichnis liegt.

```
$ ls -l .git
insgesamt 12
drwxr-xr-x 1 sujee sujee   0  1. Mär 11:50 branches
-rw-r--r-- 1 sujee sujee  92  1. Mär 11:50 config
-rw-r--r-- 1 sujee sujee  73  1. Mär 11:50 description
-rw-r--r-- 1 sujee sujee  23  1. Mär 11:50 HEAD
drwxr-xr-x 1 sujee sujee 328  1. Mär 11:50 hooks
```

```
drwxr-xr-x 1 sujee sujee  14  1. Mär 11:50 info
drwxr-xr-x 1 sujee sujee  16  1. Mär 11:50 objects
drwxr-xr-x 1 sujee sujee  18  1. Mär 11:50 refs
```

Die Dateien werden von Git im Verzeichnis `.git/objects` gespeichert. In einem leeren Repository liegen dort nur die Verzeichnisse `info` und `pack`. Nun sollten Sie die Datei durch die Erzeugung eines Commits im Repository speichern:

```
$ git add EINS
$ git commit -m "Füge Datei EINS hinzu"
```

Anschließend hat sich im `.git/objects`-Ordner etwas verändert:

```
$ tree .git/objects/
.git/objects
|-- b2
|   --- b476fa8df62a67f3c65446236d63d97444a89f
|-- c5
|   --- 8f415b63094763465e0a941eb2a38a37926b33
|   --- e1e17fe489de9897b8b1976471efb37ec9e747
|-- info
|-- pack
5 directories, 3 files
```

Es sind nun zwei weitere Ordner und drei Dateien von Git angelegt worden. Beim Erstellen einer Datei komprimiert Git die Datei und speichert sie anschließend ab. Die Datei bekommt einen eindeutigen Namen in Form eines Hashes, der anschließend im `.git/objects`-Verzeichnis gespeichert wird. Einfach ausgedrückt speichert Git nur den Namen bei einer unveränderten, komprimierten Datei in den Snapshot. Falls sich die Datei verändert hat, wird sie komprimiert im Objektordner gespeichert.

Git speichert den Hash eines Objekts sowohl im Ordner- als auch im Dateinamen. Die ersten zwei Zeichen landen daher im Ordnernamen, während der Rest im Dateinamen landet. Der einzige vorhandene Commit hat in diesem Test-Repository den Hash »c5e1e1«. Dazu existiert das passende Objekt im Verzeichnis `c5/e1e17fe489de9897b8b1976471efb37ec9e747`.

Mit `git cat-file -p` lassen sich die Inhalte der Objekt-Dateien anschauen:

```
$ git cat-file -p c5e1e17fe489de9897b8b1976471efb37ec9e747
tree b2b476fa8df62a67f3c65446236d63d97444a89f
```

```
author Sujeevan Vijayakumaran <mail@svij.org> 1546771236 +0100
committer Sujeevan Vijayakumaran <mail@svij.org> 1546771236 +0100

Füge Datei EINS hinzu
```

In der Datei stehen somit alle Informationen des Commits: Autor, Committer, Commit-Message und das dazugehörige `tree`-Objekt. Das `tree`-Objekt lässt sich nach demselben Schema ebenfalls untersuchen:

```
$ git cat-file -p b2b476fa8df62a67f3c65446236d63d97444a89f
100644 blob c58f415b63094763465e0a941eb2a38a37926b33    EINS
```

Das `tree`-Objekt ist der Snapshot des »Git-Dateisystems«. Dort sind die Dateien des Repositorys aufgelistet. In diesem Fall besteht es nur aus der einzelnen Datei. Gespeichert wird die Zugriffsberechtigung, hier `100644`, die Art des Objekts, hier `blob`, eine weitere Hash-Summe und der Dateiname. Es gibt zwei Arten von Git-Objekten: `tree` und `blob`. Würde ein weiteres Verzeichnis vorhanden sein, wäre neben dem `blob`- noch ein `tree`-Objekt vorhanden.

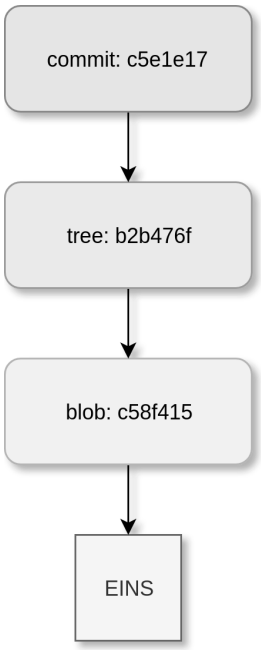


Abb. 2.15: Vereinfachte Ansicht der Git-Objekte im Repository mit einem Commit und einer Datei

Auch die letzten drei Hash-Summen sollten Sie sich anschauen:

```
$ git cat-file -p c58f415b63094763465e0a941eb2a38a37926b33
Inhalt der Datei EINS
```

Dieser Befehl gibt letztendlich den Dateinamen der gespeicherten Datei aus. Diese drei Objekte sind in dem einzigen Commit gespeichert. Ein wenig klarer wird die Funktionsweise von Git, wenn Sie eine weitere Datei hinzufügen:

```
$ echo "Inhalt der Datei ZWEI" >> ZWEI
$ git add ZWEI
$ git commit -m "Füge Datei ZWEI hinzu"
[master b41171b] Füge Datei ZWEI hinzu
[...]
```

In dem zweiten Commit-Objekt ist eine Information mehr vorhanden, als es noch im ersten Commit der Fall war:

```
$ git cat-file -p b41171b
tree 1467132782311186478 6ef248096be0095738724
parent c5e1e17fe489de9897b8b1976471efb37ec9e747
author Sujeevan Vijayakumaran <mail@svij.org> 1546771915 +0100
committer Sujeevan Vijayakumaran <mail@svij.org> 1546771915 +0100

Füge Datei ZWEI hinzu
```

Hinzugekommen ist nämlich das Parent-Objekt. Das gab es im ersten Commit nicht, da es keinen Parent-Commit gab. Da Commits verkettete Objekte sind, besitzt der erste Commit natürlich keinen Vorgänger-Commit. Das `tree`-Objekt enthält im zweiten Commit nun wie erwartet zwei Dateien:

```
$ git cat-file -p 1467132782311186478 6ef248096be0095738724
100644 blob c58f415b63094763465e0a941eb2a38a37926b33    EINS
100644 blob 577576a259cd64d349bb4a0a8fa1a9a76c400608    ZWEI
```

Die Hash-Summe von Datei `EINS` hat sich nicht verändert, da sich die Datei nicht verändert hat. Somit wird doppeltes Speichern vermieden und auf dasselbe Objekt verwiesen.

Die hier dargelegte Art und Weise soll einen Eindruck vermitteln, wie Git im Groben arbeitet. Da es bei mehreren Dateien, Revisionen und Branches komplizierter wird, habe ich es an dieser Stelle möglichst einfach gehalten.

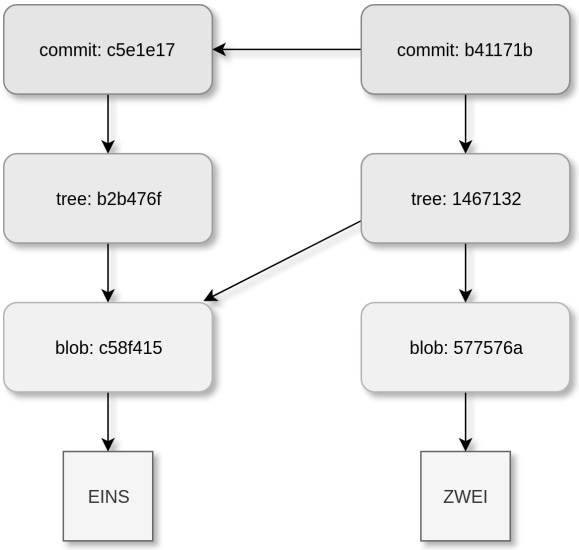


Abb. 2.16: Vereinfachte Ansicht der Git-Objekte im Repository mit zwei Commits und zwei Dateien

2.8 Git-Hilfe

Dieses Buch ist kein reines Nachschlagewerk und liefert somit keine vollständige Referenz für alle Funktionen und Kommandos, die Git zu bieten hat. Falls Sie einige Funktionen nutzen wollen, die in diesem Buch nicht beschrieben oder weiterhin unklar sind, können Sie sich die offizielle Git-Hilfe anschauen. Sie lässt sich über die Kommandozeile aufrufen.

```
$ git help
```

Die Ausgabe liefert ohne weiteren Parameter eine Liste der Basis-Befehle. Einige wurden in diesem Kapitel des Buches schon erläutert, viele aber noch nicht. Die Hilfe kann pro Befehl oder Eigenschaft aufgerufen werden.

```
$ git help commit
```

Der obige Befehl zeigt alle Parameter von `git commit` an, inklusive einer Beschreibung der Parameter.

2.9 Zusammenfassung

In diesem ersten Kapitel konnten Sie sehen, wie das erste lokale Git-Repository angelegt wird. Sie haben zudem gelernt, neue und geänderte Dateien zum Staging-

Bereich hinzuzufügen und anschließend als Commit zu speichern. Außerdem ging es darum, wie Sie einige Änderungen wieder aus dem Staging-Bereich herausnehmen und das Log und ein Diff anschauen können. Dies sind die Grundlagen, die nötig sind, um mit dem zweiten Kapitel fortfahren zu können, da Commits schließlich das Essenzielle eines Git-Repositorys sind.

Überblick der eingeführten Befehle

- `git init`
  - Legt ein neues, leeres Git-Repository an.
- `git config`
  - Ermöglicht das Setzen von Konfigurationsschaltern wie den Namen, die E-Mail-Adresse und den Standard-Editor.
- `git status`
  - Gibt sinnvolle und praktische Informationen an, wie den aktuellen Stand im Arbeitsverzeichnis des Repositorys zur aktuellen Zeit.
- `git add`
  - Fügt Änderungen an einer Datei in den Staging-Bereich hinzu, um sie für einen Commit vorzumerken.
- `git commit`
  - Speichert die hinzugefügten Änderungen aus dem Staging-Bereich als Commit fest ab. Erzeugt somit einen Eintrag in der Git-Historie.
- `git diff`
  - Zeigt die Unterschiede zwischen zwei Versionen einer oder mehrerer Dateien an.
- `git checkout`
  - Ermöglicht das Wechseln zwischen Branches und stellt Dateien aus dem Arbeitsverzeichnis wieder her.
- `git mv`
  - Verschiebt oder benennt eine Datei oder Verzeichnis um.
- `git reset`
  - Ein Soft-Reset ermöglicht ein Zurücksetzen des Commits, ohne die eigentliche Änderung zu entfernen. Die Änderung verbleibt im Staging-Bereich.
  - Ein Hard-Reset ermöglicht ein Zurücksetzen des Commits mitsamt der Entfernung der eigentlichen Änderung.
  - Ein Mixed-Reset setzt den Commit zurück, die Änderungen verbleiben im Arbeitsverzeichnis.
- `git revert`
  - Macht einen einzelnen Commit rückgängig und legt dafür einen neuen Commit an, der die besagte Änderung zurücknimmt.

Stichwortverzeichnis