

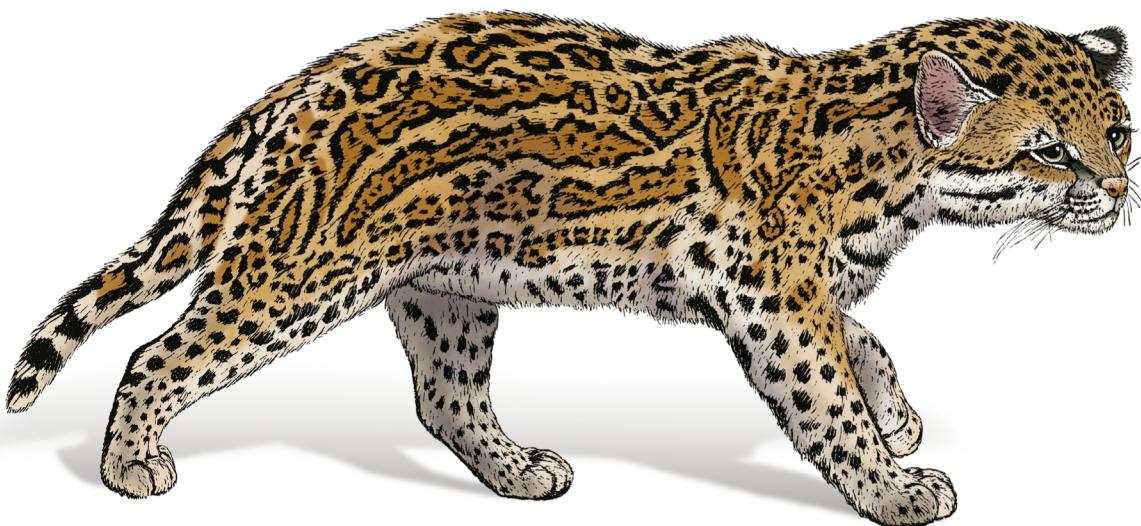
O'REILLY®

Inklusive  
Git-Basics

# GitHub

## Eine praktische Einführung

Von den ersten Schritten bis zu eigenen GitHub Actions



Anke Lederer

Papier  
plus<sup>+</sup>  
PDF.

Zu diesem Buch – sowie zu vielen weiteren O'Reilly-Büchern – können Sie auch das entsprechende E-Book im PDF-Format herunterladen. Werden Sie dazu einfach Mitglied bei oreilly.plus<sup>+</sup>: [www.oreilly.plus](http://www.oreilly.plus)

---

# GitHub – Eine praktische Einführung

*Von den ersten Schritten bis zu  
eigenen GitHub Actions*

*Anke Lederer*

O'REILLY®

Anke Lederer

Lektorat: Ariane Hesse

Korrektorat: Sibylle Feldmann, [www.richtiger-text.de](http://www.richtiger-text.de)

Satz: III-satz, [www.drei-satz.de](http://www.drei-satz.de)

Herstellung: Stefanie Weidner

Umschlaggestaltung: Michael Oréal, [www.oreal.de](http://www.oreal.de)

Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

ISBN:

Print 978-3-96009-141-7

PDF 978-3-96010-426-1

ePub 978-3-96010-427-8

mobi 978-3-96010-428-5

1. Auflage 2021

Copyright © 2021 dpunkt.verlag GmbH

Wieblinger Weg 17

69123 Heidelberg

Dieses Buch erscheint in Kooperation mit O'Reilly Media, Inc. unter dem Imprint »O'REILLY«. O'REILLY ist ein Markenzeichen und eine eingetragene Marke von O'Reilly Media, Inc. und wird mit Einwilligung des Eigentümers verwendet.

#### Hinweis:

Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.



#### Schreiben Sie uns:

Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: [kommentar@oreilly.de](mailto:kommentar@oreilly.de).

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.

Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.

Die Informationen in diesem Buch wurden mit größter Sorgfalt erarbeitet. Dennoch können Fehler nicht vollständig ausgeschlossen werden. Verlag und Autoren übernehmen keine juristische Verantwortung oder irgendeine Haftung für eventuell verbliebene Fehler und deren Folgen.

5 4 3 2 1 0

---

# Inhalt

<b>Vorwort .....</b>	<b>XI</b>
Ist dieses Buch das richtige für mich? .....	XI
Für wen ist dieses Buch nicht geeignet? .....	XII
Der Leser oder die Leserin? .....	XIII
Wie ist dieses Buch zu lesen? .....	XIII
Konventionen in diesem Buch .....	XIV
<b>Danksagung .....</b>	<b>XVI</b>
<b>1 Was ist GitHub, und wofür brauche ich es? .....</b>	<b>1</b>
Was bietet GitHub? .....	2
Einsatzgebiete von GitHub .....	3
Git, GitHub, GitLab – alles das Gleiche? .....	4
Mit welchen Kosten muss ich rechnen? .....	6
<b>2 GitHub – Wir verschaffen uns einen Überblick .....</b>	<b>9</b>
Anwendungsfälle für GitHub (oder: Was will ich da eigentlich?) .....	12
Informationen finden (interessierte Anwenderin) .....	13
Dateien finden und herunterladen (Hilfe suchender Programmierer) .....	17
<b>3 Die Basis: Das erste eigene GitHub-Projekt .....</b>	<b>19</b>
Account anlegen .....	20
Account schützen .....	22
Unsichtbar werden – die eigene Mailadresse schützen .....	23
Das erste eigene Repository anlegen .....	24
Eine inhaltliche Änderung am Projekt vornehmen .....	26
Den ersten Ablauf üben – Issue anlegen und bearbeiten .....	31
Einen Issue anlegen .....	32
Den Issue bearbeiten und schließen .....	35

Ein bestehendes Repository löschen . . . . .	39
Ein bestehendes Projekt hochladen . . . . .	40
<b>4 Die wichtigsten Grundlagen für eigene GitHub-Projekte . . . . .</b>	<b>43</b>
Den zweiten Ablauf üben – Branches, Pull-Requests und Merges . . . . .	44
Branch – unterschiedliche Handlungsstränge aufmachen . . . . .	44
Änderungen auf einem Branch vornehmen . . . . .	48
Pull-Request – Änderungen in Branches aufzeigen . . . . .	53
Merge – Änderungen aus Pull-Requests übernehmen . . . . .	59
Reviews durchführen . . . . .	61
Reviewer manuell anfordern . . . . .	65
Reviews automatisch zuweisen – CODEOWNERS . . . . .	65
Gutes schützen – Protected Branches . . . . .	67
Genehmigung vorschreiben – Required Reviews . . . . .	68
Genehmigung automatisch zurückziehen . . . . .	72
Genehmigung durch Eigentümer*innen vorschreiben . . . . .	73
Den Laden sauber halten – Vorlagen, Diskussionen eingrenzen . . . . .	75
Vorlagen für Issues . . . . .	75
Vorlagen für Pull-Requests . . . . .	79
Für Ruhe sorgen (Teil 1) – Locking Conversations . . . . .	80
Für Ruhe sorgen (Teil 2) – Interaction Limits . . . . .	83
<b>5 Rechtliches – Open-Source-Lizenzen . . . . .</b>	<b>85</b>
Warum Lizenzierung wichtig ist . . . . .	85
Lizenz Marke Eigenbau . . . . .	86
Welche Lizenzen gibt es? . . . . .	88
Softwarecode . . . . .	88
Musik, Bilder und Texte . . . . .	90
Was wählen andere als Lizenz? . . . . .	92
Welche Lizenz ist die richtige für mich? . . . . .	94
Wo finde ich mehr Infos und Unterstützung zu Lizenzen? . . . . .	95
Unterstützung bei der Wahl der richtigen Lizenz . . . . .	95
Tools und Informationen zu/über Lizenzen . . . . .	96
Eine Lizenz zu einem Repository hinzufügen . . . . .	97
<b>6 Unterstützung für GitHub-Projekte finden . . . . .</b>	<b>101</b>
Wie bringt man Leute dazu, beim eigenen Projekt mitzumachen? . . . . .	101
Dein Projekt auffindbar machen . . . . .	102
Dein Projekt anschaulich beschreiben . . . . .	104
Dein Projekt bekannt machen . . . . .	111
Dein Projekt (gegebenenfalls) zugänglich machen (Rechtevergabe) . . . . .	114

Ein Projekt finden, das du unterstützen möchtest. . . . .	117
Wer bin ich und, wenn ja, wie viele? . . . . .	117
Fremdes Projekt suchen . . . . .	117
Fremdes Projekt begutachten . . . . .	124
Fremdes Projekt unterstützen – Fork . . . . .	126
<b>7 Ein Projekt lokal mit Git verwalten . . . . .</b>	<b>135</b>
Warum GitHub allein manchmal nicht ausreicht . . . . .	135
Git, was ist das? – Eine kurze Einführung . . . . .	136
Versionsverwaltung . . . . .	136
Dezentral . . . . .	138
Exkurs: Umgang mit der Konsole . . . . .	139
Git installieren und einrichten . . . . .	141
Exkurs: Die Konsole für Git einrichten am Beispiel Bash (für Fortgeschrittene) . . . . .	143
Drei Schritte, um einen Branch farbig anzuseigen . . . . .	144
Anpassen der .bashrc . . . . .	145
Tiefer einsteigen . . . . .	146
Wie Git tickt – Staging . . . . .	147
Das eigene Projekt mit Git verwalten . . . . .	150
Das Arbeitsverzeichnis initialisieren . . . . .	150
Eine neue Datei ins lokale Repository einfügen . . . . .	151
Eine Datei im lokalen Repository ändern . . . . .	152
Dateien von der lokalen Versionsverwaltung ausschließen . . . . .	153
Branching in Git . . . . .	154
Branches erzeugen . . . . .	154
Zwischen Branches wechseln . . . . .	155
Binärdateien mit Git verwalten . . . . .	156
Installation von Git LFS . . . . .	157
Git LFS einrichten . . . . .	158
Sich weiter schlau machen über Git . . . . .	160
Oldschool: Bücher . . . . .	160
Neumodischer Kram: Internet . . . . .	160
<b>8 Git und GitHub im Zusammenspiel . . . . .</b>	<b>163</b>
Szenario 1: Lokales Git-Projekt auf GitHub hochladen . . . . .	165
Lokal ein Git-Repository mit einer Datei anlegen . . . . .	166
Leeres Repository auf GitHub anlegen . . . . .	166
Das Git- mit dem GitHub-Repository verknüpfen . . . . .	167
Git-Repository auf GitHub hochladen (pushen) . . . . .	170
Lokal Änderungen vornehmen und diese auf GitHub hochladen . . . . .	172

Szenario 2: Projekt auf GitHub lokal zu Git holen . . . . .	173
Ein neues GitHub-Repository mit einer Datei erstellen . . . . .	174
Das GitHub-Repo mittels Git lokal klonen . . . . .	174
Das GitHub-Repo anpassen und die Änderung in die lokale Git-Arbeitsumgebung holen. . . . .	175
Szenario 3: Geforktes Projekt auf GitHub lokal zu Git holen. . . . .	179
Wir forken auf GitHub ein Projekt . . . . .	180
Wir klonen den Fork lokal mittels Git. . . . .	180
Wir legen in Git ein zweites Remote-Repository fest. . . . .	181
Wir aktualisieren den lokalen Klon aus dem Originalprojekt . . . . .	182
Szenario 4: Lokale Änderung an Originalprojekt übergeben . . . . .	183
Wir richten alles so ein, wie in Szenario 3 beschrieben . . . . .	184
Wir editieren lokal eine Datei und pushen sie zum Fork auf GitHub . . . . .	184
Wir erstellen einen Pull-Request aus dem Fork an das Originalprojekt. . . . .	185
Wir üben uns in Geduld und warten auf das Mergen des Pull-Requests . . . . .	185
Merge-Konflikte lösen . . . . .	186
Wie entstehen Merge-Konflikte? . . . . .	186
Konflikte auflösen mit GitHub (Webeditor) . . . . .	187
Konflikte auflösen mit Git (Konsole) . . . . .	189
Log-in-Erlichterungen bei HTTPS . . . . .	194
Zugangsdaten auf Zeit zwischenspeichern (meine Empfehlung) . . . . .	194
Zugangsdaten dauerhaft speichern . . . . .	195
<b>9 Der GitHub Marketplace – Actions und Apps . . . . .</b>	<b>197</b>
Was können Actions und Apps? . . . . .	197
Eine App aus dem Marketplace installieren . . . . .	198
App installieren . . . . .	199
App anpassen . . . . .	202
Eine Action aus dem Marketplace installieren . . . . .	204
Action installieren . . . . .	205
Action ausprobieren und feinjustieren. . . . .	208
Hinter den Kulissen einer Action . . . . .	210
Eine eigene Action erstellen (für Fortgeschrittene) . . . . .	212
Einige grundlegende Begriffe . . . . .	212
Anatomie einer Action . . . . .	215
Unseren Anwendungsfall einrichten . . . . .	216
Unseren Anwendungsfall verstehen. . . . .	219
Passwörter geheim halten – GitHub Secrets . . . . .	222

<b>10 Pimp my Repo – Weitere GitHub-Features . . . . .</b>	<b>225</b>
Websites aus GitHub generieren (GitHub Pages) . . . . .	225
GitHub Pages einrichten . . . . .	226
GitHub Pages verschönern mit dem Theme Chooser . . . . .	227
GitHub Pages ausbauen – die Navigation einrichten . . . . .	229
GitHub Pages – weitere Themes . . . . .	233
Angriff der Klone – Repo-Templates anlegen . . . . .	235
Eigene Projektboards – mit Projects den Überblick behalten . . . . .	236
Grundlegendes zu Projektboards . . . . .	236
Ein eigenes Projektboard erstellen . . . . .	239
Projektboard automatisieren . . . . .	241
<b>11 Nützliches und Kuriöses rund um GitHub . . . . .</b>	<b>245</b>
GitHub auf der Kommandozeile . . . . .	245
GitHub CLI . . . . .	245
Hub . . . . .	247
GitHub-API (für Fortgeschrittene) . . . . .	247
Sich mit GitHub weiter auseinandersetzen . . . . .	248
GitHub Learning Lab . . . . .	248
Weitere Ressourcen zum Recherchieren . . . . .	250
Editoren und Handy-Apps . . . . .	250
Klein und schlank – Atom . . . . .	251
Visual Studio Code . . . . .	252
Für Website-Gestalter – Brackets . . . . .	252
GitHub Desktop . . . . .	253
GitHub auf dem Handy – GitHub Mobile . . . . .	253
GitHub auf dem Handy – Octodroid . . . . .	254
Nützliches und kleine Spielereien . . . . .	255
Übersetzungsmanagementtools – crowdin und Weblate . . . . .	255
Zeigen, wo man steht – Badges . . . . .	256
Sag es mit einem Bild – Gitmoji . . . . .	258
Ideen für eigene Repositories – ohne programmieren . . . . .	259
<b>A Gängige Git-Befehle zum Nachschlagen . . . . .</b>	<b>261</b>
<b>B Quellcode . . . . .</b>	<b>263</b>
<b>C Glossar (oder: Was bedeutet noch mal ...?) . . . . .</b>	<b>265</b>
<b>Index . . . . .</b>	<b>271</b>



---

# Vorwort

Liebe Leserin, lieber Leser,

schön, dass du hier bist! Warum ist dieses Buch entstanden? Als ich anfing, mich mit GitHub zu beschäftigen, war ich zunächst erschlagen von der Plattform. Ich fragte mich, wo und wie ich einsteigen sollte und was überhaupt wichtig sei. Ich habe daraufhin versucht, die unzähligen Quellen, die einen leichten Einstieg versprochen haben, zu durchforsten, um eben diesen Einstieg zu bekommen. Ich habe brillante Tutorials und grottenschlechte Videos gefunden und viel Zeit damit verbracht, die Infos zu filtern und nach Nützlichkeit zu sortieren.

Gewünscht hätte ich mir einen kompakten Einstieg, der an einer Stelle alles für eine Anfängerin Relevante zusammenfasst und erklärt – ohne einen neuen, unbekannten Fachbegriff in unverständlichem Fachchinesisch. Am liebsten wäre mir ein Buch gewesen, das mich an die Hand nimmt und mir schrittweise zeigt, was wichtig ist, wo ich was finde und wie das alles eigentlich geht. Das gab es aber in dieser Form nicht. Also habe ich versucht, eine möglichst praktische und hoffentlich verständliche Einführung zu schreiben, und du hast sie gerade vor deiner Nase.<sup>1</sup>

## Ist dieses Buch das richtige für mich?

Du bist hier richtig, wenn du dich bei der einen oder anderen nachfolgenden Beschreibung wiederfinden kannst:

- Du programmierst in deiner Freizeit gern Apps und suchst einen Weg, um sie zu veröffentlichen. Vielleicht möchtest du sogar zusammen mit deinen Freunden gemeinschaftlich an einer App arbeiten.

---

<sup>1</sup> Ich freue mich über Feedback: [githubbuch@ist-einmalig.de](mailto:githubbuch@ist-einmalig.de) – ja ich weiß, es klingt etwas überheblich, die Alternativen wären [@alphafrau.de](http://alphafrau.de) oder [@streber24.de](http://streber24.de) gewesen ...

- Du warst schon immer fasziniert von Open Source und hast dich stets gefragt, wie und wo man solche Projekte eigentlich unterstützen kann. Zudem fragst du dich, ob man dafür zwingend programmieren müssen muss.<sup>2</sup>
- Du benutzt schon lange eine App und möchtest die Entwicklerinnen und Entwickler über einen Fehler, der dich nervt, informieren. Beim Klicken auf *Fehler melden* bist du auf einer GitHub-Seite gelandet und bist erst einmal nicht weitergekommen. Das wurmt dich, und du möchtest es ändern.
- Du hast schon einmal versucht, dich mit GitHub auseinanderzusetzen, aber das war dir dann doch alles zu kompliziert. Du möchtest jetzt gern einen neuen Versuch starten und erhoffst dir, endlich wirklich zu verstehen, wie das alles läuft und warum Pull-Requests nicht Push-Requests heißen.<sup>3</sup>
- Du möchtest eigentlich nur wissen, wo und wie du den Quellcode zu einer bestimmten Software herunterladen kannst.<sup>4</sup>
- Du benutzt GitHub schon eine Weile, aber bis auf ein wenig Geklick auf der Weboberfläche hast du dich noch nicht viel damit beschäftigt. Jetzt möchtest du intensiver einsteigen und vor allem auch dieses ominöse Git ausprobieren.
- Du benutzt Git schon eine Weile und möchtest jetzt die Funktionen von GitHub kennenlernen.

Ich habe dieses Buch primär für Anfängerinnen und Anfänger geschrieben, die sich entweder noch gar nicht oder erst ein bisschen mit GitHub beschäftigt haben. Profis hingegen werden inhaltlich vermutlich nicht viel Neues finden.

Ich setze voraus, dass du auf deinem Computer mit dem Betriebssystem deiner Wahl (beispielsweise Windows, macOS, Linux etc.) auf Anwendungsniveau zurechtkommst. Du kannst z.B. einen Browser öffnen und eine Webseite aufrufen, Software installieren, ein Verzeichnis anlegen sowie Dateien erstellen und kopieren.

## Für wen ist dieses Buch nicht geeignet?

Für wen ist dieses Buch vermutlich nichts?

- Du möchtest dich intensiv mit Git beschäftigen und in die tiefsten Tiefen abtauchen, GitHub interessiert dich - wenn überhaupt - nur am Rande. Git werden wir jedoch nur relativ oberflächlich behandeln. Leseempfehlungen für einen tieferen Einstieg gebe ich in Kapitel 7, Abschnitt »Sich weiter schlau machen über Git« auf Seite 160.
- Deine erste Aktivität nach dem Frühstück ist es, alle eingegangenen Pull-Requests zu überprüfen und ein paar Merges durchzuführen, bevor du dir

---

2 Spoiler: Nein, muss man nicht.

3 GitHub-Profis werden darüber vermutlich die Stirn runzeln, aber genau solche Fragen stellen sich Neulinge. Und Spoiler: Wir werden diese Frage gemeinsam klären.

4 Auch diese Frage klären wir, sollte das aber deine einzige zu GitHub sein, empfehle ich dir eher eine entsprechende Webrecherche als dieses Buch.

nach dem Mittag alle neuen Issues anschaut und zur Kaffeezeit überlegst, ob nicht mal ein neuer Branch fällig wäre.

- Du bist für dein Unternehmen auf der Suche nach einem Werkzeug, das das gemeinschaftliche Arbeiten unterstützen soll. Dafür interessieren dich die businessrelevanten Informationen, z.B. welche Business-Features GitHub anbietet, wie man es selbst hosten könnte und ob sich der Business Case rechnet.
- Du liebst knallharte IT-Fakten und findest es albern, wenn man versucht, IT anhand von Beispielen, Bildern oder Vergleichen zu erklären.<sup>5</sup>
- Du findest Fußnoten nervig.<sup>6</sup>

Da GitHub eine lebende Plattform ist, kann es sein, dass sich nach Druck des Buchs schon wieder einiges geändert hat. Buttons könnten an einer anderen Stelle sein, vorgestellte GitHub-Projekte (auch Repositories oder kurz Repos genannt, übersetzt »Aufbewahrungsart«) nicht mehr existieren oder verwaist sein oder neue Features zur Verfügung stehen. Das sollte allerdings kein Problem sein, da dieses Buch die grundlegenden Prozesse beschreibt, sodass du dich mit diesem Wissen auch auf einer veränderten Benutzungsoberfläche zurechtfinden solltest. Ich werde im Verlauf dieses Buchs die Begriffe Projekt und Repository synonym verwenden.

## Der Leser oder die Leserin?

Ich persönlich bin ein großer Fan davon, alle Menschen gleichermaßen mit einzubeziehen, weswegen ich das generische Maskulinum<sup>7</sup> für nicht mehr zeitgemäß halte. Um Konstruktionen wie »Mein\_e Leser\_in, der\_die mein Buch liest« oder »Mein\*e Leser\*in, der\*die mein Buch liest« zu vermeiden, werde ich je nach Kontext entweder eine Beidnennung vornehmen (»Leserinnen und Leser«), das Gendersternchen (»Leser\*innen«), das generische Maskulinum (»der Leser«) oder das generische Femininum (»die Leserin«) verwenden, gemeint sind damit immer alle Geschlechter.

## Wie ist dieses Buch zu lesen?

Manche Menschen lesen ein Fachbuch von vorne bis hinten durch, andere springen in die Kapitel, die für sie attraktiv klingen. Ich habe dieses Buch so konzipiert, dass ein blutiger Anfänger es von vorne nach hinten durchlesen sollte. Sofern du bereits etwas Erfahrung hast, kann ein »Kapitel-Hopping« eventuell sinnvoll sein, beispielsweise wenn du schon weißt, wie man mit einem eigenen Projekt auf Git-

5 Wir werden in diesem Buch Häuser renovieren, Kinderkostüme aufhübschen, Teenager zum Küche putzen bewegen und Laster beladen ...

6 Schau mich nicht so an, ich habe mir das von Christina Czeschik und Matthias Lindhorst aus »Weniger schlecht über IT schreiben« abgeschaut. Ein Buch, das ich übrigens sehr empfehlen kann!

7 [https://de.wikipedia.org/wiki/Generisches\\_Maskulinum](https://de.wikipedia.org/wiki/Generisches_Maskulinum)

Hub umgeht, und nur wissen willst, wie du Open-Source-Projekte, die du unterstützen möchtest, finden kannst. Dann ist es eventuell durchaus sinnvoll, in das entsprechende Kapitel zu springen. Ich persönlich empfehle aber ein vollständiges, lineares Lesen, und das nicht nur, weil ich mir so viel Mühe gemacht habe :).

Dieses Buch enthält viele Links auf andere GitHub-Repositories und Websites. Da einige davon aufwändig abzutippen sind, findest du sie, um dir die Recherche zu erleichtern, auch in diesem Repository auf GitHub: <https://github.com/githubbuch/githubbuch.github.io> (siehe auch Abbildung 1).



Abbildung 1: Alle Links in diesem Buch sind in dem Repository <https://github.com/githubbuch/githubbuch.github.io> zu finden.

## Konventionen in diesem Buch

Ich weiß nicht, wie es dir geht, aber immer wenn ich in einem Buch die Abschnittsüberschrift »Konventionen« lese, überspringe ich das Kapitel am liebsten, weil meist nichts Spannendes drinsteht. Insofern halte ich es kurz.

Wir werden teilweise auf der Konsole arbeiten (unter Windows häufig auch Eingabeaufforderung genannt, siehe Abbildung 1). Das werde ich folgendermaßen darstellen:

```
$ ls  
datei.txt
```

Wenn ich exemplarisch Eingaben auf der Konsole zeigen möchte, also etwas, was du so nicht eins zu eins eintippen solltest, wähle ich Großbuchstaben:

```
$ vi DATEINAME
```

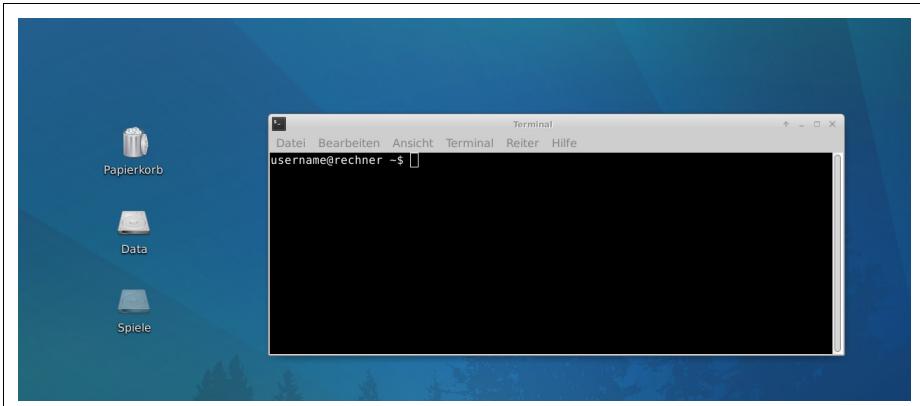


Abbildung 1: Die Konsole ist im Gegensatz zu einer grafischen Schnittstelle eine textbasierte Schnittstelle zum Betriebssystem und ermöglicht einen direkten Zugriff auf Betriebssystemressourcen und Programme.

Hier soll der Texteditor vi mit einer Datei deiner Wahl gestartet werden, DATEINAME dient als Platzhalter. Wer mit der Konsole und deren Ausgaben noch nicht gearbeitet hat, für den habe ich extra einen Abschnitt geschrieben (siehe Kapitel 7, Abschnitt »Exkurs: Umgang mit der Konsole« auf Seite 139).

Konsolenbefehle oder Ausschnitte aus Code-Beispielen im Fließtext, werden in Listingschrift dargestellt. URLs, E-Mail-Adressen, Dateinamen oder Dateierweiterungen sind in *Kursivschrift* formatiert.

Tipps, Anmerkungen und alles, was ich für hervorhebenswert halte, kommen in folgende schnuckelige Kästen:



### Tippkasten

In einem solchen Kasten werde ich wichtige Hinweise, Tipps, Tricks und Best Practices kurz hervorheben, die meiner Meinung nach hilfreich und/oder interessant sein könnten.

Stolperfallen und alles, was das Leben schwerer machen könnte, wenn man es nicht kennt und beachtet, werde ich in solche Kästen schreiben:



### Stolperfallenkasten

In einem solchen Kasten werde ich Stolperfallen kenntlich machen.

Für längere Erklärungen gibt es die »Erklärbärbox«:

### Erklärbärbox

In diese Boxen kommen längere Erläuterungen zu einem bestimmten Thema, die den Lesefluss im »normalen« Text stören würden. Falls dich das Thema nicht interessiert oder du darüber schon gut Bescheid weißt, kannst du diese Boxen einfach überspringen.

Alles, was meiner Ansicht nach den Lesefluss stören würde, werde ich in Fußnoten packen: beispielsweise Webadressen, Details, die für das weitere Verständnis nicht unbedingt wichtig sind, und auch persönliche Kommentare.

## Danksagung

Danksagungen sind für Leser\*innen häufig öde, daher mache ich es kurz und schmerzlos:

- Als Erstes möchte ich meinem Mann **Rüdiger** danken, der mich während des Schreibens oft genug aus meinem Schreibtunnel hervorlocken oder aber meine Begeisterung für irgendwelche fachlichen Details ertragen musste.
- Weiterer Dank gebührt meinen ganzen Querleser\*innen, die das Buch in Teilen gelesen und mir Feedback gegeben haben: **Henry Hillje, Alina Robbers, Halil Ege, Jan Schnedler** und (wieder) **Rüdiger**.
- Dann gab es noch ein paar Verrückte, die das Buch als Ganzes gelesen und mir ebenfalls sehr hilfreiches Feedback gegeben haben: **Julia Barthel, Florian Diedrich, Sven Riedel, Nina Siessenger** und **Ayleen Weiß**. Dank eurer Unterstützung sind noch einige Praxistipps eingeflossen, konnte ich einige Unklarheiten ausräumen und habe zudem meinen wortreichen Schreibfluss etwas im Zaum gehalten ;-).
- Danke auch an meine Lektorin **Ariane Hesse**, die (hoffentlich) alle Stolpersteine gefunden und dadurch dieses Buch zu einem besseren gemacht hat.
- Nicht vergessen möchte ich **Christina Czeschik** und **Matthias Lindhorst**, beide Autor\*innen des Buchs *Weniger schlecht über IT schreiben*, die mich durch ihr Buch überhaupt erst dazu inspiriert und motiviert haben, selber ein Buch zu schreiben. Danke dafür!
- Danken möchte ich auch der **Open-Source-Community**. Ohne die unermüdliche Unterstützung dieser Menschen wäre die (Software-)Welt eine ärmere. Ich werde daher ein Teil des Bucherlöses an einige Open-Source-Projekte spenden.



#### Git als Unterstützungstool für dieses Buch

Dieses Buch ist mithilfe von Git erstellt worden. Dafür habe ich 882 Commits auf dem master-Branch durchgeführt.

## KAPITEL 1

# Was ist GitHub, und wofür brauche ich es?

GitHub ist eine der wichtigsten Anlaufstellen im Internet für Softwareentwickler\*innen, Open-Source-Enthusiast\*innen und andere Interessierte. Viele bekannte Projekte werden dort weiterentwickelt und veröffentlicht, wie beispielsweise Googles User-Interface-Entwicklungskit *Flutter*<sup>1</sup>, das Machine-Learning-Framework *TensorFlow*<sup>2</sup> (ebenfalls Google) oder das CAD-Programm *FreeCAD*<sup>3</sup> (CAD steht für *Computer Aided Design*). Auch wer Bibliotheken und Erweiterungen für Programmiersprachen oder Unterstützung bei Projekten mit dem günstigen Einplatinencomputer Raspberry Pi sucht, wird häufig auf einem GitHub-Repository landen. Spätestens seit der Corona-Warn-App<sup>4</sup>, deren Quellcode auf GitHub veröffentlicht wurde,<sup>5</sup> ist die Plattform auch einem breiteren Publikum namentlich bekannt.

Unternehmen wie Facebook, Ford, Spotify, 3M und viele weitere haben die Plattform ebenfalls für sich entdeckt und entwickeln und veröffentlichen dort eigene Projekte. Mittlerweile ist es sogar nicht unüblich, dass potenzielle Arbeitgeber\*innen einen Blick auf den GitHub-Account einer sich bewerbenden Person werfen und diesen in ihre Entscheidungsfindung mit einfließen lassen. Es kann also viele Gründe geben, sich mit GitHub beschäftigen zu wollen.



### Am Ende des Kapitels weißt du ...

- was GitHub ist und wofür es eingesetzt wird.
- wie Git und GitHub zusammenhängen.
- welche Funktionen von GitHub kostenlos nutzbar sind.

1 <https://github.com/flutter/flutter>

2 <https://github.com/tensorflow/tensorflow>

3 <https://github.com/FreeCAD/FreeCAD>

4 <https://www.coronawarn.app/de/>

5 <https://github.com/corona-warn-app>

## Open Source

Auf den ersten Blick könnte man denken, mit Open Source sei ausschließlich gemeint, den Quellcode (Quelle, englisch *Source*) offenzulegen. Das ist aber nicht das einzige Merkmal. Vom Grundgedanken her gewährt Open Source die folgenden vier Freiheiten:

- **Verwenden:** die Software für beliebige Zwecke nutzen.
- **Vertreiben:** die Software uneingeschränkt an andere weitergeben.
- **Verändern/verbessern:** die Software an eigene Bedürfnisse anpassen.
- **Verstehen:** den Quellcode untersuchen.

Es existieren Hunderte von Open-Source-Lizenzen, die jeweils eine andere Kombination dieser Freiheiten regeln. Das sehen wir uns in Kapitel 5 noch etwas genauer an. Open-Source-Software ist meistens kostenlos, muss es aber nicht sein.

Rund um das Thema Open Source wirst du zudem eine Vielzahl von Begriffen wie *Free Software*, *Libre Software*, *Free and Open Source Software* (FOSS) sowie *Free/Libre and Open Source Software* (FLOSS) finden. Diese Begriffe haben nicht unbedingt die gleiche Bedeutung, sind inhaltlich aber verwandt. Lass dich davon jedoch nicht verwirren. Diese Namensvielfalt kommt daher, dass jeweils unterschiedliche Aspekte von Freiheit besonders betont werden.<sup>6</sup> Ich werde in diesem Buch den Begriff Open Source verwenden.

## Was bietet GitHub?

GitHub ist ein webbasierter kollaborativer Hosting-Dienst für Git-Projekte. »Aha«, mag sich der eine oder die andere denken und ist immer noch nicht schlauer. Gehen wir das mal gemeinsam durch.

- **Webbasiert** bedeutet nichts anderes als »befindet sich im Internet«.
- **Kollaborativ** bedeutet »zusammenarbeiten«.
- Ein **Hosting-Dienst** ist ein Dienst, der Ressourcen wie beispielsweise Speicherplatz oder Softwareanwendungen bereitstellt. Häufig handelt es sich dabei um »Plattenspeicherplatz«, um Dateien im Internet ablegen zu können, beispielsweise für eine Webseite.
- **Git**<sup>7</sup> ist eine freie Software zur Versionsverwaltung von Dateien. Versionsverwaltungen bieten Werkzeuge an, um mit Arbeits- und Zwischenständen von Dateien besser arbeiten zu können (mehr dazu später im Abschnitt »Versionsverwaltung« auf Seite 136 in Kapitel 7). Git wird in der Regel lokal auf dem eigenen Rechner installiert und verwaltet häufig programmierten Code. Aber

<sup>6</sup> Wer tiefer eintauchen möchte: <https://fsfe.org/freesoftware/basics/comparison.de.html>.

<sup>7</sup> <https://git-scm.com/>

auch Webseiten, Firmenkorrespondenz oder Gedichte wären möglich (genau genommen alle Arten von Textdateien). Darüber hinaus gibt es die Möglichkeit, einen eigenen Git-Server zu installieren.

- Unter **Projekten** sind alle Dateien zu verstehen, die zu einem bestimmten Thema gehören. Ein Projekt könnte beispielsweise eine Vereins-Website sein, die neben den eigentlichen Seiten beispielsweise Bilder und Beitriffsformulare für den Verein beinhaltet.

GitHub bietet also *Projekte*, die mit der Software *Git* verwaltet werden, eine *Speichermöglichkeit im Internet* an, um *zusammen mit anderen* daran arbeiten zu können. Manche bezeichnen es auch als soziales Netzwerk rund um Softwareprojekte.

Vielleicht fragst du dich gerade, ob du Git jetzt installieren musst, um GitHub überhaupt nutzen zu können. Die Antwort lautet: Nein, GitHub geht auch ohne Git!<sup>8</sup> Wir werden sogar den Großteil dieses Buchs ohne Git auskommen und erst in einem späteren Kapitel Git zu Hilfe nehmen (siehe Kapitel 7).

GitHub basiert auf vielen Konzepten von Git, wie beispielsweise dem Erstellen von Branches oder dem Durchführen von Merges. Es vereinfacht einige Git-Vorgänge dadurch, dass Aktivitäten durch einfaches Anklicken im Browser durchgeführt werden können, anstatt auf der Konsole Befehle mit mehreren Parametern angeben zu müssen. GitHub stellt aber auch noch einige zusätzliche Funktionen bereit, um das kollaborative Arbeiten zu erleichtern. All das wirst du im Verlauf des Buchs kennenlernen.

## Einsatzgebiete von GitHub

Wofür kann ich GitHub denn jetzt konkret einsetzen? GitHub ist vor allem dann sinnvoll, wenn du vorhast, mit mehreren Personen gemeinschaftlich an einem Projekt zu arbeiten (Stichwort *kollaborativ*). Ich habe bisher folgende Anwendungsfälle für GitHub identifiziert:

1. Gemeinschaftlich mit mehreren Personen an einem Projekt arbeiten.
2. Ein Projekt bzw. die Ergebnisse eines Projekts veröffentlichen, häufig nach dem Prinzip »fire and forget« (sinngemäß: »einmal veröffentlichen und danach nicht weiter anpassen«) – das ist beispielsweise immer mal wieder auch bei (Programmier-)Büchern anzutreffen.
3. Ein Projekt veröffentlichen, um anderen interessierten Personen eine Schnittstelle zu bieten, beispielsweise um zu unterstützen oder Fehler zu melden.
4. Dateien »im Internet« abspeichern, um schnell und einfach von überall auf sie zugreifen zu können.<sup>9</sup> Oft ist das mit dem Wunsch verbunden, zeitgleich an-

---

<sup>8</sup> Andersherum natürlich ebenfalls: Git geht auch ohne GitHub!

<sup>9</sup> Ich kenne Menschen, die für ihren personalisierten Linux-Arbeitsplatz die entsprechenden Konfigurationsdateien auf GitHub stellen, um sie mit wenigen Befehlen auf ihren aktuellen Rechner ziehen zu können.

deren die Dateien ebenfalls zur Verfügung zu stellen (wie bei 2) sowie Feedback und Unterstützung zu bekommen (wie bei 3).

Wobei die Übergänge fließend sind. Zu einem gemeinschaftlichen Projekt können plötzlich – manchmal ungefragt – interessierte Personen dazustoßen, oder ein ehemals stark frequentiertes Projekt verwaist und ist nur noch eine historische Veröffentlichung ohne weiteres Leben.

Eine große Stärke von GitHub ist es, ein Repository relativ einfach mit anderen Diensten verknüpfen zu können, seien es Containerlösungen, die einem das Ausliefern von Software erleichtern (etwa Docker), Übersetzungsunterstützungstools, Projektmanagementanwendungen und vieles mehr.

Ein wichtiger Punkt ist das Thema Sprache. Wer sich auf GitHub tummelt, sollte englisch lesen und schreiben können, da viele Informationen, Tutorials und Projekte nur auf Englisch verfügbar sind und die meiste Kommunikation auf Englisch stattfindet. Deutschsprachige Projekte gibt es zwar auch, sie sind aber eher selten.

Grundsätzlich sind die auf GitHub veröffentlichten Dokumente erst einmal für jeden und jede offen einsehbar. Es gibt aber auch die Möglichkeit, private Repositories anzulegen, darauf werde ich in Kapitel 3, Abschnitt »Das erste eigene Repository anlegen« auf Seite 24, noch eingehen.

## Git, GitHub, GitLab – alles das Gleiche?

Wir haben bereits erklärt, dass GitHub etwas mit der Versionsverwaltungssoftware Git zu tun hat. Die beiden arbeiten eng zusammen, sind aber nicht dasselbe oder würden einander gar ersetzen. Beide sind unabhängig voneinander nutzbar – ich kann ausschließlich mit Git arbeiten oder ausschließlich mit GitHub oder eben beide in Kombination einsetzen. Was ich wann brauche, hängt davon ab, was ich eigentlich genau machen will. Insbesondere bei Softwareentwicklungsprojekten werden beide häufig zusammen verwendet.

### Namensgebung GitHub

Der Name GitHub kommt von seiner engen Verzahnung mit Git und lässt sich grob mit »Knotenpunkt für Git-Anwendungen« übersetzen. Initiator und Namensgeber von Git ist Linus Torvalds, der vor allem als Entwickler des Betriebssystems Linux bekannt ist.<sup>10</sup> Das Wort »Git« (ausgesprochen wie im Deutschen) bedeutet wörtlich übersetzt übrigens »Depp« oder »Blödmann«, und diese ungewöhnliche Namenswahl wird von Torvalds wie folgt begründet:

<sup>10</sup> Aber nicht nur – Torvalds ist auch bekannt für seine Schimpftiraden, die mittlerweile auf eigenen Webseiten gesammelt werden, beispielsweise auf <https://www.reddit.com/r/linusrants/>.

»I'm an egoistical bastard, and I name all my projects after myself. First ›Linux‹, now ›Git‹.«

»Ich bin ein egoistischer Mistkerl, und ich benenne all meine Projekte nach mir. Zuerst ›Linux‹, jetzt eben ›Git‹.«

– Linus Torvalds

Mit einem Augenzwinkern könnte man »GitHub« daher auch mit »Deppentreff« übersetzen. ;)

Neben GitHub gibt es weitere Plattformen, wie beispielsweise *GitLab* oder *Bitbucket*, die im Grunde einen ähnlichen Funktionsumfang bieten. Nicht unerwähnt lassen möchte ich, dass GitHub 2018 von Microsoft gekauft wurde und viele Entwickler\*innen das bis heute kritisch betrachten. Deswegen gab es diverse Projekte, die von GitHub nach GitLab umgezogen sind, und manche Projekte findet man heute sogar auf beiden Plattformen.

In Tabelle 1-1 habe ich Git, GitHub und GitLab gegenübergestellt, damit du ein Gefühl für die Unterschiede bekommst. An den Zahlen erkennst du auch, dass GitHub deutlich größer ist als GitLab. Was teilweise auch darin begründet liegt, dass GitLab später gestartet ist.

Tabelle 1-1: Übersicht und Vergleich von Git, GitLab und GitHub

Name	Git	GitLab	GitHub
Logo	 git	 GitLab	 GitHub
Aufgabe	dezentrales Versionsverwaltungssystem	Onlineplattform für (Git-)Projekte	Onlineplattform für (Git-)Projekte
Webseite	<a href="https://git-scm.com/">https://git-scm.com/</a>	<a href="https://about.gitlab.com/">https://about.gitlab.com/</a>	<a href="https://www.github.com">https://www.github.com</a>
Zahlen, Daten, Fakten	k. A.	100.000 Unternehmen und Organisationen, 30 Millionen registrierte Nutzer*innen (Stand Dezember 2020)	2,1 Millionen Unternehmen und Organisationen, 100 Millionen Repositories, 40 Millionen registrierte Nutzer*innen (Stand August 2019)
Erscheinungsjahr	2005	2011	2008

Im Verlauf unserer gemeinsamen Entdeckungsreise werden wir mit GitHub zunächst einsteigen und die ersten Schritte gehen. Später nehmen wir noch Git dazu, um die volle Bandbreite der Möglichkeiten ausschöpfen zu lernen. Am Ende wirst du selbst einschätzen können, wann du was am besten einsetzen kannst. GitLab und Bitbucket werden nicht Teil dieses Buchs sein, sollten aber mit den in diesem Buch vermittelten Grundlagen ebenfalls relativ leicht erlernbar sein. GitLab nutzt beispielsweise zum Teil dasselbe Vokabular wie GitHub.

## Wie ich mein erstes fremdes Projekt unterstützt habe

Obwohl ich programmieren kann, habe ich zu meinem ersten fremden Projekt keinen Programmcode beigetragen. Ich möchte diese Geschichte erzählen, um auch Menschen ohne Programmierkenntnisse zu ermutigen, bei Open-Source-Projekten mitzuwirken.

Ich habe ein Open-Source-Spiel auf meinem Smartphone gespielt, bei dem mir die schlechte deutsche Übersetzung aufgefallen war. Manches wirkte zusammengewürfelt, und an vielen Stellen war das schönste »Denglisch«<sup>11</sup> zu lesen.

Ich stellte fest, dass der Autor sein Spiel auf GitHub weiterentwickelte, und begann damit, eine – in meinen Augen bessere – deutsche Übersetzung in kleineren Häppchen beizusteuern. Da der Autor sehr schnell auf meine Änderungsvorschläge reagierte, habe ich mich ermutigt gefühlt, weiterzumachen, bis ich das ganze Spiel einmal »generalüberholt« hatte. Das Gefühl, als ich dann das erste Mal »meiner« Übersetzung beim Spielen des Spiels begegnete, war unbeschreiblich!

## Mit welchen Kosten muss ich rechnen?

GitHub kannst du grundsätzlich erst einmal kostenlos verwenden, ebenso wie das in diesem Buch vorgestellte Git. GitHub bietet allerdings auch kostenpflichtige Funktionen und Erweiterungen an. In diesem Buch werden wir uns aber ausschließlich mit den kostenfreien Features beschäftigen. Zum Zeitpunkt der Drucklegung dieses Buchs waren das unter anderem:

- Unbegrenzte Anzahl öffentlicher Repositories.
- Unbegrenzte Anzahl privater Repositories.
- Unbegrenzte Anzahl an Mitarbeitenden.
- 2.000 Action-Minuten pro Monat für öffentliche Repositories.
- Issues und Fehler-Tracking.
- Projektmanagement.

Das bedeutet: Man kann kostenfrei beliebig viele Projekte anlegen, und es können beliebig viele Menschen an diesen Projekten mitarbeiten. Actions erlauben es, das eigene Repository zu automatisieren, und es gibt dafür ein gewisses Freikontingent (das schauen wir uns in Kapitel 9 noch genauer an). Issues, Fehler-Tracking und Projektmanagement sind Werkzeuge zur Unterstützung bei der Projektabwicklung. Auch diese Werkzeuge lernst du noch kennen.

<sup>11</sup> Kunstwort aus »Deutsch« und »Englisch«, denglische Texte sind ein Mischmasch dieser beiden Sprachen und finden sich auch in diesem Buch wieder, z. B. bei »Committe bitte ...«

Für die meisten privaten Personen, die gerade ihre ersten Schritte in die Veröffentlichung von Software oder Ähnlichem gehen, ist der kostenlose Zugang völlig ausreichend.

Wer bereit ist, Geld auszugeben, kann beispielsweise das Actions-Kontingent erhöhen, bekommt sieben Tage die Woche rund um die Uhr Unterstützung bei Fragen oder Problemen (24/7-Support) oder kann die Anmeldung bei GitHub über Single Sign-on realisieren (siehe Erklärbärbox »Single Sign-on«). Die GitHub-Website<sup>12</sup> gibt Aufschluss darüber, was gegen Einwurf kleiner Münzen noch an weiteren Features mit welchem Preismodell möglich ist.

## Single Sign-on

*Single Sign-on* (SSO) kann man mit »einmaliger Anmeldung« oder »Einmalanmeldung« übersetzen. Eine Benutzerin meldet sich bei einem einzelnen Dienst an, beispielsweise auf einer Website oder im Firmennetzwerk, und wird dann automatisch auch bei anderen Diensten angemeldet. Dadurch ist es für die Benutzerin nicht mehr notwendig, sich die Anmelddaten für jeden einzelnen Dienst merken zu müssen (siehe auch Abbildung 1-1).

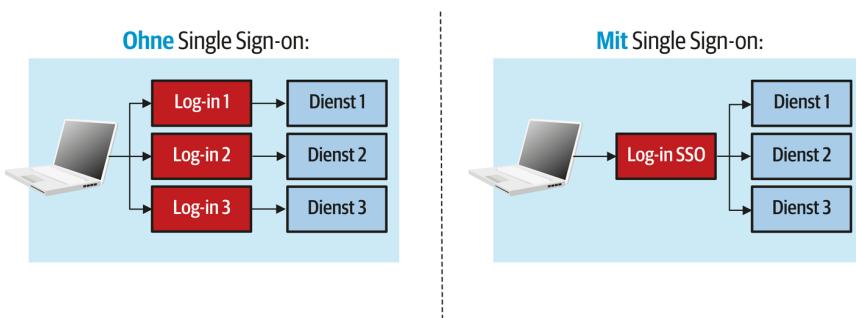


Abbildung 1-1: Mit Single Sign-on (SSO) muss sich eine Benutzerin nur noch die Zugangsdaten für den SSO-Dienst merken.

Vorteile sind unter anderem eine schnellere Anmeldung und ein erhöhter Komfort für die nutzende Person. Zudem sinkt das Risiko, dass Nutzer sich zu schwache oder leicht zu erratene Passwörter auswählen, da sie sich nicht mehr so viele unterschiedliche Passwörter ausdenken müssen (manchmal auch »Passwortmüdigkeit« genannt). Auf der anderen Seite sollte der SSO-Dienst sicherstellen, dass das gewählte SSO-Passwort stark und schwer zu knacken ist. Ein großer Nachteil: Fällt der SSO-Dienst einmal aus, kann man sich auch nicht mehr oder nur noch umständlich an den anderen Diensten anmelden.

12 <https://github.com/pricing>



## KAPITEL 2

# GitHub – Wir verschaffen uns einen Überblick

Die Startseiten vieler GitHub-Projekte wirken leider nicht sonderlich einladend. Man klickt auf einen Link in der Hoffnung, weitere Informationen zu einer bestimmten App zu bekommen, und das Erste, womit man begrüßt wird, ist ein Hinweis, sich einen GitHub-Account zuzulegen, und eine Auflistung von Dateien (siehe Abbildung 2-1).

Selbst für Menschen, die viel mit Dateien und Verzeichnissen arbeiten und programmieren können, wirken die meisten GitHub-Projekte daher auf den ersten Blick nicht besonders einladend. Es ist also völlig normal, wenn auch du nach etwas Herumgeklickte noch nicht schlauer bist.



### Am Ende des Kapitels kannst du ...

- auf GitHub navigieren, und du weißt, wie du an benötigte Informationen kommst.
- Dateien auf GitHub finden und herunterladen.

The screenshot shows the GitHub repository page for 'moby/moby'. At the top, there's a navigation bar with links for 'Why GitHub?', 'Team', 'Enterprise', 'Explore', 'Marketplace', 'Pricing', 'Search', 'Sign in', and 'Sign up'. Below the header, the repository name 'moby / moby' is displayed, along with statistics: 3.1k stars, 58.1k forks, and 16.8k issues. A 'Code' tab is selected, showing a 'Join GitHub today' banner. Below the banner, there are sections for 'About' (describing Moby as a collaborative project for container ecosystems) and a 'Commits' table. The commits table lists recent activity, such as a merge pull request from 'tiborvass' and updates to 'github', 'api', and 'builder' branches. On the right side, there are links to 'mobyproject.org/' and categories like 'docker', 'containers', and 'go'.

Abbildung 2-1: Ein x-beliebiges GitHub-Repository

Wir schauen uns daher zunächst an, was für den Anfang wichtig ist. In Abbildung 2-2 habe ich die Bereiche hervorgehoben, mit denen wir uns primär beschäftigen werden.

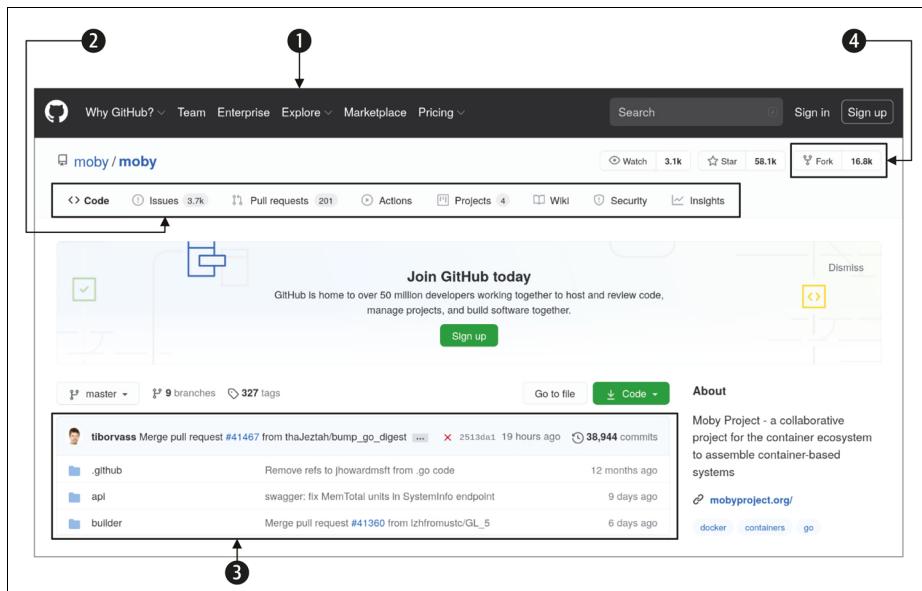


Abbildung 2-2: Die für den Anfang wichtigen Bereiche eines GitHub-Projekts

- **Bereich ①** ist das Menü von GitHub, um auf die Startseite von GitHub zurückzukommen, GitHub zu durchsuchen oder um einen Account anzulegen.
- **Bereich ②** ist das Menü des Repositorys. Hier kannst du zwischen den verschiedenen Funktionalitäten, die GitHub für ein Projekt bereitstellt, hin und her springen.
- **Bereich ③** ist der Arbeitsbereich eines Repositorys. An dieser Stelle wirst du später unter anderem Projektdateien bearbeiten, Fehler im Projekt notieren und mit anderen kommunizieren.
- **Bereich ④** ist der »magische Knopf«, um bei anderen Repositories zu unterstützen.

Wir besprechen zunächst Anwendungsfälle für GitHub und werden dann ein eigenes Projekt bzw. Repository aufbauen. Wir ignorieren aber erst einmal, dass es auch noch andere Repositories und Menschen da draußen gibt. Dafür werden wir überwiegend in den mit ① bis ③ gekennzeichneten Bereichen arbeiten (siehe Kapitel 3).

Dann schauen wir uns an, was es so alles zu tun und zu beachten gibt, wenn man seine Freundinnen zum eigenen Projekt einlädt (siehe Kapitel 4). Auch hier werden wir in den Bereichen ① bis ③ unterwegs sein.

Später, wenn du etwas erfahrener (und mutiger) bist, werden wir zudem schauen, wo und wie wir andere Repositories unterstützen können (siehe Kapitel 6). Dafür werden wir in den Bereichen ❶ bis ❹ arbeiten.

GitHub bietet aber noch viel mehr, z.B. den *Marketplace* oder den Menüpunkt *Explore*, der uns helfen kann, unser Projekt bekannter zu machen. Auch das Zusammenspiel mit Git hält weitere Möglichkeiten bereit. Das schauen wir uns aber erst an, wenn du einigermaßen trittsicher bist und wir die Dinge auch wirklich brauchen (das folgt in Kapitel 6, Kapitel 8 und Kapitel 9).

## Das GitHub-Logo – Mona Lisa Octocat

Vielleicht ist dir schon das Logo von GitHub aufgefallen (etwa in Tabelle 1-1 in Kapitel 1): eine Katze mit Oktopus-Armen, auch als »Octocat« bekannt mit dem Namen »Mona Lisa«.<sup>1</sup> Diese Figur hat im Laufe der Zeit eine solche Beliebtheit unter GitHub-Nutzer\*innen erlangt, dass es mittlerweile Folgendes gibt:

- Ein Blog<sup>2</sup>, auf dem alle offiziell erstellten Octocat-Varianten veröffentlicht werden, das sogenannte »Octodex«.
- Einen Onlineshop<sup>3</sup>, um das Maskottchen auf unterschiedlichsten Materialien zu erwerben, unter anderem auf T-Shirts, Stickern und Tassen.
- Einen eigenen Twitter-Account<sup>4</sup>, auf dem Neuigkeiten rund um GitHub veröffentlicht werden.
- Und eine Applikation im Webbrowser<sup>5</sup>, mit der man sich seine eigene Octocat erstellen kann.

Es gibt sogar kleine Comic-Verfilmungen mit Mona als Hauptfigur,<sup>6</sup> um beispielsweise Werbung für ein Event oder eine Neuerung zu machen. Auch in manchen Tutorials trifft man auf Mona.

1 Die Entstehungsgeschichte kannst du hier nachlesen: <http://cameronmcefey.com/work/the-octocat/>.

2 <https://octodex.github.com/>

3 <https://github.myshopify.com/>

4 <https://twitter.com/monatheoctocat>

5 <https://myoctocat.com/build-your-octocat/>

6 Zu finden im offiziellen GitHub-YouTube-Kanal: <https://www.youtube.com/user/github/>.

# Anwendungsfälle für GitHub (oder: Was will ich da eigentlich?)

Jeder und jede hat unterschiedliche Gründe, sich auf GitHub zu tummeln. Nach meiner Erfahrung gibt es folgende Personengruppen:

1. Die **interessierte Anwenderin** möchte zu einem bestimmten Projekt ein paar weitergehende Informationen haben, z.B. Installationsanleitungen oder wie man mit einem bekannten Fehler der Software umgeht. Sobald sie einen Fehler melden möchte, wechselt sie in die Rolle der Contributorin (siehe weiter unten).
2. Der **Hilfe suchende Programmierer** programmiert gerade an etwas Eigenem (auf GitHub veröffentlicht oder auch nicht)<sup>7</sup> und sucht für ein spezifisches Problem Inspiration und Hilfe. Oder er sucht nur eine Software, die er herunterladen und nutzen möchte.
3. Der **Contributor** (englisch *contribute* = mitwirken, beisteuern) ist Teil eines oder mehrerer (nicht eigener) Projekte (oder möchte es mal werden) und unterstützt mit Code, Dokumentation und/oder möchte einen Fehler melden. Er macht in der Regel Vorschläge für Anpassungen, hat aber keine Rechte, diese auch durchzusetzen.
4. Die **Maintainerin** dagegen kümmert sich bei Projekten um deren Instandhaltung (englisch *maintain* = instand halten, GitHub nennt diese Rolle den *Collaborator*) und verfügt dafür über mehr Rechte als andere. Unter Instandhaltung kann alles Mögliche verstanden werden: die Codebasis aufräumen, Änderungsvorschläge annehmen oder verwerfen, eingehende Fehlermeldungen klassifizieren, das Wiki befüllen oder die entsprechende Website auf dem aktuellen Stand halten.
5. Die **Projekteignerin** hat vollen Zugriff auf ein oder mehrere Projekte. Das bedeutet, sie kann sowohl inhaltliche Änderungen als auch personelle Besetzungen vornehmen, beispielsweise Maintainer (Collaborator) benennen.

In diesem Kapitel schauen wir uns zunächst alle Aktivitäten genauer an, für die du keinen GitHub-Account benötigst. Konkret sind das die Suche nach Informationen und das Herunterladen von Dateien. Sobald du dich aktiv einbringen möchtest, beispielsweise in Form von Kommentaren oder durch das Einreichen von Änderungsvorschlägen, ist das Anlegen eines Accounts notwendig (siehe auch Tabelle 2-1). Die Konzepte, Begriffe und die typischen Arbeitsschritte, die du dafür benötigst, folgen in den restlichen Kapiteln.

---

<sup>7</sup> Auf GitHub gibt es auch Inspiration zu anderen Themen als nur zu Programmierprojekten, diese anderen Themen sind hier ebenfalls gemeint. In Kapitel 11, Abschnitt »Ideen für eigene Repositories – ohne programmieren« auf Seite 259, sind ein paar solcher Repositories aufgelistet.

Tabelle 2-1: Übersicht darüber, welche Aktivitäten mit und ohne GitHub-Account durchführbar sind

Kein Account notwendig	Account notwendig
Informationen finden	kommentieren
Dateien finden	Wünsche äußern
Dateien herunterladen	Fehler melden
	Änderungsvorschläge einreichen
	Projekte veröffentlichen und bearbeiten

## Informationen finden (interessierte Anwenderin)

Eine interessierte Anwenderin möchte sich über ein oder mehrere Projekte informieren. Die Startseite eines Projekts ist der beste Ausgangspunkt dafür. Oben links in der Ecke steht in der Regel der Name des Repositorys und auch der Accountname der Projektleiterin nach dem Muster »Accountname/Repositoryname«. Das Geheimnis, um an mehr Informationen zu kommen, besteht darin, einfach mal runterzuscrollen, und auf einmal zeigt sich eine ganz neue Welt an Informationen (siehe Abbildung 2-3).

Diese Datei wird gerade angezeigt.

Der Rest der Liste der Dateien.

<a href="#">TESTING.md</a>	Bump golang 1.12.8 (CVE-2019-9512, CVE-2019-9514)	6 months ago
<a href="#">VENDORING.md</a>	fix the bare url and the Summary of http://semver.org	3 years ago
<a href="#">codecov.yml</a>	Add code coverage report and codecov config	2 years ago
<a href="#">poule.yml</a>	Poule: remove random assign	5 months ago
<a href="#">vendor.conf</a>	awslogs: Update aws-sdk-go to support IMDSv2	2 days ago

► [README.md](#)

Ab hier wird es interessant.

**The Moby Project**

**moby**  
project

Moby is an open-source project created by Docker to enable and accelerate software containerization.

It provides a "Lego set" of toolkit components, the framework for assembling them into custom container-based systems, and a place for all container enthusiasts and professionals to experiment and exchange ideas. Components include container build tools, a container registry, orchestration tools, a runtime and more, and these can be used as building blocks in conjunction with other tools and projects.

Abbildung 2-3: Nach dem Runterscrollen können die ersehnten Informationen zu einem Projekt auftauchen. Eine besondere Bedeutung hat die Datei README.md. Deren Inhalt wird hier nämlich angezeigt.

## README.md

Bei GitHub gibt es eine Konvention bezüglich des Dateinamens *README.md*. Sobald sich eine solche Datei entweder im Hauptverzeichnis (auch Wurzel- oder Root-Verzeichnis genannt) oder in den Unterordnern *docs* oder *.github* (den Punkt vorweg beachten!) befindet, interpretiert GitHub diese als »Willkommenseite« und zeigt deren Inhalt den Besuchern des Repositorys an.

Idealerweise sollte in der Datei eine Reihe von Fragen zum Projekt beantwortet werden wie:

- Was macht das Projekt, und warum ist das irgendwie nützlich?
- Wie benutzt man das Projekt (z.B. wie installiert und konfiguriert man die Software)?
- Was kann eine Anwenderin machen, wenn sie Hilfe braucht?
- Eventuell: Wie kann man das Projekt unterstützen und in welcher Form?
- Eventuell: Wo findet man weiterführende Informationen?
- Eventuell: Wer entwickelt und betreibt das Projekt?

Die Endung *md* steht übrigens für »Markdown«. Das ist eine Auszeichnungssprache mit simplen Anweisungen, um optische Anpassungen und Formatierungen wie Überschriften, Fettschrift und Aufzählungszeichen zu erzeugen oder um Bilder einzufügen und Links zu setzen. Die Überschrift »The Moby Project« in Abbildung 2-3 kann man beispielsweise wie folgt erzeugen<sup>8</sup>:

```
# The Moby Project
```

Gute Auflistungen gängiger Befehle und wie deren Darstellung aussieht, findest du im Internet.<sup>9</sup> GitHub selbst bietet auf seinen Hilfeseiten ebenfalls weiterführende Informationen an.<sup>10</sup>

Vorlagen für eine gute *README.md* gibt es natürlich auch, beispielsweise unter <https://gist.github.com/jxson/1784669>.

**Tipp:** Man kann übrigens auch in Unterverzeichnissen jeweils eine *README.md* anlegen. In diesem Verzeichnis wird sie dann ebenfalls als Willkommenseite angezeigt.

Jetzt sich kann die interessierte Anwenderin ab sofort glücklich und wissend zurücklehnen.

<sup>8</sup> Für Menschen mit Programmiererfahrung sieht es auf den ersten Blick wie ein Quellcode-Kommentar aus, es erzeugt aber tatsächlich eine Überschrift der ersten Ordnung.

<sup>9</sup> Beispielsweise <https://github.com/tchapi/markdown-cheatsheet/blob/master/README.md>.

<sup>10</sup> <https://help.github.com/en/github/writing-on-github/basic-writing-and-formatting-syntax>

Aber hoppla, was macht die interessierte Anwenderin, wenn das Projekt gar keine *README.md* anbietet oder diese Datei nur spärliche Angaben enthält? Projekte können ein *Wiki* haben, in dem weitere Informationen enthalten sein könnten (zu finden in Zeile 1 in Abbildung 2-2). Nach meiner Erfahrung ist es jedoch eher unwahrscheinlich, dass es ein gut dokumentiertes Wiki gibt, aber keine Willkommensseite. Einen Versuch ist es dennoch allemal wert.



### Tipp für eigene Projekte

Wer möchte, dass sich die Menschen schnell im Projekt zurechtfinden, befüllt die *README.md* entsprechend aussagekräftig.

Keine *README.md*, kein *Wiki* – und nu? Jetzt hängt es davon ab, wie wichtig es der interessierten Anwenderin ist, eine bestimmte Information zu erhalten. Will sie sich z.B. über einen Fehler informieren, lohnt sich ein Blick in das Register *Issues*, zu Deutsch »Thema«, »Frage« oder auch »Angelegenheit« (siehe Abbildung 2-4, die Zahl zeigt die derzeit offenen Issues an). Ein *Issue* ist eine Art offener Brief an die Projektleiterin oder deren Maintainer und soll Handlungsbedarf aufzeigen. Der Handlungsbedarf kann sich auf Fehler beziehen, aber auch Wünsche für weitere Funktionen (sogenannte *Feature-Requests*) oder Anmerkungen darüber, wie beispielsweise eine fehlende Willkommensseite, sind üblich. Issues können kommentiert werden, und es kommt dort häufig zu regen Diskussionen. Mit etwas Glück findet die interessierte Anwenderin hier die Lösung für ihr Problem. Leider gibt es dafür keine Garantie, aber immerhin die Chance.

Abbildung 2-5 zeigt die *Issues*-Seite, auf der die interessierte Anwenderin suchen, filtern, Fehler melden oder sich Issues anschauen kann. Man beachte das vorausgefüllte Textfeld, in dem *is:issue is:open* steht. Das ist das Suchfeld. Da sie auf das Register *Issues* geklickt hat, nimmt GitHub an, dass sie an Issues interessiert ist (*is:issue*) und dass mit hoher Wahrscheinlichkeit die noch offenen für sie im Moment interessant sind (*is:open*). Diese beiden vorausgefüllten Filter können jederzeit gelöscht und/oder um eigene Suchwörter ergänzt werden. Um Antworten auf eine Frage zu bekommen, empfiehlt es sich, nach Issues Ausschau zu halten, die entsprechende Suchwörter enthalten und in denen rege diskutiert wird. Ob in dem Issue diskutiert wird, erkennt man an einem Sprechblasensymbol auf der rechten Seite. Die dort aufgeführte Zahl ist die Anzahl der Kommentare.



Abbildung 2-4: Im Register *Issues* besteht eine gute Chance, hilfreiche Informationen zu finden.

Filter- und Suchmöglichkeit

Weitere Filtermöglichkeiten

Fehler melden

Pinned issues

[ANNOUNCEMENT] Shutting down dockerproject.org APT and YUM re...

#40466 opened 2 days ago by chris-crone

① Open

is:issue is:open

Labels 126 Milestones 3 New issue

① 3,617 Open ✓ 16,329 Closed Author ▾ Label ▾ Projects ▾ Milestones ▾ Assignee ▾ Sort ▾

① Cannot run process isolated containers anymore after updating to Windows 1909 ←  
#40468 opened 2 days ago by dtauch

① [ANNOUNCEMENT] Shutting down dockerproject.org APT and YUM repos 2020-03-31  
area/packaging

#40466 opened 2 days ago by chris-crone

① docker inspect containerID blocks status/more-info-needed version/18.06  
#40463 opened 2 days ago by qihui

Hier gibt es noch eine überschaubare Diskussion

Einer von vielen Issues

Abbildung 2-5: Bei den Issues gibt es vielfältige Filter- und Suchmöglichkeiten. Auch sieht man auf einen Blick, bei welchem Issue viel diskutiert wurde bzw. wird.

Sollte die interessierte Anwenderin hier immer noch nichts finden, hilft häufig nur noch eine Webrecherche<sup>11</sup>, um außerhalb von GitHub hilfreiche Informationen zu finden. Es gibt viele Foren im Internet, und mit etwas Glück unterhalten sich andere Menschen dort genau über das gesuchte Thema.



### Tipp: Forum Stackoverflow

Meine Lieblingsseite für Fragen über Software im Allgemeinen und Softwarecode im Speziellen ist Stackoverflow<sup>12</sup>. Es gibt dort viele kompetente Menschen, die ihr Wissen gern teilen.

Der Clou ist in meinen Augen aber das ausgefeilte Bewertungssystem, das dafür sorgt, dass die am besten bewertete Antwort auf eine Frage ganz oben angezeigt wird. Nie wieder langes Scrollen durch wenig hilfreiche Beiträge, das Wichtigste steht immer oben.

Im schlimmsten Fall gibt es aber keine Informationen, und die interessierte Anwenderin muss auf Alternativen ausweichen. Beispielsweise kann sie versuchen,

11 Ich empfehle dafür immer datenschutzfreundliche Suchmaschinen wie beispielsweise <https://duckduckgo.com/> oder <https://www.startpage.com/>.

12 <https://stackoverflow.com/questions>

mit einem Softwarefehler zu leben, oder sie sucht sich eine andere Software, die einen ähnlichen Funktionsumfang hat.

## Dateien finden und herunterladen (Hilfe suchender Programmierer)

Ein Hilfe suchender Programmierer wird in der Regel installierbare Softwarepakete oder Beispielprogrammcode suchen. Er hat die Möglichkeit, auf der Startseite eines Projekts nach Dateien zu suchen und sie herunterzuladen. Diese beiden Funktionen befinden sich direkt nebeneinander (siehe Abbildung 2-6). Zu beachten ist, dass er sich über die Startseite nur das ganze Projekt als Zip-File herunterladen kann und keine einzelnen Dateien. Es ist ebenfalls möglich, das Repository ganz ohne die GitHub-Oberfläche mit Git auf den heimischen Rechner zu holen (das nennt man »klonen«), dazu aber später in Kapitel 8 mehr.

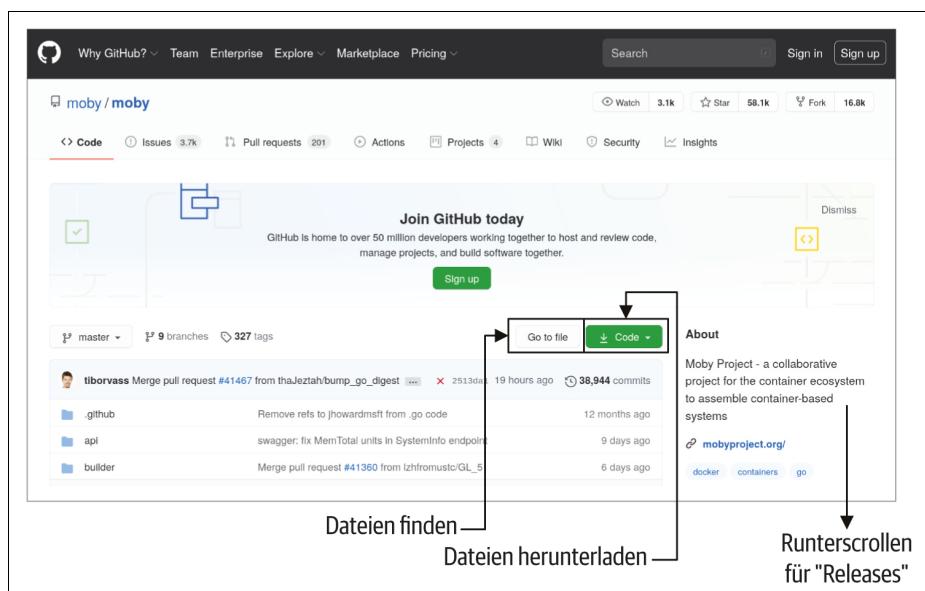


Abbildung 2-6: Um Dateien zu finden oder herunterzuladen, können diese beiden schmucken Buttons verwendet werden.

Damit lädt er sich den aktuellen Arbeitsstand des Projekts herunter, der unter Umständen noch fehlerbehaftet ist. Sucht er eine getestete und lauffähige Version, wird er unter *Releases* fündig (auf der Startseite eines Projekts am rechten Rand). Dort sind die vom Projektteam freigegebenen Versionen des Projekts zu finden und herunterzuladen.

Es ist auch möglich, einzelne Dateien herunterzuladen, z.B. indem man die Datei in GitHub öffnet und *Raw* auswählt (siehe Abbildung 2-7). GitHub zeigt dann die Datei in ihrem »rohen« Zustand an, beispielsweise sieht man in einer *README.md*

Formatierungsbefehle wie # und Ähnliches. Per Rechtsklick im Browser ist es dann möglich, mit einem *Seite speichern unter...* oder Ähnlichem (abhängig vom eingesetzten Browser) die Datei herunterzuladen.



Abbildung 2-7: Um einzelne Dateien herunterzuladen, ist der Button »Raw« nützlich.



#### Tipp: Zugriff auf Rohdaten über die Webadresse

Die rohen (Raw-)Daten einer Datei kann man – neben dem erwähnten Button – auch durch das Anhängen von `?raw=true` an die Webadresse der entsprechenden Datei erhalten, z.B. <https://github.com/moby/moby/blob/master/README.md?raw=true>.

Die Navigation innerhalb der Dateiliste erfolgt ähnlich wie in den Dateiverzeichnissen der gängigen Betriebssysteme. Wie auf dem eigenen Computer sind auch hier die Dateien in Ordnern strukturiert, was die entsprechenden Symbole anzeigen.

Für den Hilfe suchenden Programmierer lohnt sich ebenfalls ein Blick in die *Issues*, da hier häufig Verbesserungen des Codes diskutiert werden. Auch hier gilt: Je mehr rege Diskussionen zu einem Issue, desto größer ist die Wahrscheinlichkeit, dass etwas Hilfreiches dabei ist.

## KAPITEL 3

# Die Basis: Das erste eigene GitHub-Projekt

Was brauchst du alles für den Einstieg mit GitHub? Einen funktionstüchtigen Rechner mit einem der Betriebssysteme Windows, Linux oder macOS, darüber hinaus eine stabile Internetanbindung und einen Internetbrowser. Ansonsten solltest du Folgendes tun:

- Dir einen kostenlosen GitHub-Account zulegen (wie, beschreibe ich im folgenden Abschnitt »Account anlegen« auf Seite 20).
- Falls du mit Git arbeiten willst: Git auf deinem Rechner installieren (eine entsprechende Anleitung findest du in Kapitel 7, Abschnitt »Git installieren und einrichten« auf Seite 141).

Solltest du bereits einen Account bei GitHub eingerichtet haben, kannst du dieses Kapitel gegebenenfalls überspringen. Vielleicht schaust du aber trotzdem kurz in die Abschnitte »Account schützen« auf Seite 22 und »Unsichtbar werden – die eigene Mailadresse schützen« auf Seite 23, falls du dich mit dem Schutz deines Accounts oder deiner Mailadresse noch nicht befasst hast.



### Am Ende des Kapitels kannst du ...

- auf GitHub einen Account anlegen und diesen sowie deine Mailadresse schützen.
- Repositories anlegen und Änderungen an diesen vornehmen.
- Issues anlegen, klassifizieren, jemandem zuweisen und schließen.
- ein Repository löschen.
- ein lokales Projekt auf GitHub hochladen.

Falls du dir noch nicht sicher bist, ob du Git nutzen möchtest, kannst du die Installation auch später nachholen, sobald du im entsprechenden Kapitel angekommen bist. Für große Teile des Buchs werden wir Git nicht brauchen. Und wer weiß, vielleicht reichen dir ja auch die Funktionen von GitHub?

# Account anlegen

Einen Account bei GitHub einzurichten, ist relativ einfach. Dafür klickst du im Menü von GitHub<sup>1</sup> auf den Button *Sign up* (deutsch »sich registrieren«) und füllst die geforderten Informationen entsprechend aus (siehe auch Abbildung 3-1).

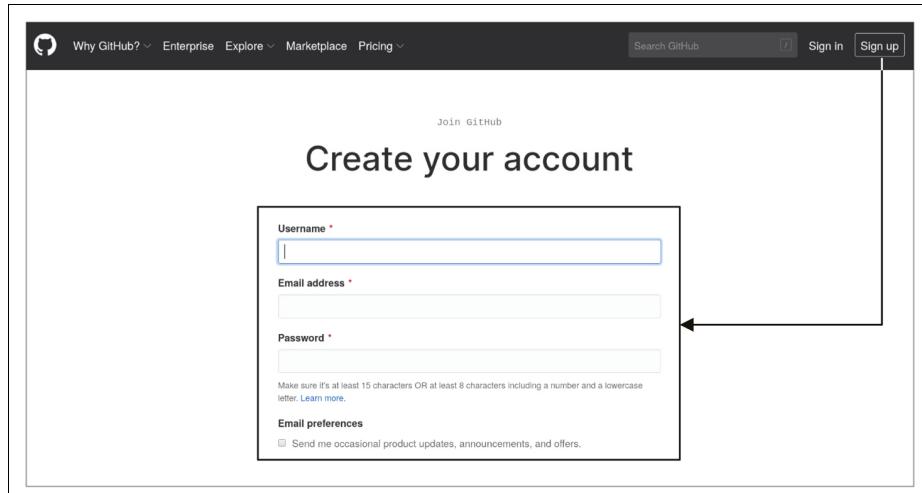


Abbildung 3-1: Sich bei GitHub zu registrieren, ist einfach: Username wählen, eigene Mailadresse eingeben und ein Passwort vergeben – fertig.

Sobald du eingeloggt bist, ändert sich das GitHub-Menü etwas (siehe Abbildung 3-2).

- Über ❶ kommst du immer zurück auf die Startseite von GitHub.
- In ❷ befindet sich die Suchfunktion, mit der du nach Repositories, aber auch innerhalb eines Repositorys suchen kannst.
- Die Punkte unter ❸ sind zum einen Schnellzugriffe auf die selbst erstellten *Pull requests* und *Issues* (die du in diesem und Kapitel 4 noch näher kennenlernenst). Der Punkt *Marketplace* ist für das Erweitern des eigenen Repositorys gedacht (siehe Kapitel 9), und *Explore* hilft uns beim Finden anderer Projekte (siehe Kapitel 6).
- Bei ❹ steht der Schnellzugriff bereit, um beispielsweise ein neues Repository anzulegen.
- In ❺ findest du alles Relevante für deinen Account, z.B. deine Repositories oder die Möglichkeit, dich auszuloggen.

<sup>1</sup> <https://github.com>



Abbildung 3-2: Nach dem Log-in verändert sich das GitHub-Menü etwas.

## Standardprofilbild – Identicon

Vielleicht ist dir nach einem Log-in in GitHub ein merkwürdiges kleines pixeliges Bild oben rechts auf der Seite aufgefallen (in Abbildung 3-2 bei Punkt 5), das so ähnlich aussieht wie eines der drei in Abbildung 3-3. Das ist ein sogenanntes *Identicon* – ein Kofferwort<sup>2</sup> aus »Identification« und »Icon« (zu Deutsch »Identifikation« und »Symbol«) –, und du siehst hier dein Profilbild.



Abbildung 3-3: Identicons sind das Standardprofilbild bei GitHub.  
(Bildquelle: <https://github.blog/2013-08-14-identicons>)

GitHub generiert diese Bilder automatisch in Abhängigkeit von deiner Benutzer-ID. Es gibt also keine Möglichkeit, die Farbe oder das Muster anzupassen. Du kannst das Bild aber jederzeit durch ein eigenes ersetzen (und auch wieder zum Identicon zurückkehren).



### Tipp: Passwortsafe

An dieser Stelle möchte ich für den Einsatz eines Passwortsafes werben, der nicht nur alle Accounts und Passwörter sicher aufbewahrt, sondern häufig auch einen Passwortgenerator für sichere Passwörter mitbringt.

Ich persönlich nutze KeePassXC<sup>3</sup>, das viele Erweiterungsmöglichkeiten bietet. Beispielsweise gibt es für den Browser eine Erweiterung, mit der ich Zugangsdaten aus dem Safe direkt in die Eingabemaske einer Webseite »beamten« kann. Und mit der Smartphone-App *Keepass2Android* habe ich meine Zugangsdaten auch mobil verfügbar, falls ich sie brauche.

2 Verschmelzung zweier Wörter zu einer neuen Bedeutung, z.B. breakfast + lunch = brunch.

3 <https://keepassxc.org/>

# Account schützen

Eine Sache, die du einrichten kannst, um die Sicherheit deines Accounts zu erhöhen, ist die 2-Faktor-Authentifizierung. Du findest sie in den *Settings* deines Profils unter dem Punkt *Account security* (siehe auch Abbildung 3-4). Folge den Anweisungen dort, um diese Form der Authentifizierung einzurichten. Du hast die Möglichkeit, entweder eine Authenticator-App als zweiten Faktor zu wählen oder ein Verfahren über SMS. GitHub empfiehlt als Authenticator-App für das Smartphone *Authy*<sup>4</sup>, *1Password*<sup>5</sup> oder *LastPass Authenticator*<sup>6</sup>. Jedes Mal, wenn du dich einloggst, wird GitHub dich auffordern, entweder via App oder SMS – je nachdem, was du eingerichtet hast – einen Code einzugeben, bevor du auf deinen Account zugreifen kannst.



## Wiederherstellungsoptionen bei der 2-Faktor-Authentifizierung

Sobald die 2-Faktor-Authentifizierung eingerichtet ist, solltest du auf jeden Fall eine der von GitHub angebotenen Wiederherstellungsoptionen (*Recovery Options*) einrichten. Falls dein Smartphone verloren geht, hast du über diese Optionen die Möglichkeit, wieder Zugriff auf deinen GitHub-Account zu bekommen. GitHub bietet als Optionen an:

- **Recovery codes:** Eine Reihe an Codes, die bei Verlust des Smartphones zur Wiederherstellung genutzt werden können (am besten in einem Passwortsafe abspeichern).
- **Fallback SMS number:** Eine alternative Rufnummer, an die GitHub im Fall des Smartphone-Verlusts die Recovery Codes schickt.
- **Recovery tokens:** Die Möglichkeit, über Facebook wieder Zugriff auf den GitHub-Account zu bekommen.

The screenshot shows the GitHub profile settings interface. At the top, there's a navigation bar with links for 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. On the left, a sidebar lists various settings categories: Profile, Account, Appearance, Account security (which is highlighted with a red box), Billing & plans, Security log, Security & analysis, Emails, Notifications, and Scheduled reminders. The main content area has two sections: 'Change password' and 'Two-factor authentication'. The 'Change password' section contains fields for 'Old password', 'New password', and 'Confirm new password', along with a note about character requirements and buttons for 'Update password' and 'I forgot my password'. The 'Two-factor authentication' section is partially visible below. On the right side of the page, there's a sidebar with user information ('Signed in as githubbuch') and links to 'Your profile', 'Your repositories', etc., as well as buttons for 'Settings' and 'Sign out'.

Abbildung 3-4: Im Profil kommst du über »Settings« zu den relevanten Einstellungsmöglichkeiten.

4 <https://authy.com/guides/github/>

5 <https://support.1password.com/one-time-passwords/>

6 <https://support.logmeininc.com/lastpass/help/lastpass-authenticator-lp030014>

## 2-Faktor-Authentifizierung

Die 2-Faktor-Authentifizierung – auch 2FA genannt – ist eine zusätzliche Sicherungsebene, um einen Account vor unbefugtem Zugriff zu schützen. Wenn ich Zugriff auf meinen Account haben will, benötige ich mindestens zwei verschiedene »Faktoren«, um mich überhaupt einzuloggen zu können. Faktoren können sein:

- Etwas, das ich kenne: Passwort, PIN-Code, spezielles Muster etc.
- Etwas, das ich habe: Smartphone, Kreditkarte, Hardwaretokens etc.
- Etwas, das ich bin: Fingerabdruck, Irisscan, Stimmerkennung etc.

Sofern ein Faktor abhandenkommt – beispielsweise wenn eine fremde Person mein Passwort herausgefunden hat –, erhält sie trotzdem noch keinen Zugriff auf meinen Account, solange sie nicht auch den zweiten Faktor kennt.

Manchmal liest man auch den Begriff »MFA«, was für *Multi-Faktor-Authentifizierung* steht. Das bedeutet, dass ich mehr als einen Faktor nutze. 2FA ist also eine Untermenge von MFA.

Unter dem Menüpunkt *Security log* werden alle für die Sicherheit deines Accounts relevanten Ereignisse gespeichert, beispielsweise wann du dich eingeloggt hast, von wo aus und mit welcher IP-Adresse<sup>7</sup>. Das *Security log* kannst du ebenfalls nutzen, um dir andere Ereignisse – nicht nur solche, die deinen Account betreffen – anzeigen zu lassen. Gibst du *operation:create* in das vorhandene Suchfeld ein, werden alle Ereignisse angezeigt, an denen du etwas Neues erstellt hast, etwa ein neues Repository. Mit *created:2020-02-18* kannst du dir alles anzeigen lassen, was du am 18. Februar 2020 erstellt hast.<sup>8</sup>

## Unsichtbar werden – die eigene Mailadresse schützen

Eine Sache, die du beachten solltest, betrifft deine Mailadresse, die du beim Einrichten deines Accounts angibst. Falls du es nicht explizit änderst, wird deine Mailadresse auf deiner Profilseite angezeigt und auch für Aktivitäten innerhalb von GitHub verwendet.<sup>9</sup> Manche möchten das nicht, deswegen bietet GitHub dir die Möglichkeit, etwas anonymer unterwegs zu sein. In deinem Profil im Menü *Settings* unter dem Punkt *Emails* kannst du festlegen, dass GitHub deine Mailadresse privat behandelt (siehe Abbildung 3-4 und Abbildung 3-5).

7 Das ist die »Postanschrift« deines Rechners.

8 Eine ausführliche Auflistung dessen, was noch alles geht, findest du unter <https://help.github.com/en/github/authenticating-to-github/reviewing-your-security-log>.

9 Sogenannte webbasierte Git-Operationen, das wird verständlicher, sobald wir uns mit Git in Kapitel 7 beschäftigen.

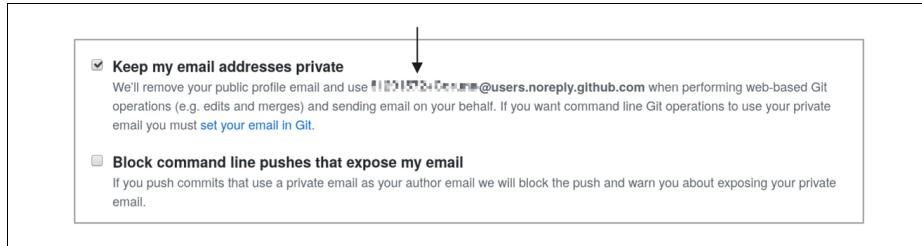


Abbildung 3-5: GitHub bietet die Möglichkeit, deine Mailadresse geheim zu halten – meine anonyme Mailadresse habe ich hier noch einmal zusätzlich durch Verpixeln anonymisiert.<sup>10</sup>

Wie du auf dem Bild sehen kannst, bekommst du eine Art »Alternativ-E-Mail-Adresse« angezeigt, die aus einer siebenstelligen Identifikationsnummer (ID) und deinem Usernamen erzeugt wird: `ID+username@users.noreply.github.com` (beispielsweise `1234567+geheimniskraemer@users.noreply.github.com`). Und wofür brauche ich diese Mailadresse? In »Git installieren und einrichten« auf Seite 141 zeige ich dir später einen Anwendungsfall, im Moment nehmen wir erst einmal hin, dass es diese anonyme Mailadresse gibt.

Des Weiteren siehst du auf dem Bild, dass du auch die Option hast, etwas zu blockieren. Diese Einstellung ist nur relevant, wenn du vorhast, mit Git später zu arbeiten.

Man kann bei GitHub auch noch eine Menge mehr einstellen, wie beispielsweise ein Profilbild hochladen oder Ähnliches. Da uns das aber nicht dabei hilft, die grundsätzlichen Funktionen von GitHub zu verstehen, gehe ich hier nicht näher darauf ein. Aber natürlich kannst du dir die weiteren Einstellungen investigativ erschließen.

## Das erste eigene Repository anlegen

Da du jetzt startklar bist, werden wir ab diesem Abschnitt ein erstes eigenes Projekt anlegen und Veränderungen an den dort vorhandenen Dateien vornehmen. Wir lernen *Commit* und *Issues* genauer kennen, die die Basis für viele andere Aktivitäten bilden.

Für das Anlegen eines eigenen Projekts gibt es mehrere Möglichkeiten. Wir wenden uns mit dem Plussymbol oben rechts in der Ecke im GitHub-Menü zunächst der offensichtlichsten zu (siehe auch Abbildung 3-6) und wählen *New repository*.

<sup>10</sup> Immer wenn du in den Screenshots irgendwelche verwischten Spuren siehst, ist das mein Accountname. Ich habe für die Screenshots in diesem Buch den Account so zugemüllt, dass er mir ein wenig peinlich ist und ich ihn hier daher nicht präsentieren möchte :).

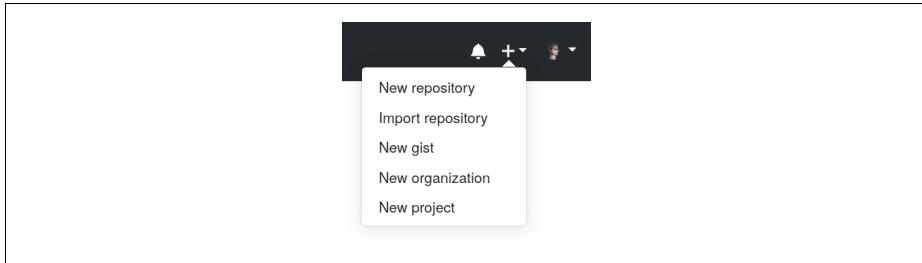


Abbildung 3-6: Ein neues Repository kann man unter anderem über das Plussymbol oben rechts erzeugen.



### Tipp: Shortcut für neue Repositories

Eine weitere Möglichkeit, Repositories anzulegen, bietet die Webadresse <https://repo.new> im Browser. Dann bist du direkt auf der GitHub-Seite für neue Repositories. Das funktioniert natürlich nur, wenn du bereits eingeloggt bist, ansonsten landest du erst mal auf der Log-in-Seite.

Nun werden wir aufgefordert, eine Reihe von Angaben zu machen (siehe Abbildung 3-7). Gib deinem Projekt einen sprechenden Namen – das kann auch so etwas Originelles wie »testprojekt« sein – und, wenn du magst, eine entsprechende Beschreibung. Alles lässt sich später noch ändern.

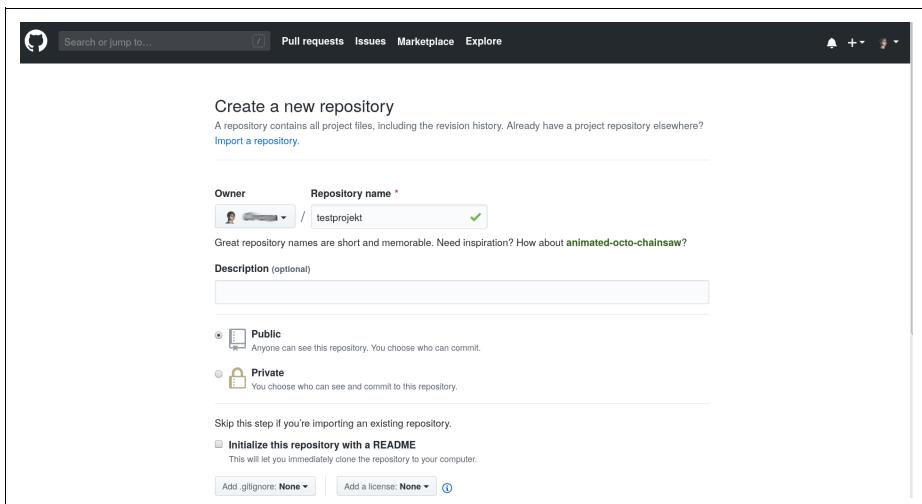


Abbildung 3-7: Ein neues Projekt/Repository ist mit wenigen Klicks erstellt.

Beim Erstellungsprozess wirst du auch gefragt, ob du ein öffentliches (*Public*) oder ein privates (*Private*) Repository anlegen möchtest. Öffentliche Repositories können später von jedem beliebigen Besucher angesehen, aber nur mit Erlaubnis verändert werden. Auf private Repositories hast dagegen ausschließlich du Zugriff. Wenn du für ein Repository einen *Collaborator* festlegst, kann dieser Veränderun-

gen vornehmen und natürlich auch alles sehen, selbst wenn das Repo als privat eingestellt ist. Wir sind hier mal mutig und wählen *Public* aus.



#### Tipp: Test-Repositories anlegen

Wenn du dich erst einmal nur mit GitHub und dessen Funktionen vertraut machen möchtest, empfiehlt es sich, ein oder mehrere Repositories zum Testen und Rumspielen anzulegen. Diese kannst du nach deinen Experimenten wieder löschen (siehe Abschnitt »Ein bestehendes Repository löschen« auf Seite 39 weiter unten in diesem Kapitel).

Zudem wirst du gefragt, ob GitHub für dich eine *README.md* anlegen soll (wir erinnern uns an Kapitel 2, Abschnitt »Informationen finden (interessierte Anwenderin)« auf Seite 13). Da das generell eine gute Idee ist, machen wir das natürlich. Alles andere kannst du erst einmal so lassen, wie es ist. Sobald du auf *Create Repository* geklickt hast, erstellt GitHub automatisch ein Repository mit einer noch recht leeren *README.md* und allen Möglichkeiten, die GitHub so bietet – beispielsweise dem Anlegen von Issues.

**Wichtig zu wissen:** Wir werden im Verlauf des Buchs ausschließlich auf public Repositories arbeiten. Sollte irgendeine Funktion, die ich dir vorstelle, nicht funktionieren, könnte es eventuell daran liegen, dass dein Repository privat ist. GitHub stellt viele Features für öffentliche Repos kostenlos zur Verfügung. Will man die gleichen Features in privaten Repos nutzen, funktioniert das nur nach Einwurf kleiner Münzen.

## Eine inhaltliche Änderung am Projekt vornehmen

Als Erstes befüllen wir die *README.md*. Gehe dazu auf *<> Code*, falls du nicht schon dort bist. Auf der rechten Seite des Screenshots siehst du ein Stiftsymbol (siehe Abbildung 3-8). Das Symbol zeigt generell an, dass Dateien editiert werden können. Über das Stiftsymbol gelangst du in einen Texteditor, in dem du die *README.md* mit Inhalt füllen kannst.

Wie du in Abbildung 3-9 sehen kannst, bietet der Texteditor zwei Optionen an: *Edit file* (Datei editieren) und *Preview changes* (die Vorschau der Änderungen). Standardmäßig bist du im Modus *Edit file*, in dem du deine Änderungen machen kannst. Der Modus *Preview changes* bietet dir eine Vorschau der aktuellen Änderungen an, ohne dass du sie abspeichern musst. Dann lass uns die Datei mal mit Leben füllen. Wie bereits erwähnt, kannst du für die Formatierung der Datei das sogenannte *Markdown* verwenden. Du könnest zum Beispiel Folgendes eintippen:<sup>11</sup>

---

<sup>11</sup> Nur der Vollständigkeit halber: Hierbei handelt es sich um »GitHub Flavored Markdown«, also Markdown, das durch GitHub etwas angereichert wurde. Beispielsweise gehört die Aufgabenliste dazu. Weitere Details findest du hier: <https://guides.github.com/features/mastering-markdown/>.

# Mein wunderbares Projekt

Dies hier ist mein \*\*erstes Projekt\*\*, um \*GitHub\* auszuprobieren.

## Meine To-dos:

- [x] README.md befüllen
- [ ] Andere Dinge ...

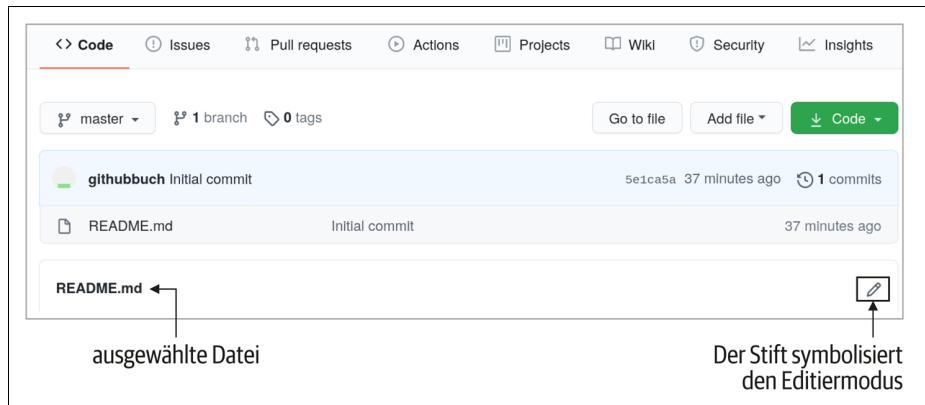


Abbildung 3-8: Durch Anwählen des Stiftsymbols kann man die Datei in einem Editor bearbeiten.

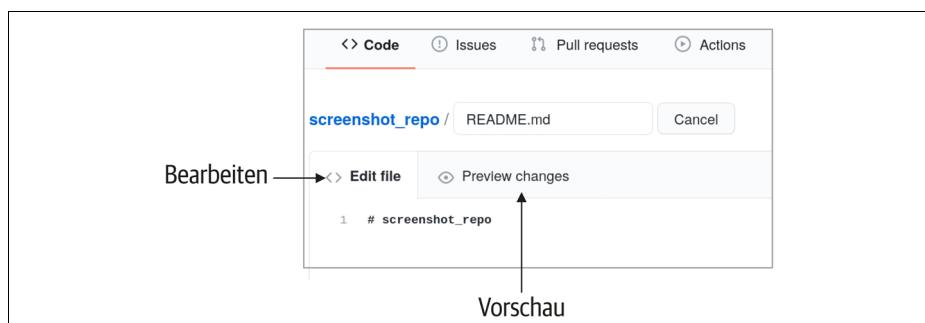


Abbildung 3-9: GitHub bietet einen Texteditor mit den Modi »Bearbeiten« und »Vorschau« an.

Spiel ruhig ein bisschen damit herum, Anregungen findest du in der Erklärbärbox »README.md« in Kapitel 2.

Über die Vorschaufunktion kannst du jederzeit überprüfen, ob das Ganze so aussieht, wie du es dir vorgestellt hast. Wenn du zufrieden bist mit deinem Werk, möchtest du es natürlich abspeichern. Die Möglichkeit dazu findest du, wenn du nach unten scrollst (siehe Abbildung 3-10). Aber hoppla, da steht ja gar nicht speichern, da steht irgendwas von »Commit« und »Branch«. Wir merken uns erst einmal: *Commit* ist so etwas wie *speichern* und fügt die Änderungen zum Projekt hinzu. Später, wenn wir mit Git arbeiten werden (siehe Kapitel 7), werden wir

sehen, dass das nicht ganz korrekt ist. Für den Augenblick reicht es aber, die beiden Begriffe gleichzusetzen.

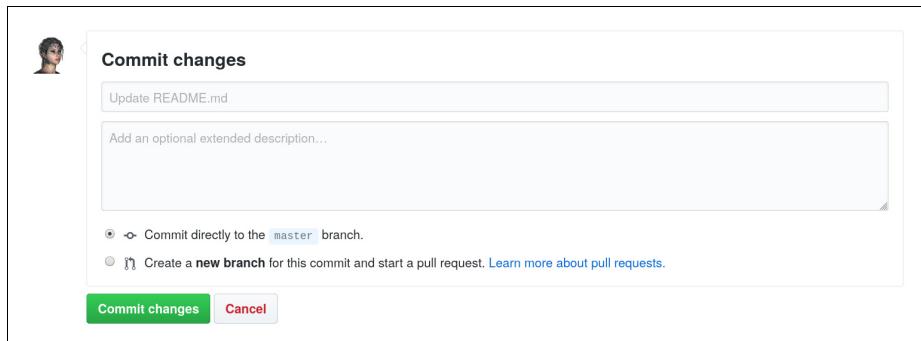


Abbildung 3-10: Abspeichern von Änderungen nennt man bei GitHub »Commit«.

Was den Branch anbelangt, lassen wir zunächst die Standardeinstellung stehen (*Commit directly to the master branch.*, zu Deutsch etwa »direkt auf dem Hauptzweig speichern«). Wir werden später sehen, was es damit auf sich hat (siehe Kapitel 4). Was du noch befüllen solltest, ist die Commit-Nachricht (die Textfelder). Netterweise macht uns GitHub bereits einen Vorschlag für den Titel der Commit-Nachricht, den du erst einmal so übernehmen kannst.

## Commit-Nachrichten

Commit-Nachrichten bei GitHub bestehen aus zwei Teilen: dem Titel und einer (optionalen) detaillierteren Beschreibung. Den Titel solltest du immer aussagekräftig befüllen, damit jede und jeder (und auch du in ein paar Monaten) weiß, was du mit dem Commit bewirken wolltest. GitHub selbst gibt dazu folgende Empfehlungen, ergänzt um meine Tipps:

- Der Commit-Titel sollte nicht mit einem Punkt aufhören, da es sich um eine Überschrift handelt.
- Der Titel sollte nicht aus mehr als 50 Zeichen bestehen (Leerzeichen mitgezählt), siehe Abbildung 3-11.<sup>12</sup> Für detailliertere Informationen dient darunter das Feld *extended description* (»erweiterte Beschreibung«).
- Benutze aktive Sprache, keine passive, also »füge hinzu« anstatt »hinzugefügt« (im Englischen »add« anstatt »added«). Am einfachsten ist es, wenn man sich den Commit-Titel wie einen Befehl ans System denkt: »Ändere dies!«
- Versuche, mit dem Titel deine Absicht auszudrücken.
- Versuche, das Was und Warum zu beschreiben (z.B.: »Restrukturiere Modul A für bessere Lesbarkeit«).

<sup>12</sup> Niemand reißt einem den Kopf ab, wenn es 51 Zeichen sind, es geht eher um das Prinzip »fasse dich kurz und präzise«.

- Schreibe die Nachricht so, dass auch du sie in fünf Jahren noch verstehen kannst.

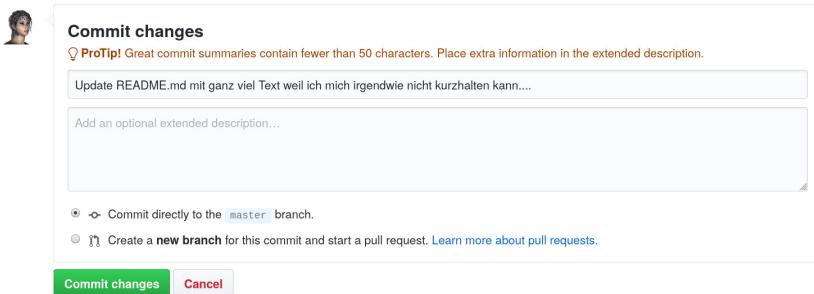


Abbildung 3-11: Wer mehr als 50 Zeichen eintippt, bekommt von GitHub einen entsprechenden Hinweis.

Ein schönes Zitat, das gut verdeutlicht, wieso präzise und klare Commit-Nachrichten so wichtig sind, ist Folgendes:

Jedes Softwareprojekt ist ein gemeinschaftliches Projekt. Es hat mindestens zwei Entwickler, den ursprünglichen Entwickler und den ursprünglichen Entwickler ein paar Wochen oder Monate später, wenn der Gedankengang längst den Bahnhof verlassen hat.

– Who-T, On commit messages, unter <https://who-t.blogspot.com/2009/12/on-commit-messages.html> (Übersetzung von deepl.com)

Nachdem du alle Änderungen dem Projekt hinzugefügt (bzw. committet) hast, gehe auf die Startseite des Projekts und bewundere das neue Erscheinungsbild. Genieße diesen Augenblick! Nichts ist so toll wie der erste Commit, der sofort auch optisch etwas verändert. Wenn du die *README.md* erneut anklickst, bekommst du noch ein paar Zusatzinformationen, wie beispielsweise die Anzahl der Mitwirkenden oder die Dateigröße (siehe Abbildung 3-12).

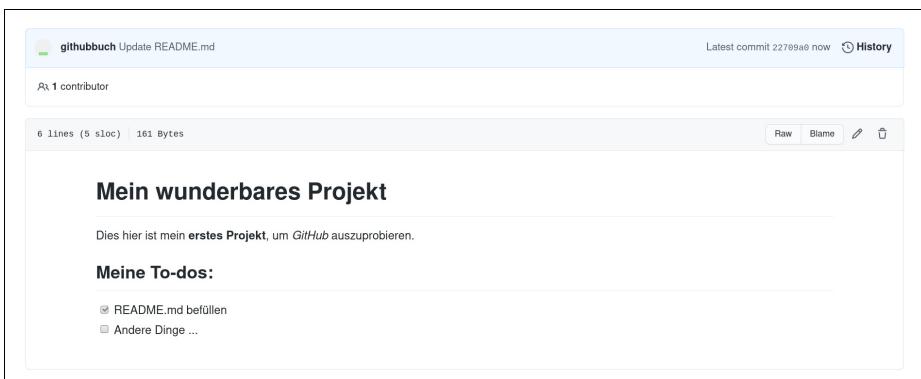


Abbildung 3-12: Durch Anklicken einer Datei erhält man weitere Zusatzinformationen.

## SLOC

Vielleicht ist dir in Abbildung 3-12 die Zeile oben links aufgefallen:

6 lines (5 sloc) | 161 Bytes

Während die Angaben *6 lines* (Anzahl der Zeilen) und *161 Bytes* (Größe der Datei) vermutlich noch relativ eingängig sind, erscheinen die *5 sloc* doch erst einmal ominös, oder? SLOC ausgeschrieben bedeutet *Source Lines of Code* (»Quellcodezeilen« oder auch »Anzahl der Programmzeilen«) und ist eine Metrik, um Software zu vermessen. Wenn du dir unseren Markdown-Text oben noch mal anschaust, siehst du, dass wir sechs Zeilen benutzt haben, aber nur in fünf Zeilen auch wirklich Inhalte (Source Lines) stehen. Das sind unsere SLOC.

SLOC werden häufig genutzt, um ein Gefühl für die Größe und Komplexität einer Software zu bekommen. Beispielsweise hat Windows XP 40 Millionen SLOC, der Linux-Kernel über 25 Millionen.<sup>13</sup>

SLOC sind übrigens *kein* Maßstab für Qualität oder Produktivität, auch wenn sie in Softwareprojekten hin und wieder als Messung für den Fortschritt angewendet werden. Gute Softwareentwicklung zeichnet sich unter anderem auch dadurch aus, dass Codezeilen vereinfacht (»umgeräumt«) oder auch mal entfernt werden<sup>14</sup> und dass nicht nur immer mehr Code hinzugefügt wird. Das nachfolgende Zitat beschreibt das in meinen Augen ganz gut:

»Die Verwendung von SLOC zur Messung des Softwarefortschritts ist wie die Verwendung von kg zur Messung des Fortschritts bei der Flugzeugherstellung.«

– Autor\*in unbekannt, angeblich Bill Gates

Wenn du dich genug an deinem Tun ergötzt hast, sollten wir einen Blick auf die Statistik für unser Projekt werfen (siehe Abbildung 3-13).

Vielleicht hast du schon bemerkt, dass sich die Anzahl der Commits auf deinem Projekt erhöht hat. Kein Wunder, du hast ja auch gerade deinen ersten gemacht. Eventuell wunderst du dich aber, dass da die Zahl 2 steht – schließlich war es ja gerade dein erster und nicht bereits der zweite Commit. Wir erinnern uns: Beim Anlegen des Repositorys haben wir GitHub gesagt, es möge für uns doch eine *README.md* anlegen – das war der erste Commit auf deinem Repo.<sup>15</sup>

13 Quelle: [https://de.wikipedia.org/wiki/Lines\\_of\\_Code](https://de.wikipedia.org/wiki/Lines_of_Code).

14 Das nennt man dann »Refactoring«.

15 Das gilt übrigens auch, wenn du beim Erstellen *.gitignore* oder *license* anwählst (siehe Abbildung 3-7).



Abbildung 3-13: Auf der Startseite eines Projekts ist die Gesamtanzahl aller Commits zu finden.

Herzlichen Glückwunsch, du hast dein erstes Repository angelegt und auch schon die erste Änderung vorgenommen. Jetzt wollen wir uns Issues und den Umgang damit anschauen.

## Den ersten Ablauf üben – Issue anlegen und bearbeiten

Wir wissen aus Abschnitt »Informationen finden (interessierte Anwenderin)« auf Seite 13 in Kapitel 2 bereits, dass Issues Handlungsbedarf aufzeigen sollen (Fehler, Anfragen nach weiteren Funktionen – sogenannte Feature-Requests –, Anmerkungen etc.). Ich habe sie als eine Art offener Brief bezeichnet. Ein Issue ist aber viel mehr als ein einfacher Brief. Stell ihn dir als einen Zettel an einer Pinnwand vor – alle können diesen lesen und auch eigene Kommentare anbringen. Häufig entstehen rege Diskussionen in einem Issue, zum Beispiel über das Für und Wider einer neuen Funktion.

Der größte Vorteil ist, dass Issues als eine Art Dokumentation für alle (auch für zukünftige Teammitglieder) dienen können, da dort optimalerweise alle Informationen zu einem bestimmten Thema zusammenlaufen, z.B. wieso eine Entscheidung so getroffen wurde, wie sie getroffen wurde.

Issues können von dir selbst angelegt werden, in vielen Projekten werden sie aber auch häufig von anderen Menschen (Contributoren) angelegt. Einen Issue schließen können die Projekteignerin, die Maintainerin oder der Contributor, der den Issue erstellt hat.

Um den Umgang damit zu üben, legen wir mal einen Issue an, in dem wir aufschreiben, dass uns eine wichtige Information auf der Startseite fehlt. Es kann dir durchaus passieren, dass andere Menschen auf dein Projekt stoßen und ihnen irgendwelche Informationen fehlen. In der Regel werden sie dies über einen Issue kundtun. Aber auch wenn du selbst Themen findest, die du noch bearbeiten und nicht vergessen möchtest, ist das Anlegen eines Issues eine gute Sache (in Kapitel 10, Abschnitt »Eigene Projektboards – mit Projects den Überblick behalten« auf Seite 236, werden wir sehen, dass es dafür auch andere Wege gibt). Zum einen vergisst du die Dinge nicht, und zum anderen können Besucherinnen deiner Seite

sehen, dass du dir des Themas durchaus bewusst bist und dort in (hoffentlich) naher Zukunft Abhilfe schaffst.<sup>16</sup> Im Folgenden werden wir:

1. einen Issue anlegen,
2. den Issue bearbeiten und
3. den Issue schließen.

## Einen Issue anlegen

Gehe zum Register *Issues*, wähle *New Issue* und gib dem Issue einen aussagekräftigen Namen und einen Inhalt, z.B. den aus Abbildung 3-14.<sup>17</sup>

Speichere deinen neuen Issue, indem du auf den Button *Submit new issue* (deutsch etwa »neuen Issue einreichen«) klickst. Wenn du danach im Projektmenü *Issues* auswählst, erscheint jetzt der eben von dir neu angelegte Issue.

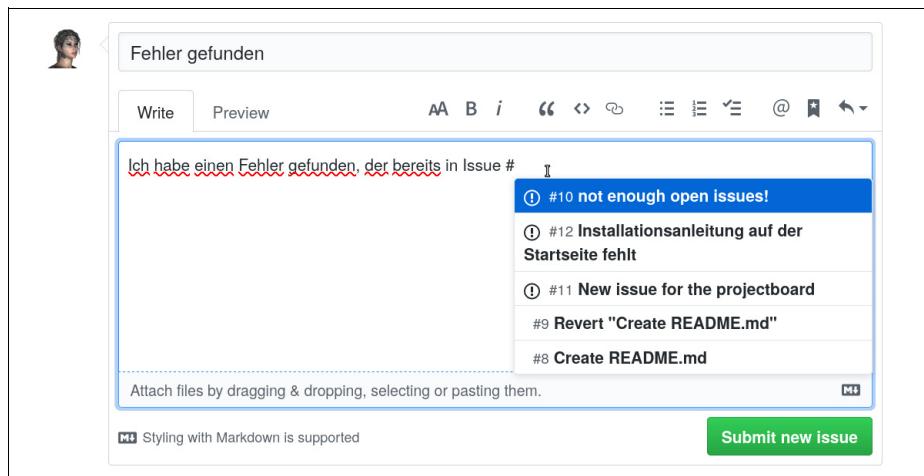


Abbildung 3-14: Das Erstellen eines Issues ist handwerklich ziemlich einfach. Die größere Herausforderung besteht darin, einen gescheiten Titel und eine gute Beschreibung zu finden.

Auf dem Bild habe ich noch eine Besonderheit eingebaut. Innerhalb eines Issues gibt es die Möglichkeit, auf andere Issues oder Pull-Requests zu verlinken. Das ist vor allem dann nützlich, wenn man zeigen möchte, dass ein anderer Issue ein ähnliches Problem adressiert oder ein bestimmter Pull-Request das Problem lösen könnte (Pull-Requests lernst du in Kapitel 4 noch kennen). Diese Verlinkung funktioniert über das Eingeben des Rautesymbols #, und GitHub unterstützt uns mit einem Drop-down und einer Auswahl an Möglichkeiten. Das funktioniert allerdings nur, wenn bereits ein Issue oder Pull-Request vorhanden ist.

<sup>16</sup> Es kann sogar passieren, dass jemand anderes um die Ecke kommt und den Issue für dich löst ...

<sup>17</sup> Wie du in der Abbildung gut sehen kannst, unterkringelt GitHub einige Wörter. Das ist die Rechtschreibprüfung, die leider kein Deutsch spricht.

Es gibt auch noch die Möglichkeit, mit sogenannten @mentions<sup>18</sup> Menschen direkt zu adressieren, die beispielsweise vielleicht einen Blick auf den Issue werfen sollten. Diese bekommen dann eine Benachrichtigung, dass sie erwähnt wurden. Abbildung 3-15 ist ein realer Beitrag in einem Issue aus dem *moby*-Projekt<sup>19</sup>, das wir in Kapitel 2 bereits kurz kennenlernen durften.

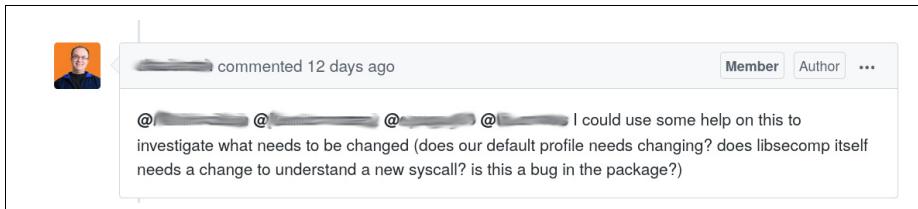


Abbildung 3-15: Über @mentions kann man Menschen direkt ansprechen. Diese erhalten eine entsprechende Benachrichtigung.

Sofern du Projekteignerin oder Maintainerin bist, kannst du Issues bestimmten Personen zuordnen, die dann für den entsprechenden Issue verantwortlich sind (englisch *Asignee*). Diese Verantwortung kann im Laufe der Zeit wechseln, z.B. wenn der Datenbankspezialist bei der Suche feststellt, dass der Fehler an einer ganz anderen Stelle ist. Wenn er den Fehler deswegen nicht lösen kann, gibt er den Issue vielleicht weiter an die Hauptentwicklerin. Da dein Projekt noch relativ frisch und neu ist, gibt es bisher nur eine Person, der du die Verantwortung übertragen kannst.<sup>20</sup>

Um Issues besser wiederfinden zu können oder auch um die Dringlichkeit oder Wichtigkeit eines Issues klarzumachen, kann man sie mit sogenannten *Labels* klassifizieren. GitHub bietet von Haus aus eine Reihe vorgefertigter Labels an (siehe Abbildung 3-16). Es ist aber auch möglich, eigene Labels zu erstellen, wenn es für das jeweilige Projekt sinnvoll erscheint. Labels werden uns später noch helfen, andere Projekte zu finden, bei denen wir unterstützen können (siehe Abschnitt »Fremdes Projekt suchen« auf Seite 117 in Kapitel 6).

Wir wollen zu Übungszwecken dem Issue ein Label geben und dir zuweisen. Es gibt dafür zwei Wege.

- **Weg 1:** Du wählst den entsprechenden Issue aus und bekommst auf der rechten Seite ein entsprechendes Menü (siehe Abbildung 3-17).
- **Weg 2:** Du gehst auf die Issue-Übersichtsseite (siehe Abbildung 3-18) und wählst über die Checkbox den entsprechenden Issue an ①. Danach kannst du über die Drop-down-Felder die Aktionen durchführen ② + ③. Dieser Weg hat den großen Vorteil, dass du mehrere Issues gleichzeitig bearbeiten kannst.

18 Das @-Symbol wird mitgesprochen.

19 <https://github.com/moby/moby>

20 Wenig überraschend: dir selbst!

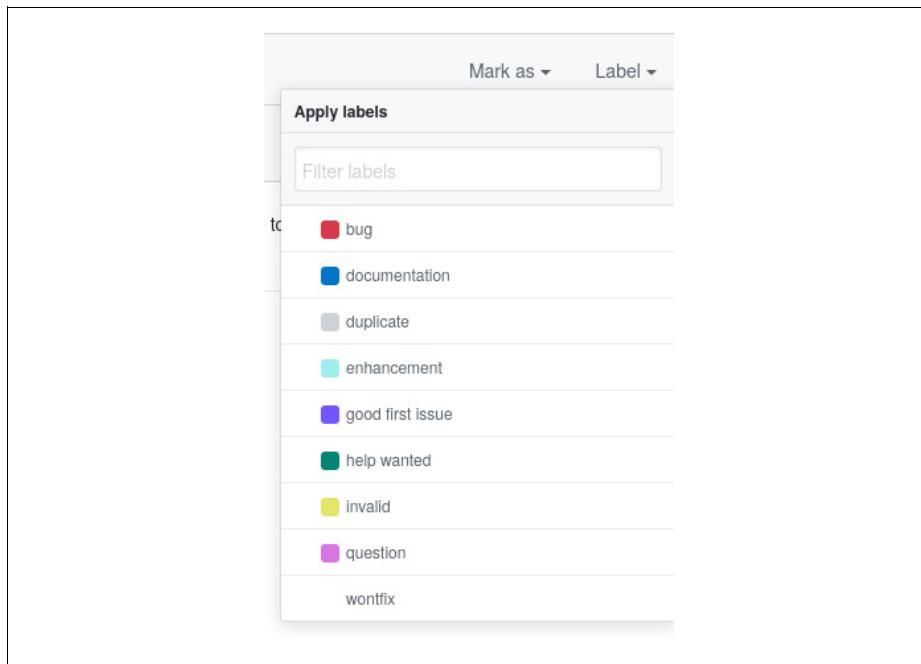


Abbildung 3-16: GitHub bietet einige Labels von Haus aus bereits an. Jedes Label kann eine unterschiedliche Farbe haben.

The screenshot shows a GitHub issue page for a pull request titled 'Installationsanleitung auf der Startseite fehlt #12'. The issue is marked as 'Open' and has 0 comments. The right sidebar contains several sections: 'Assignees' (No one—assign yourself), 'Labels' (None yet), 'Projects' (None yet), 'Milestone' (No milestone), and 'Linked pull requests' (Successfully merging a pull request may). Arrows point from the text labels in the caption to the corresponding sections in the sidebar.

Abbildung 3-17: Issue labeln und zuweisen, Weg 1: direkt im Issue auf der rechten Seite

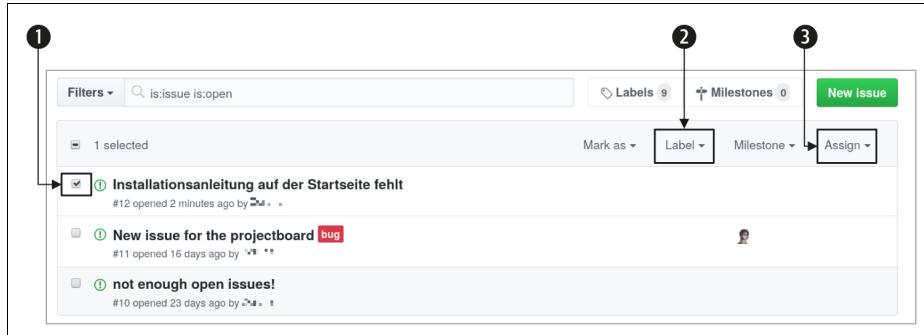


Abbildung 3-18: Issue labeln und zuweisen, Weg 2: in der Issue-Übersicht nach Anklicken des/ der jeweiligen Issues

Wähle ein Label deiner Wahl (z.B. *enhancement*) und wähle dich als *Asignee* aus. Das Ganze könnte dann so aussehen wie in Abbildung 3-19. Das Zuweisen eines Labels oder Asignees erfolgt durch einen einfachen Klick und muss nicht noch gesondert gespeichert oder bestätigt werden. Das kann leicht dazu führen, dass man aus Versehen etwas falsch zuweist. Wie man in der Abbildung sieht, habe ich zunächst das falsche Label *question* ausgewählt und bin danach erst zum richtigen Label *enhancement* gegangen. Hier sieht man sehr gut, dass GitHub jede Aktivität in einem Issue auch dokumentiert und dadurch für jeden und jede nachvollziehbar macht – selbst wenn es sich um einen Fehler handeln sollte.

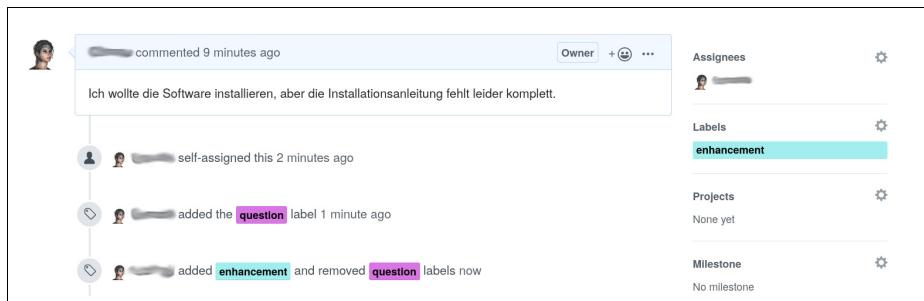


Abbildung 3-19: Issues können Personen (»Asignees«) zugeordnet werden und können Etiketten (»Labels«) haben, um sie klassifizieren zu können.

## Den Issue bearbeiten und schließen

Jetzt wollen wir das Problem lösen, das in dem Issue beschrieben wird, indem wir die Startseite entsprechend editieren (man nennt das häufig auch »den Issue lösen«). Gehe zur *README.md*, wechsle in den Editormodus und führe ein paar Änderungen durch. Bevor du speicherst, tippst du in die detaillierte Beschreibung *fixes #1* ein. Sofern du nicht schon andere Dinge innerhalb deines Projekts gemacht hast, adressiert das unseren eben erstellten Issue (1). GitHub gibt dir dazu via Tool-

tipp<sup>21</sup> einen Hinweis dazu, ob dem so ist (siehe Abbildung 3-20). Solltest du schon etwas »rumgespielt« haben, musst du eventuell eine andere Identifikationsnummer (ID) angeben, wie man bei mir mit der Nummer 12 gut sieht. Die ID deines Issues findest du in der Issue-Übersicht (siehe Abbildung 3-21).

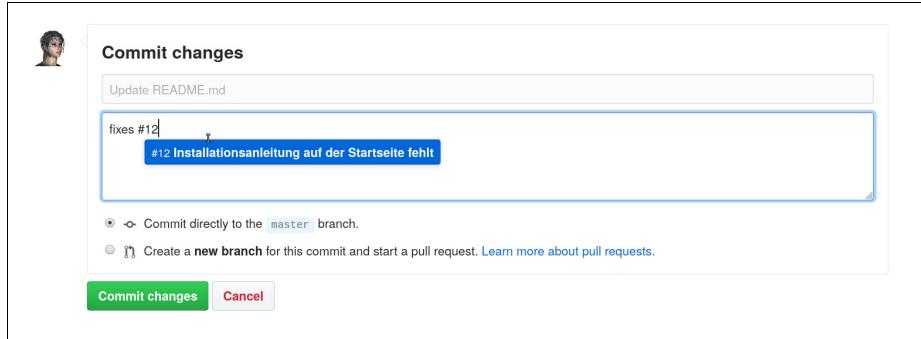


Abbildung 3-20: GitHub unterstützt dich mit nützlichen Tooltips.

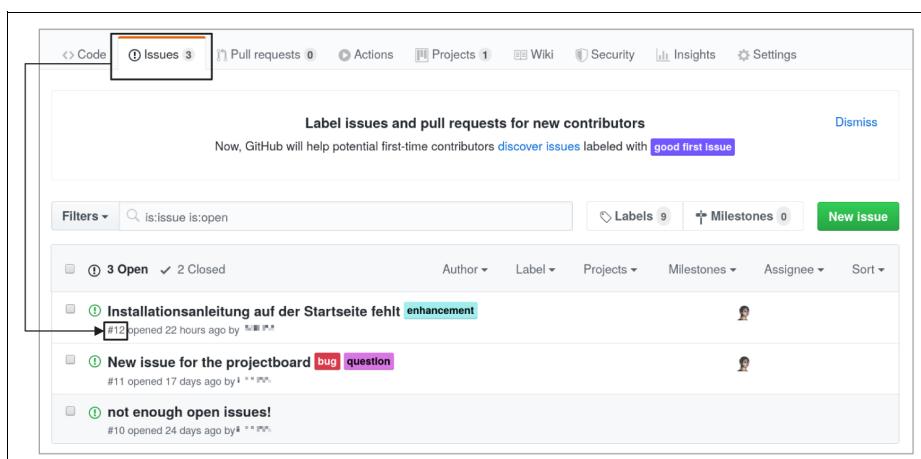


Abbildung 3-21: Die Identifikationsnummer (ID) eines Issues ist in der Issue-Übersicht zu finden.

Sobald du die richtige ID angegeben hast, speicherst du wie vorher auch. Jetzt ist aber noch etwas Bemerkenswertes passiert. Wenn du in die Issue-Übersicht wechselst, siehst du, dass der Issue weg ist?! Ja und nein, der Issue ist nicht weg, sondern wurde automatisch geschlossen. Der Standardfilter sorgt dafür, dass nur offene Issues angezeigt werden, daher kannst du ihn nicht mehr sehen (erinnere dich an

<sup>21</sup> Ein kleines Fenster, das sich in Programmen oder Webseiten öffnet und in der Regel eine kurze Beschreibung oder Hilfestellung gibt.

Kapitel 2, Abschnitt »Informationen finden (interessierte Anwenderin)« auf Seite 13, hier hatte ich die beiden Standardfilter `is:issue is:open` bereits erklärt). Wenn du den Filter veränderst, um dir die geschlossenen Issues anzeigen zu lassen, siehst du den bearbeiteten Issue wieder. Klickst du nun den entsprechenden Issue noch mal an, bekommst du ein paar weitere Informationen, beispielsweise dürfte dort so etwas Ähnliches stehen wie:

DeinUsername closed this in 7a0eeb2 2 minutes ago

Dabei ist die komische kryptische Zeichenkette (in meinem Fall `7a0eeb2`) mit einem Link versehen. Diese Zeichenkette ist die sogenannte *Commit-ID*. Jeder Commit, den du machst (oder andere machen), hat eine eigene ID, um die einzelnen Commits voneinander unterscheiden zu können. Später im Kapitel zu Git (siehe Kapitel 7) werden wir der ID erneut begegnen.

## Commit-ID

Die Commit-ID bei Git/GitHub besteht aus einem sogenannten SHA-1-Hash. Es handelt sich dabei um einen Algorithmus, der aus einer Eingabemenge eine 40 Zeichen lange Ausgabemenge produziert. Das heißt, hinten kommen immer 40 Zeichen raus – egal wie groß die Eingabemenge ist.

Das entscheidende Feature dabei ist, dass die exakt gleiche Eingabe auch immer die exakt gleichen 40 Zeichen erzeugt. Das hat den Vorteil, dass man dadurch relativ schnell sehen kann, ob zwei Commits identisch sind oder nicht. Git/GitHub bildet einfach mit zwei Commit-Eingabemengen (Inhalt, Datum etc.) den Hash-Wert, und wenn dieser gleich ist, sind beide Commits identisch.

Da 40 Zeichen ziemlich lang sind, um sie ständig anzuzeigen oder auch noch ständig irgendwo einzutippen, verwendet man in der Regel nur die ersten sieben bis zwölf Zeichen. Diese sind meist eindeutig genug.

Du kannst Issues auch manuell schließen, indem du in dem Issue ganz nach unten scrollst und den Button *Close issue* (deutsch »Issue schließen«) anwählst (siehe Abbildung 3-22). Das ist natürlich nicht ganz so toll wie das automatische Schließen, aber manchmal gibt es Issues, die einfach nicht gelöst werden – beispielsweise weil irgendeine Besucherin ein völlig abgefahrenes Feature über einen Issue bei dir eingekippt hat, das du nicht in dein Projekt aufnehmen möchtest. Sollte so etwas der Fall sein, schreibe in den Kommentar, warum du den Issue nicht umsetzen wirst, und zwar nicht nur, weil es höflicher ist, es erhöht auch die Chance, dass die Menschen weiterhin Issues anlegen (so du das denn möchtest). Nichts ist frustrierender, als ohne Begründung »beobachtet« zu werden.

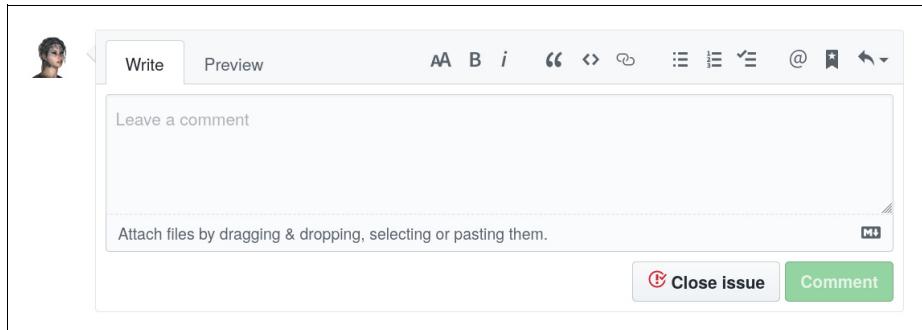


Abbildung 3-22: Issues kann man auch manuell schließen.



### Gelöste Issues schließen

Es ist gute Praxis, gelöste Issues zu schließen. Du behältst dadurch die Übersicht, und potenzielle Conributoren versuchen sich nicht an einer Lösung und verbraten Zeit.

## Issues mithilfe von Schlüsselwörtern automatisch schließen

GitHub bietet die Möglichkeit, Issues über Schlüsselwörter zu schließen. Die Schlüsselwörter sehen dabei wie folgt aus:

- *close*
- *closes*
- *closed*
- *fix*
- *fixes*
- *fixed*
- *resolve*
- *resolves*
- *resolved*

Sobald in einem Commit eines dieser Schlüsselwörter mit einer Referenz auf den Issue eingegeben wird (#ISSUE\_NR), wird dieser Issue geschlossen. Man kann auch mehrere gleichzeitig schließen, dafür muss aber vor jeder Referenz eines der Schlüsselwörter stehen, also etwa *fixes #1, fixes #2*.<sup>22</sup>

Man kann auch Issues in anderen Repositories schließen, vorausgesetzt, man hat die entsprechenden Berechtigungen. Das geht mit einem der Schlüsselwörter und als Referenz *USERNAME/REPONAME#ISSUE\_NR*.

22 Das Ganze gilt übrigens nur für den Standard-Branch (master), erfolgt ein Commit auf einen anderen Branch, wird der Issue nicht automatisch geschlossen. Wir waren bisher nur auf dem Standard-Branch unterwegs, daher sei diese Info nur der Vollständigkeit halber ergänzt. Keine Sorge, das Thema Branches wird im nächsten Kapitel noch klarer.

Du weißt nun, wie man Issues anlegt, sie labelt und jemandem zuweist. Als cooles neues Zusatzwissen kannst du jetzt auch Issues automatisch schließen lassen, wenn du mit bestimmten Schlüsselwörtern bei einem Commit arbeitest.

Eventuell hast du bereits ein oder mehrere Repositories zum Testen und Rumspielen angelegt. Um deinen Account wieder etwas aufzuräumen, zeige ich dir im nächsten Abschnitt, wie du ein Repository löschen kannst.

## Ein bestehendes Repository löschen

Das Löschen eines bestehenden Repositorys ist relativ einfach, aber auf den ersten Blick etwas furchteinflößend, wir müssen nämlich dafür in die *Danger Zone* (also in den »Gefahrenbereich«), siehe Abbildung 3-23. Gehe auf dein zu lösches Repository und wähle *Settings* aus. Um in die *Danger Zone* zu kommen, musst du ganz nach unten scrollen. Achte darauf, dass im linken Menü *Options* ausgewählt ist (sollte beim Anklicken von *Settings* standardmäßig der Fall sein).

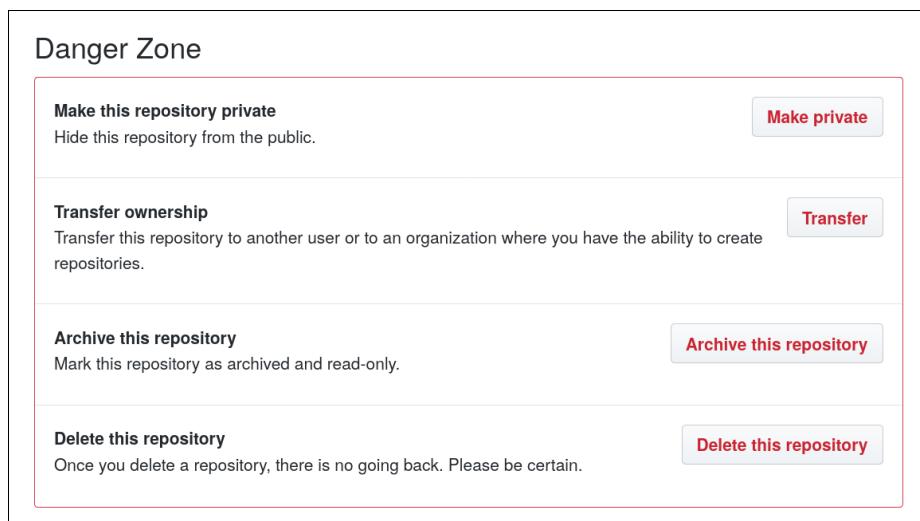


Abbildung 3-23: In der Danger Zone werden alle gefährlichen Aktionen eines Repositorys gebündelt.

In der *Danger Zone* wähle den Button *Delete this repository* (deutsch »Lösche dieses Repository«) aus. Es erscheint eine Warnmeldung, die dich dazu auffordert, den Namen deines Repositorys nach dem Muster USERNAME/REPONAME einzutippen, bevor du die Löschung mit Klick auf den Button *I understand the consequences, delete this repository* (deutsch etwa »Ich verstehe die Konsequenzen, lösche dieses Repository«) bestätigen kannst (siehe Abbildung 3-24). Das ist ein Sicherheitsmechanismus, damit du dein Repository mit all seinem Inhalt (Dateien, Issues, Kommentare, Wiki etc.) nicht aus Versehen löscht.

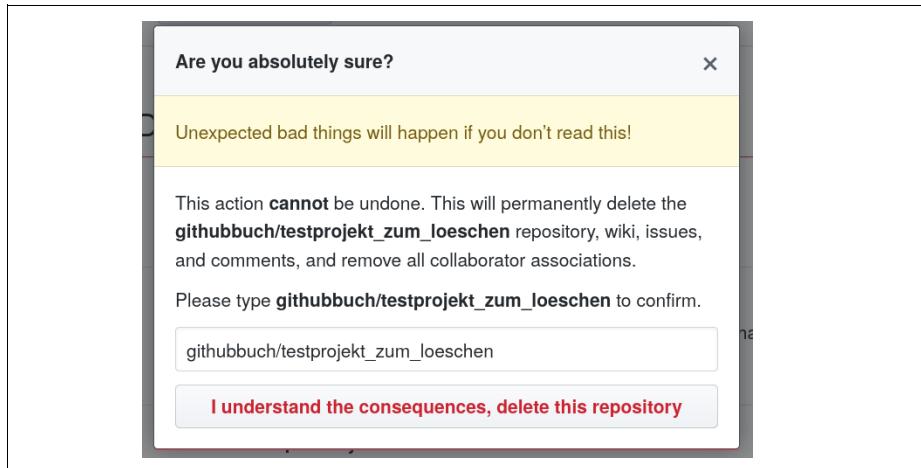


Abbildung 3-24: Als Sicherheitsmechanismus ist zur Löschung der Name des Repositorys vorher einzugeben.

Jetzt weißt du, wie du überflüssige und eventuell auch verbastelte Repositories löschen kannst. Als Nächstes (und Letztes für dieses Kapitel) möchte ich mit dir einmal durchgehen, wie du ein bereits bestehendes eigenes Projekt, das aktuell irgendwo auf der Festplatte schlummert, auf GitHub bekommst.

## Ein bestehendes Projekt hochladen

Ein bestehendes Projekt hochzuladen, ist simpel. Lege ein neues Repository für dein Projekt an. Klicke auf den Button *Add file* und dann auf *Upload files* (siehe auch Abbildung 3-25). Danach hast du die Möglichkeit, per Drag-and-drop oder durch Anklicken des Links *choose your files* die gewünschten Dateien und/oder Ordner auszuwählen und hochzuladen (siehe auch Abbildung 3-26).



Abbildung 3-25: Dateien und Ordner hochladen geht über Add file und Upload files.

Am Ende muss nur noch ein Commit durchgeführt werden, und die Dateien und/oder Ordner sind auf dein Repository hochgeladen. Bitte beachte: Die Dateien dürfen die Größe von 25 MByte pro Datei nicht überschreiten! Solltest du größere Dateien hochladen wollen, musst du auf die Konsole ausweichen (das lernst du in Kapitel 8 kennen).

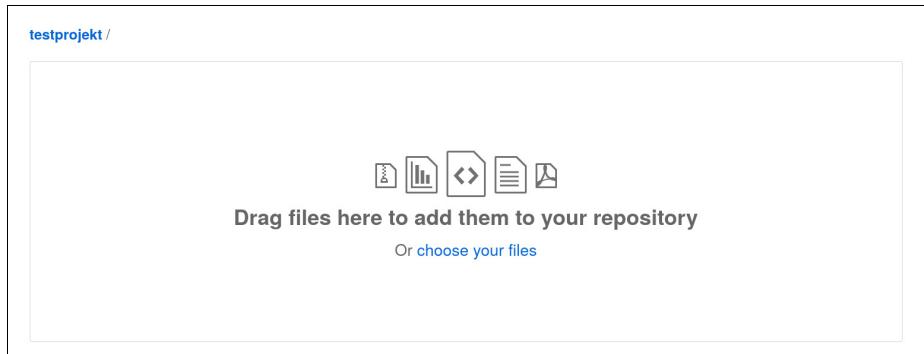


Abbildung 3-26: Via Drag-and-drop lassen sich Dateien und Ordner ganz leicht hinzufügen.

Damit hast du jetzt die Basis, um mit GitHub die ersten Schritte unfallfrei zu gehen. Um aber wirklich produktiv mit anderen zusammenzuarbeiten, fehlen noch ein paar Dinge. Da bei GitHub die Kollaboration im Vordergrund steht (wir erinnern uns an Kapitel 1, Abschnitt »Einsatzgebiete von GitHub« auf Seite 3), werden wir uns im nächsten Kapitel damit beschäftigen, was es zu beachten gibt, wenn plötzlich andere an unserem Projekt mitarbeiten.



## KAPITEL 4

# Die wichtigsten Grundlagen für eigene GitHub-Projekte

Im vorherigen Kapitel haben wir unsere ersten Commits durchgeführt, uns Issues und deren Bearbeitung angeschaut, Repositories gelöscht und ein Projekt, das bisher auf deiner Platte schlummerte, auf GitHub hochgeladen. In diesem Kapitel werden wir alles beleuchten, was wichtig sein könnte, wenn an deinem eigenen Projekt eine oder mehrere Personen beteiligt sind.



### Am Ende des Kapitels kannst du ...

- Branches anlegen und Änderungen darauf vornehmen.
- Pull-Requests erstellen und diese mergen.
- Reviews von Änderungen durchführen.
- Reviewer manuell und automatisch einem Review zuteilen.
- Branches vor ungewollten Änderungen schützen.
- Vorlagen für Issues und Pull-Requests erstellen.
- hitzige Debatten in deinem Projekt durch technische Mittel wieder beruhigen

Wir werden uns dafür als Erstes den wohl wichtigsten Ablauf von GitHub anschauen, den sogenannten *GitHub Flow* (Flow bedeutet so etwas wie »Ablauf«). Das Verständnis dieses Ablaufs wird uns auch helfen, wenn wir später bei anderen Projekten unterstützen wollen. Hier lernen wir *Branches*, *Pull-Requests* und *Merges* kennen.

Wenn du mit anderen zusammen an einem Projekt arbeitest, ist es zur Steigerung der Qualität sinnvoll, Änderungen durch mehrere Personen noch einmal gegenzutesten, sensible Bereiche besonders zu schützen und mit Vorlagen zu arbeiten. Ich zeige dir die Möglichkeiten, die GitHub für die Erfüllung dieser Aufgaben bietet.

## Den zweiten Ablauf üben – Branches, Pull-Requests und Merges

Im vorherigen Kapitel haben wir Änderungen an unserem Repository direkt gespeichert (»committet«). Sobald du aber mit mehreren auf einem Projekt unterwegs bist, kann es passieren, dass ihr euch schnell in die Quere kommt, da in der Regel für eine größere Änderung mehr als eine Datei angefasst werden muss. Es ist daher häufig sinnvoll, wenn jeder und jede für sich isoliert innerhalb eines eigenen »Handlungsstrangs« arbeitet und dort alle benötigten Änderungen durchführt. Erst später, wenn der Handlungsstrang in sich fertig und stimmig ist, wird er zurück in das eigentliche Projekt »eingefädelt«. So kann man sich nicht ins Gehege kommen.

Diese Handlungsstränge heißen bei GitHub (und auch anderswo) *Branches* (deutsch »Zweige«). Wir haben den Begriff schon bei unserem ersten Commit im vorherigen Abschnitt kennengelernt. Das »Wiedereinfädeln« nennt sich *Merge* (deutsch »Verschmelzung«). Zusammen mit den *Pull-Requests* (deutsch etwa »Aufforderung, sich eine Veränderung zu holen«) bilden sie den wichtigsten Ablauf bei GitHub: den *GitHub Flow*<sup>1</sup>. Wir schauen uns nacheinander diese drei Kernelemente von GitHub genauer an.

### Branch – unterschiedliche Handlungsstränge aufmachen

Wenn wir mit GitHub (und später auch mit Git) arbeiten, haben wir es immer mit Branches zu tun – ob wir wollen oder nicht. Selbst wenn wir selber keine anlegen, befinden wir uns immer auf einem »Standard-Branch«, der *master* genannt wird (siehe hierzu auch Abbildung 4-1). Jedes Mal, wenn wir committen, verlängern wir den Zweig um einen Schritt.



#### Änderung beim Namen des Standard-Branches

GitHub hat bereits angekündigt, zukünftig den Namen des Standard-Branchs von *master* auf *main* zu ändern.<sup>2</sup> Auslöser hierfür ist unter anderem die Black-Lives-Matter-Bewegung. GitHub und andere Softwareanbieter (z.B. auch Git) bemühen sich, Wörter, die beleidigend oder anstößig empfunden werden könnten, durch neutrale zu ersetzen. Der Begriff »master« erinnert viele an Sklaverei (Master und Slaves).

Ich werde im weiteren Verlauf des Buchs weiterhin *master* nutzen, da dies der aktuelle Stand ist und dir vermutlich auch noch eine Weile begegnen wird.

<sup>1</sup> Es gibt auch noch andere Ansätze, beispielsweise das sogenannte Trunk Based Development, siehe <https://trunkbaseddevelopment.com/>. Das zu erklären, würde hier aber zu weit führen.

<sup>2</sup> <https://github.com/github/renaming>

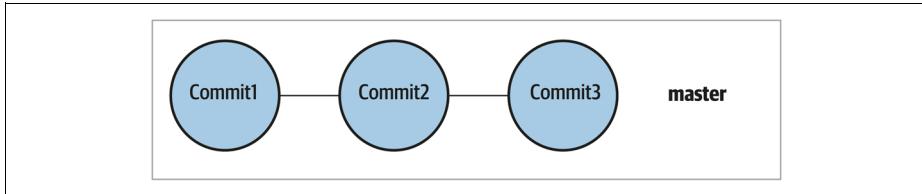


Abbildung 4-1: Ein Branch ist eine lineare Liste an Commits, hier auf dem master-Branch.

Branches dienen unter anderem dazu, unterschiedliche »Handlungsstränge« oder auch »isolierte Umgebungen« aufzumachen, sodass man mit mehreren Personen an einer Sache arbeiten kann, ohne sich ins Gehege zu kommen oder irgendetwas komplett kaputt zu machen. Auch wenn man alleine an einem Projekt werkelt, können Branches durchaus sinnvoll sein, z.B. um Veränderungen am Quellcode besser zu organisieren. Schau dir hierzu Abbildung 4-2 an. Für die Entwicklung einer neuen Funktion (eines neuen Features) wird häufig ein neuer Branch erzeugt, der konsequenterweise *Feature-Branch* heißt. Trotzdem können parallel weiter auch Veränderungen an master-Branch erfolgen, d.h., nach Commit 3 bzw. 4 können noch weitere Commits folgen, selbst ein weiteres Verzweigen in noch mehr Branches ist möglich. Feature-Banches sind aber nur eine Möglichkeit, auch Branches für die Fehlerbehebung (*bugfixing*) oder die Veröffentlichung einer Softwareversion (*releases*) werden häufig eingesetzt.

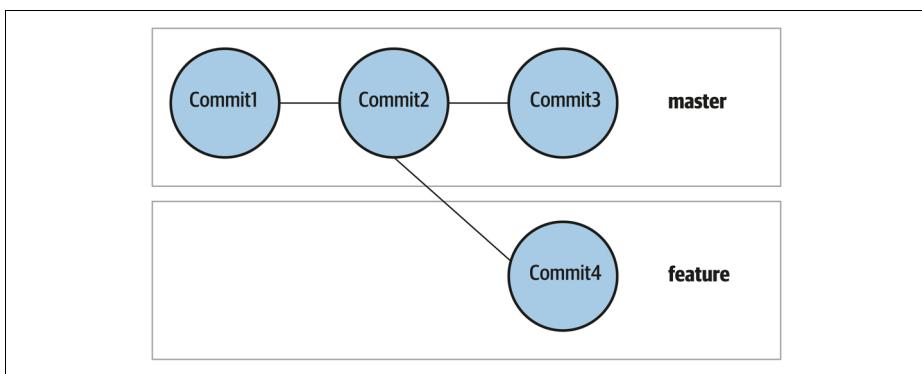


Abbildung 4-2: Ein neuer Branch wird häufig genutzt, um Veränderungen vorzunehmen, ohne die stabile Version auf dem master-Branch anzufassen.

Wenn man zwei Branches wieder zusammenführen will, nennt man das einen *Merge* (siehe hierzu Abbildung 4-3). *Commit5*, der beide Zweige wieder zusammenführt, wird auch *Merge Commit* genannt. Dieser Commit hat deswegen eine eigene Bezeichnung, weil er im Unterschied zu »echten Commits« inhaltlich nichts Neues erzeugt, sondern nur als eine Art Verwaltungsakt durchgeführt wird. Das schauen wir uns im Abschnitt »Merge – Änderungen aus Pull-Requests übernehmen« auf Seite 59 gleich noch genauer an.

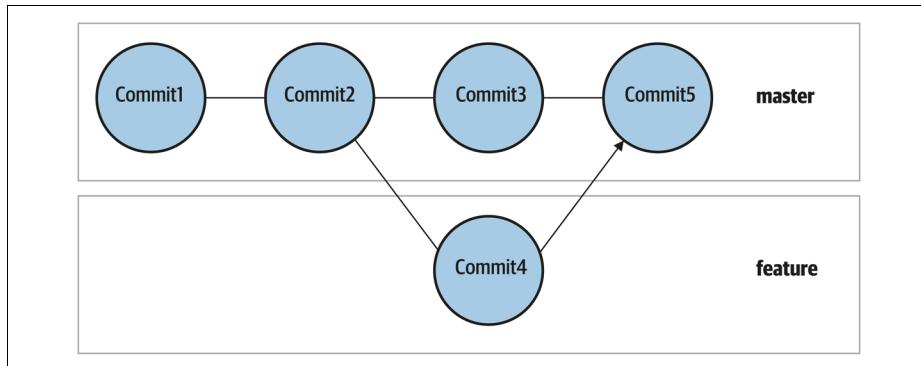


Abbildung 4-3: Veränderungen in einem Feature-Branch werden mittels Merge zurück in den master-Branch übernommen.

Wann und wofür du einen neuen Branch sinnigerweise anlegen solltest, hängt von dem Aufbau deines Projekt ab, z.B. wie komplex es ist oder ob du mit mehreren Leuten auf demselben Projekt arbeitest. Nachfolgendes soll dir eine Idee davon geben, wann ein neuer Branch eventuell sinnvoll sein könnte:

1. Du möchtest eine größere oder einschneidende Veränderung vornehmen.
2. Du möchtest einige Änderungen vornehmen, die möglicherweise nicht verwendet werden.
3. Du möchtest an etwas experimentieren, von dem du nicht sicher bist, ob es funktioniert.
4. Du arbeitest mit mehreren Entwicklerinnen parallel auf demselben Projekt.
5. Du willst Bugfixes auch noch für ältere Versionen bereitstellen.

In größeren Projekten ist es durchaus üblich, festzulegen, wofür Branches angelegt werden dürfen, wie Branches benannt werden, wer das machen darf und welche Entwicklungsaufgaben (beispielsweise Fehlerbehebung oder Neuentwicklung) auf welchen Branches durchgeführt werden.



#### Tipp: Namenskonvention für Branches

Der Name eines Branchs sollte seine Aufgabe beschreiben, sodass andere sehen können, an was da gerade gearbeitet wird. Eine Möglichkeit besteht darin, dass über eine Namenskonvention zu erreichen. Beispielsweise könntest du die Branchnamen wie folgt aufbauen: »Grund-Details«, beispielsweise »fix-login«, »neues-feature-kalorienrechner« oder »entferne-doppelte-bilder«.

In der beruflichen Praxis hat es sich auch bewährt, Namenskürzel oder Initialen als Präfix von Branchnamen zu wählen, beispielsweise »al/fix-login«. Dann sieht man sofort, wer für einen Branch verantwortlich ist, und kann bei Bedarf die Person direkt ansprechen.

## Warum Branches sinnvoll sein könnten

Mal ein Beispiel außerhalb der Computerwelt, um das Ganze etwas griffiger zu machen: Stell dir vor, dein Kind bräuchte eine Verkleidung für eine Grundschulveranstaltung (z. B. für eine Aufführung oder weil gerade Karneval ist).<sup>3</sup> Ihr habt schon gemeinsam ein Kostüm ausgesucht, Kind ist glücklich, der Geldbeutel etwas schmäler. Jetzt hat das Kind aber irgendwo gesehen, dass andere Kinder kurze Hosen tragen, und möchte so etwas auch haben. Du selber hast ebenfalls schon über eine Veränderung nachgedacht, nämlich ob nicht ein paar zusätzliche Sterne – mit der Heißklebepistole angebracht – das Kostüm verschönern würden.

Wenn ihr jetzt beide, dein Kind und du, gleichzeitig oder auch nacheinander eure Änderungswünsche am Originalkostüm (master-Branch) durchführt, könnte jetzt Folgendes passieren:

- Das Kind könnte feststellen, dass ihm der »Scherenschnitt« doch nicht so gut gelungen ist, und jetzt mag es das Kostüm bzw. die dazugehörige Hose nicht mehr anziehen.
- In deinen Augen sehen deine aufgeklebten Sterne ganz gut aus, aber dein Kind (= die Entscheidungsinstanz) findet sie blöd und möchte das Kostüm jetzt als Ganzes nicht mehr tragen.
- Ihr müsst noch mal los und ein komplett neues Kostüm kaufen, da das Originalkostüm verhunzt ist ... und morgen früh ist die Aufführung, ihr habt euch zerstritten, und die Geschäfte schließen in fünf Minuten.

Ich denke, du bekommst eine Idee davon, was für Konsequenzen es haben kann, wenn mehrere Menschen am »Original« arbeiten, obwohl noch nicht sicher ist, ob die Änderungen gut sein werden. Wie geht man mit dieser Situation um? Es wäre doch viel besser, wenn man eine Art Sicherheitskopie von dem Kostüm machen könnte (das nennt man »Feature-Branch erstellen«), um die gewünschten Änderungen an der Kopie erst einmal auszuprobieren. Gefallen die Änderungen nicht, wird die Sicherheitskopie einfach weggeworfen,<sup>4</sup> und das Originalkostüm ist immer noch so schön wie am ersten Tag.

Da ihr zwei ändernde Personen seid (du und dein Kind), könnte man für jede Person bzw. jedes anzupassende Feature eine eigene Sicherheitskopie (= einen eigenen Feature-Branch) machen und dort die jeweiligen Änderungen erst einmal ausprobieren. Wie kommen jetzt die Änderungen, die der Entscheidungsinstanz gefallen, in das Kostüm? Den Begriff haben wir auch schon gehört: Dieses nennt sich *Pull-Request* (siehe auch Abschnitt »Pull-Request – Änderungen in Branches aufzeigen« auf Seite 53 weiter unten). In einem Pull-Request könnten jetzt auch noch Details diskutiert werden (»Das sind zu viele Sterne, wenn du fünf wieder entfernst, würde es mir besser gefallen!«), der Pull-Request kann angepasst und schlussendlich auf dem Originalkostüm umgesetzt werden (»in den master-Branch gemergt werden«).

3 Mit lieben Grüßen an Franz, die mich zu dem Beispiel inspiriert hat.

4 In der nicht digitalen Welt würde ich das als Verschwendug ansehen und jetzt die Konsumgesellschaft anprangern, es ist zum Glück nur ein Gedankenexperiment.

Wir wollen unseren ersten Branch anlegen und gehen dafür auf die Startseite unseres Projekts. Dort gibt es einen Button, der mit einem Symbol und dem Wort *master* beschriftet ist (siehe hierzu auch Abbildung 4-4). Klicke darauf (1) und befülle das Textfeld mit dem Namen, den du dem Branch geben möchtest (2, beispielsweise `fix-login`). Sobald du die Enter-Taste gedrückt hast, wird der Branch erzeugt. Über den Button kannst du später auch zwischen den vorhandenen Branches hin- und herwechseln (siehe Abbildung 4-5).

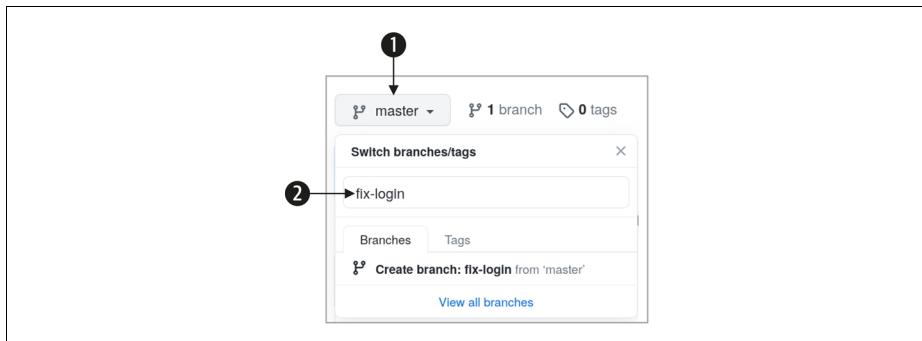


Abbildung 4-4: Einen neuen Branch kannst du über diesen Button anlegen.

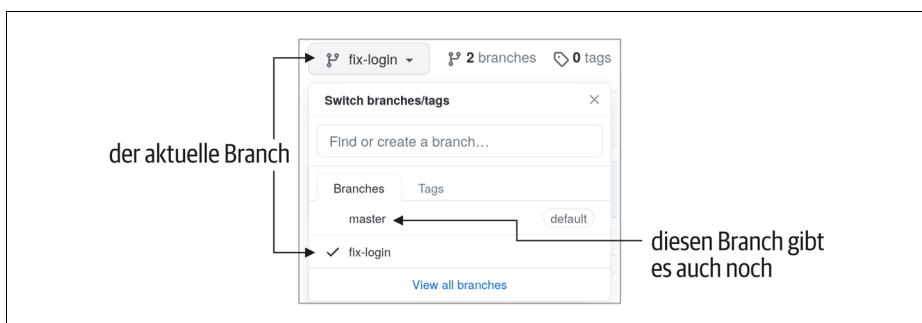


Abbildung 4-5: Derselbe Button dient auch dazu, zwischen den vorhandenen Branches zu wechseln.

Und was machen wir jetzt mit diesem Branch? Ein Branch an sich ist erst einmal wenig spektakulär. Erst wenn wir anfangen, Änderungen an unserem Repository vorzunehmen (sprich: Commits durchführen), kommen sie wirklich zum Einsatz. Das nehmen wir uns im nächsten Abschnitt vor.

## Änderungen auf einem Branch vornehmen

Du hast im vorherigen Abschnitt einen Branch erzeugt, den wir jetzt einsetzen wollen. Achte darauf, dass du dich auf dem gerade neu erstellten Branch (bei mir `fix-login`) befindest, wie in Abbildung 4-6 dargestellt. Erstelle jetzt eine neue Datei,

indem du den Button *Add file* und dann *Create new file* anklickst, einen Dateinamen wählst und die Enter-Taste drückst. Der Dateiname oder der Inhalt der Datei ist erst einmal egal. Bevor du das Ganze mit einem Commit speicherst, schau dir Abbildung 4-7 an. Wenn hier jetzt dein neu erstellter Branchname auftaucht, bist du wie gewünscht auf dem richtigen Branch.



Abbildung 4-6: Auf dem neuen Branch soll eine neue Datei erstellt werden.

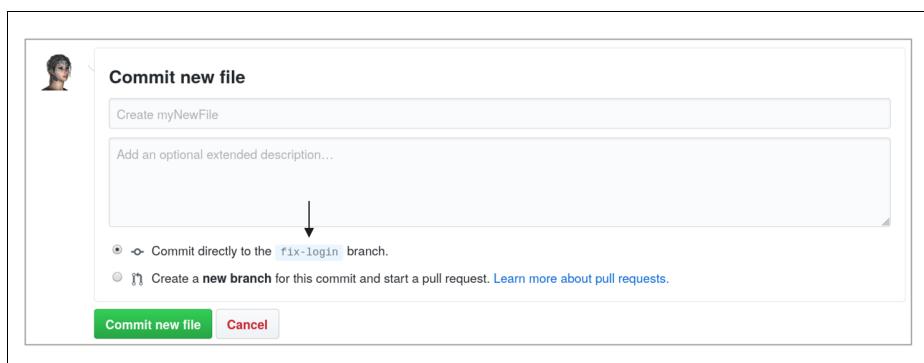


Abbildung 4-7: Spätestens beim Commit sieht man, ob man auf dem richtigen Branch ist.

Vielleicht verstehst du jetzt auch schon die Auswahloption darunter etwas besser (*Create a new branch for this commit and start a pull request*, deutsch etwa »Erzeuge für diesen Commit einen neuen Branch und starte einen Pull-Request«). Solltest du vorher vergessen haben, den neuen Branch für deine Änderungen anzulegen, hättest du hier noch mal die Möglichkeit, das nachzuholen.



#### Tipp: Verzeichnis anlegen

Verzeichnisse in deinem Repository kannst du ganz einfach während des Anlegens einer neuen Datei erzeugen. Gib dafür vor dem Dateinamen den Verzeichnisnamen inklusive / (Slash) ein (z.B. testdir/, *dir* wird häufig als Abkürzung für *directory* genutzt, siehe auch Abbildung 4-8) und danach den Dateinamen. Ein leeres Verzeichnis anzu-

legen, geht jedoch nicht, es muss immer eine Datei dabei erzeugt werden.

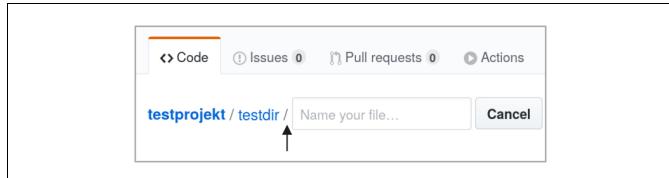


Abbildung 4-8: Ein Verzeichnis kann man in GitHub anlegen, wenn man einen Slash (/) beim Erstellen einer neuen Datei einfügt.

Nach deinem Commit erscheint die neue Datei erwartungsgemäß in der Dateiaufstellung des Repositorys, wie in ❶ in Abbildung 4-9 dargestellt. Jetzt sind aber auch die Bereiche ❷ und ❸ aufgetaucht, die vorher noch nicht da waren. Bereich ❷ will uns lediglich darüber informieren, dass unser derzeitiger Branch (*fix-login*) einen Commit mehr hat als unser master-Branch (*This branch is 1 commit ahead of master*, zu Deutsch etwa »Dieser Branch ist dem master-Branch einen Commit voraus«).<sup>5</sup> Bereich ❸ zeigt uns die Handlungsoption *Compare & pull request* auf (*Compare* bedeutet »Vergleiche«, *pull request* hatte ich weiter oben schon mit »Aufforderung, sich eine Veränderung zu holen« übersetzt), die wir wahrnehmen können, aber nicht müssen.

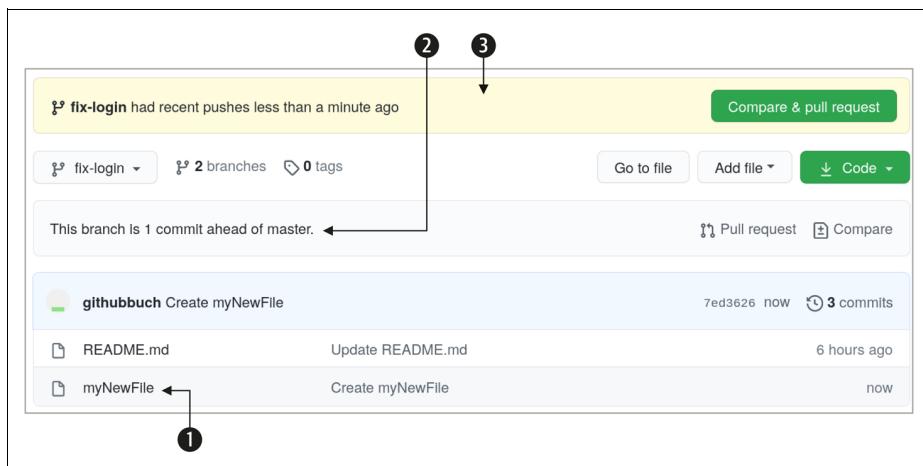


Abbildung 4-9: Nach einem Commit auf einen neuen Branch bietet die GitHub-Oberfläche weitere Informationen und Handlungsoptionen an.

5 Hättest du vorhin zwei Dateien auf dem neuen Branch erstellt, hätte er auch zwei Commits mehr als der master-Branch etc. vergleiche auch den Kasten »Die GitHub-Metriken ›behind‹ und ›commits ahead‹« weiter hinten in diesem Kapitel.



### Tipp: Branchverwaltung

Vielleicht möchtest du in deinem Projekt den Standard-Branch von *master* auf *development* anpassen? Oder du möchtest alte Branches löschen (oder auch wiederherstellen)? Hier bietet die Branchverwaltung gute Dienste an. Du findest sie direkt auf der Startseite des Repositorys (siehe auch Abbildung 4-10 und Abbildung 4-11).

The screenshot shows the top navigation bar of a GitHub repository. It includes a dropdown for 'fix-login', a count of '2 branches', and '0 tags'. On the right are buttons for 'Go to file', 'Add file', and 'Code'. An arrow points upwards from the text below to the 'branches' button.

Abbildung 4-10: Durch Anklicken von »branches« kommt man in die Branchverwaltung.

The screenshot shows the 'Branches' page with tabs for 'Overview', 'Yours', 'Active', 'Stale', and 'All branches'. A search bar is at the top right. Below, the 'Default branch' section shows 'master' as the default branch, updated 6 hours ago by 'githubbuch'. The 'Your branches' section shows 'fix-login' updated 5 minutes ago by 'githubbuch'. The 'Active branches' section also shows 'fix-login' updated 5 minutes ago by 'githubbuch'. Each branch entry has a 'New pull request' button and a trash icon.

Abbildung 4-11: In der Branchverwaltung lassen sich Branches löschen, oder es kann ein anderer Standard-Branch festgelegt werden.

Wir wechseln zurück auf den *master*-Branch, ohne die Handlungsoption auszuwählen. Dort sollte es so ähnlich aussehen wie in Abbildung 4-12.

The screenshot shows the top navigation bar of a GitHub repository. It includes a dropdown for 'fix-login', a count of '2 branches', and '0 tags'. On the right are buttons for 'Go to file', 'Add file', and 'Code'. An arrow points upwards from the text below to the 'master' dropdown. Another arrow points down to the commit history. A circled '1' is at the bottom left, and a circled '2' is at the top right of the commit history area.

Abbildung 4-12: Auf dem master-Branch fehlt die neue Datei, aber die Handlungsoption ist auch hier vorhanden.

Unsere neue Datei ist nicht mehr zu sehen, da sie sich auf einem anderen Branch befindet (bei mir *fix-login*). Das wird verständlicher, wenn du dir Abbildung 4-2 noch mal anschaugst. Der Commit, mit dem wir die Datei gespeichert haben, ist dort *Commit4*. Die Info zu *commit ahead* ist ebenfalls verschwunden ①. Das liegt

daran, dass GitHub alle Änderungen immer mit dem master-Branch vergleicht. Er ist der Dreh- und Angelpunkt von allem. Die Handlungsoption *Compare & pull request* sehen wir aber weiterhin (siehe ②), d.h., du könntest sie starten, egal auf welchem der beiden Branches du dich befindest.

## Die GitHub-Metriken »behind« und »commits ahead«

Wenn du mit Branches arbeitest, werden die einzelnen Branches zwangsläufig »auseinanderdriften«, während du am Projekt arbeitest. GitHub bietet dir eine einfache Metrik an, mit deren Hilfe du sehen kannst, wo dein jeweiliger Branch im Vergleich zu deinem Standard-Branch (in der Regel dem master-Branch) steht. Dies ist einerseits die Metrik *This branch is X commits ahead of master*, die du schon kennengelernt hast, und zum anderen *This branch is X commits behind master* (deutsch etwa »Dieser Branch ist X Commits hinter master«).

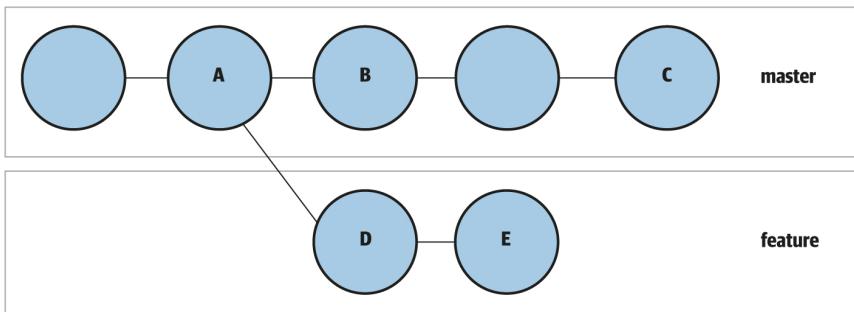


Abbildung 4-13: Beim Einsatz von Branches driften diese im Laufe der Zeit zwangsläufig auseinander.

Sieh dir hierzu Abbildung 4-13 an. Wir haben zwei Branches mit unterschiedlicher Anzahl an Commits (jeder Kreis stellt wieder einen Commit dar). In Tabelle 4-1 siehst du, was GitHub dir in welcher Konstellation anzeigt und was es jeweils bedeutet. Wir betrachten dabei immer nur die beiden Branches (master und feature) in Relation zueinander, d.h., den Unterschied von Commit B zu Commit C könnte man zwar darstellen, das brauchen wir hier aber nicht, da beide auf demselben Branch sind.

Tabelle 4-1: Metriken von GitHub und ihre Bedeutung

Commit1 (feature)	in Relation zu	Commit2 (master)	Bedeutung
D	feature is 1 commit ahead of master	A	Auf den Feature-Branch wurde 1x committet, bei master ist nichts los.
D	feature is 1 commit behind and 1 commit ahead of master	B	Auf beide Branches wurde 1x committet.
D	feature is 3 commits behind and 1 commits ahead of master	C	Beide Branches in der Entwicklung, auf den master-Branch wurde 3x committet, beim Feature-Branch 1x.

Tabelle 4-1: Metriken von GitHub und ihre Bedeutung (Fortsetzung)

Commit1 (feature)	in Relation zu	Commit2 (master)	Bedeutung
E	feature is 2 commits ahead of master	A	Stärkere Weiterentwicklung auf dem Feature-Branch (2 Commits), bei master ist wieder nichts los.
E	feature is 1 commit behind and 2 commits ahead of master	B	Beide Branches in der Entwicklung, auf den master-Branch wurde 1x committet, beim Feature-Branch 2x.
E	feature is 3 commits behind and 2 commits ahead of master	C	Beide Branches in der Entwicklung, auf den master-Branch wurde 3x committet, beim Feature-Branch 2x.

Diese Metrik gibt dir also einen groben Überblick darüber, wie dein aktueller Branch zum master-Branch aussieht, und gibt dir damit unter anderem ein erstes Indiz, ob es eventuell zu Problemen kommen könnte, wenn die beiden Branches zusammengeführt werden sollen, z. B. bei Commit E zu Commit C.

Wir haben jetzt zwei Branches, und einen davon haben wir etwas weiterentwickelt. Wenn wir diese Weiterentwicklung für gut befunden haben, sollten wir sie in unseren Standard-Branch zurücküberführen. Wie der erste Schritt dafür aussieht, schauen wir uns im nächsten Abschnitt an.

## Pull-Request – Änderungen in Branches aufzeigen

Der Pull-Request ist vom Namen her ein eigenartiges Konstrukt, oder?<sup>6</sup> Ich hatte ihn weiter oben mit »Aufforderung, sich eine Veränderung zu holen« übersetzt, und genau das ist der erste Schritt, um unsere Änderungen aus dem vorherigen Abschnitt wieder in den Standard-Branch zu überführen, d.h., wir fordern jetzt jemanden dazu auf, unsere Veränderung zu holen.

Vielleicht fragst du dich gerade, wer genau wen auffordert, die Veränderung zu holen? Falls du (noch) allein auf deinem Projekt unterwegs bist bzw. du die Änderung in deinem Projekt durchgeführt hast<sup>7</sup>, forderst du dich gerade selber dazu auf. Merkwürdig? Ganz sicher ergibt dieses Vorgehen aber mehr Sinn, wenn dir klar wird, welche Rollen das eigentlich sind: Die Maintainerin fordert die Projekteigenerin dazu auf,<sup>8</sup> eine Veränderung zu übernehmen, und in deinem Fall sind diese Rollen zufällig mit ein und derselben Person besetzt.<sup>9</sup>

<sup>6</sup> GitLab nennt den gleichen Vorgang übrigens »Merge-Request«, etwas eingängiger, wie ich finde.

<sup>7</sup> und niemand anderes entsprechende Berechtigungen hat

<sup>8</sup> Es könnte auch umgekehrt sein.

<sup>9</sup> Warum ist das nicht die Rolle Contributor? Wir haben sie definiert als jemanden, der keine Rechte auf unserem Repository hat, er kann also auch keine Branches anlegen.



### Tipp: Den ersten Pull-Request herausfinden

Um herauszufinden, was der erste Pull-Request einer Person war, gibt es die Seite <https://firstpr.me>. Dort gibst du den GitHub-Accountnamen ein und bekommst alle Infos zum entsprechenden Pull-Request, wie Titel, Datum und Status.

Das Gute an diesem Vorgehen ist, dass die Veränderung erst einmal von einer anderen Person begutachtet und daraufhin überprüft werden kann, ob beispielsweise Standards eingehalten wurden (»Quellcode immer so und so dokumentieren«, »Bei Gedichten beginnt jede Zeile mit Großbuchstaben«) oder Ähnliches. Und noch viel schöner: Man kann auf einem Pull-Request auch einige automatisierte Prüfungen laufen lassen, die einem die Arbeit erheblich erleichtern.<sup>10</sup> Wenn ein Pull-Request erstellt wurde, kann darüber auch erst einmal diskutiert werden, wie wir im Beispiel mit dem Kind und dem Kostüm schon gesehen haben (»Das sind zu viele Sterne, wenn du fünf entfernst, würde es mir besser gefallen«). Hastest du – wie vorher auch – »einfach so« auf den master-Branch committest, hättest du all diese Möglichkeiten nicht.

Wir erstellen jetzt als Maintainerin den Pull-Request, der die Projekteignerin auf unsere Änderungen im Feature-Branch aufmerksam machen wird. Am einfachsten ist das, wenn du entweder die oben genannte Handlungsoption auswählst (siehe beispielsweise Abbildung 4-9) oder wenn du auf dem Feature-Branch bist und den Button *Pull request* auswählst (siehe Abbildung 4-14).



Abbildung 4-14: Das Anlegen eines Pull-Requests auf einem Feature-Branch geht auch über diesen Button.

Beide Male landest du in derselben Maske (siehe Abbildung 4-15). Solltest du dich etwas verheddert haben mit den ganzen Branches, hast du hier die Möglichkeit, über die Drop-down-Felder die richtigen Branches auszuwählen ①. Links sollte dabei unser master-Branch stehen, da wir ja in ihn hineinmergen wollen, beachte den kleinen Pfeil ②, der uns die Richtung weist. Erfreulicherweise informiert uns der grüne Haken und das *Able to merge* ③ darüber, dass der Pull-Request ohne Probleme in den master-Branch gemerget werden kann. Es kann auch passieren, dass es Konflikte gibt, die du erst auflösen musst, bevor der Pull-Request gemerget werden kann (sogenannte Merge-Konflikte). Wie das geht, schauen wir uns im Abschnitt »Merge-Konflikte lösen« auf Seite 186 in Kapitel 8 an.

<sup>10</sup> In Kapitel 9 schauen wir uns ein paar Möglichkeiten dazu an.

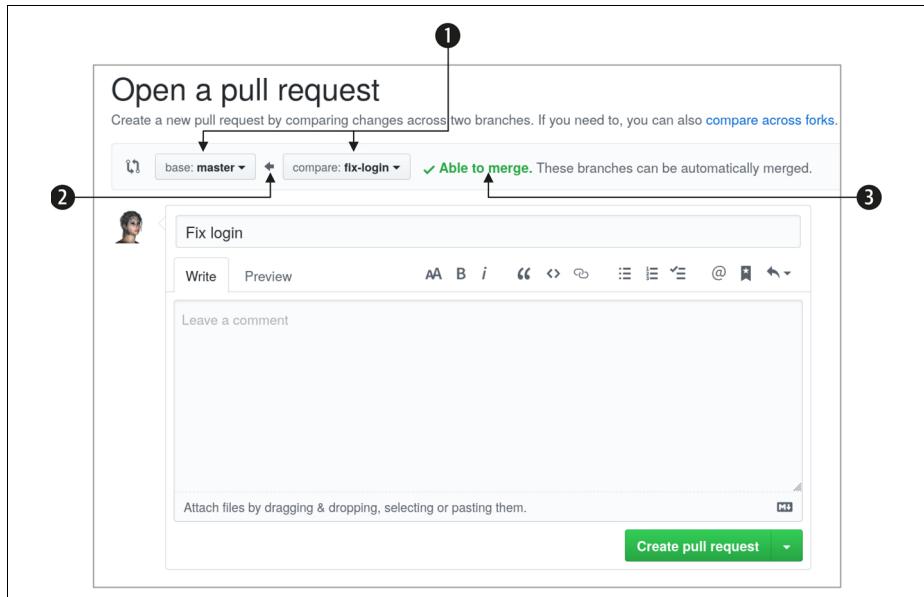


Abbildung 4-15: Für einen Pull-Request ist wichtig zu wissen, von wo nach wo etwas gemerget werden soll.

Gib dem Pull-Request einen sinnvollen Namen und, wenn du möchtest, einen Kommentar im Textfeld darunter. Solange du dich allein auf deinem »Territorium« bewegst, kannst du den Kommentar auch weglassen. Sobald du aber mit anderen arbeitest, empfehle ich dir, hier kurz zu beschreiben, was genau dieser Pull-Request ändern soll. Sobald du auf *Create pull request* geklickt hast, wird er im Repository-Menü unter den Pull-Requests auftauchen und kann von dort aus weiterbearbeitet werden.

Wenn du dir die Details zum Pull-Request anschaugst, könnte das aussehen wie in Abbildung 4-16. Es gibt, neben dem aktuellen Status des Pull-Requests und welcher Branch hier in welchen gemerget werden soll ①, auch ein Menü für weitere Details zum Pull-Request ②. In meinem Beispiel siehst du, dass ich zwei Commits in diesem Pull-Request habe ③ – in deinem Pull-Request können das auch mehr oder weniger sein. Die Information zu *Continuous integration* (deutsch »Kontinuierliche Integration«) ④ ist für uns erst einmal nicht interessant, das sehen wir uns später in Kapitel 9 noch genauer an. Unser Branch ist konfliktfrei und könnte ohne Probleme gemerget werden ⑤ – dies haben wir schon beim Erstellen gesehen. In der Zwischenzeit könnten aber weitere Änderungen beispielsweise am Standard-Branch stattgefunden haben, weshalb GitHub die Mergbarkeit regelmäßig prüft. Zu guter Letzt bietet uns der Pull-Request auch gleich eine entsprechende Handlungsoption an ⑥.

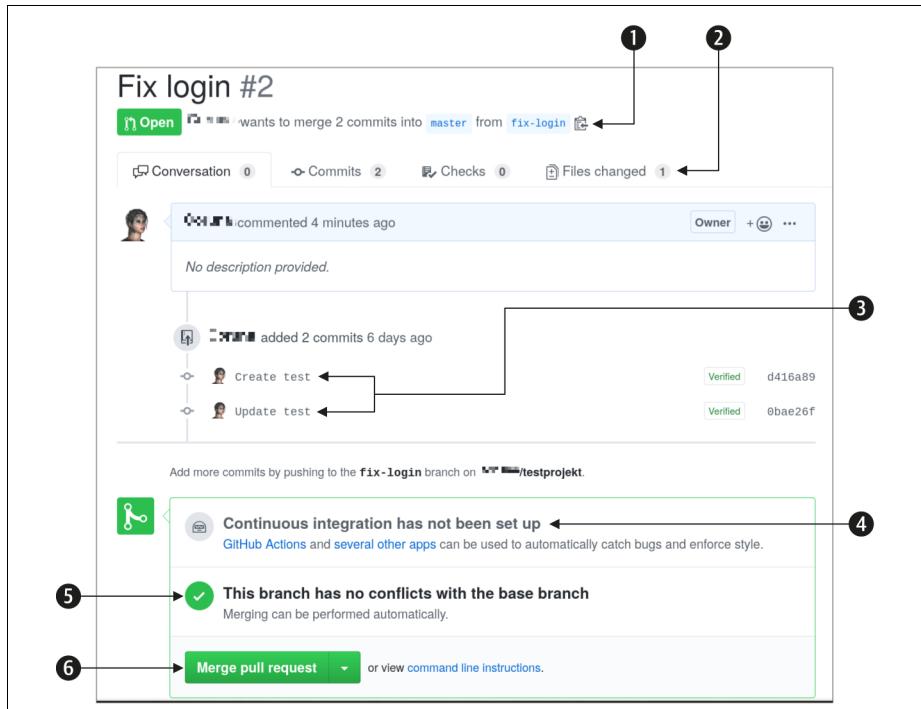


Abbildung 4-16: Ein Pull-Request bietet reichlich Informationen.

## Continuous Integration

Bei der *Continuous Integration*, der »kontinuierlichen Integration«, handelt sich um ein Konzept aus dem Bereich der Softwareentwicklung, um Software im Team reibungsloser entwickeln zu können. Du findest den Begriff oft im Zusammenhang mit den Begriffen *Continuous Delivery* (deutsch »Kontinuierliche Auslieferung«) und *Continuous Deployment* (deutsch »Kontinuierliche Bereitstellung«), die zwei sehr ähnliche Konzepte für die Auslieferung und Veröffentlichung von Software beschreiben.

Continuous Integration im Zusammenhang mit einem der beiden Auslieferungskonzepte wird häufig mit dem Kürzel CI/CD abgekürzt – je nachdem, für was du dich entschieden hast, steht das »D« für Deployment oder Delivery. Im Detail bedeuten die Begriffe:

- **Continuous Integration:** Entwicklerinnen und Entwickler bringen ihren Code möglichst früh und oft in ein gemeinsames Repository, einen gemeinsamen Branch oder einfach nur zusammen. Durch automatisierte Tests können frühzeitig Integrationsfehler entdeckt werden.
- **Continuous Delivery:** Codeänderungen werden kontinuierlich in produktivnahen Umgebungen bereitgestellt. Man könnte jederzeit die neue Version der Software ausliefern. Durch die kontinuierliche Bereitstellung der Software wer-

den Reibungspunkte minimiert, die mit den Bereitstellungs- oder Freigabeprozessen verbunden sind.

- **Continuous Deployment:** Geht einen Schritt weiter als Continuous Delivery und hat einen höheren Grad an Automatisierung. Codeänderungen werden nicht mehr in produktivnahen Umgebungen, sondern direkt in der produktiven Umgebung bereitgestellt.

Ich möchte wieder ein Bild bemühen, um das etwas nachvollziehbarer zu gestalten. Stell dir vor, Software zu programmieren, wäre so etwas wie, ein großes Weihnachtspaket für weit entfernt wohnende Verwandte packen. Alle aus der Familie wollen und dürfen Geschenke in das eine Paket packen.

- **Continuous Integration:** Jedes Mal, wenn ein Familienmitglied ein Geschenk in das Paket packt (den eigenen Code in das gemeinsame Repository speichert), wird (automatisch) überprüft, ob beispielsweise der Deckel noch zugeht und ob das Paketgewicht noch unter einem bestimmten Wert liegt (sonst wird das Paket eventuell zu teuer).
- **Continuous Delivery:** Regelmäßig wird das Paket »auslieferungsfertig« gemacht, indem es zugeklebt, mit einem Adressaufkleber versehen und probehalber vor die Tür der Gartenhütte gestellt wird, um den Auslieferungsprozess bis zu diesem Punkt durchzuspielen und zu testen.
- **Continuous Deployment:** Wie bei Continuous Delivery, nur dass das Paket dieses Mal direkt an die entfernten Verwandten geliefert wird.

Ob sich jemand für CI/CD und für welches »D« genau entscheidet, hängt wie immer vom Projekt und dem Projektkontext ab. Eine Firma zum Beispiel, die eher vorsichtig mit Veröffentlichungen ist, wird vermutlich nicht Continuous Deployment als Konzept wählen.

CI lässt sich auch auf GitHub nutzen. Dafür gibt es mehrere Apps im GitHub Marketplace (Apps und den Marketplace lernst du in Kapitel 9 noch genauer kennen). Zwei der bekannteren Apps sind *Travis CI*<sup>11</sup> und *CircleCI*<sup>12</sup>. GitHub bietet mit Actions auch die Möglichkeit, eigene CI-Anwendungen zu entwickeln. Dies lernst du ebenfalls in Kapitel 9 noch kennen.

Bevor wir in die Rolle der Projektleiterin schlüpfen und den Pull-Request weiterbearbeiten, schlage ich vor, dass du eine Sache ausprobierst:

1. Geh zurück auf die Startseite deines Projekts.
2. Achte darauf, dass du auf deinem Feature-Branch bist (bei mir *fix-login*).
3. Führe eine weitere Änderung durch (lege beispielsweise eine neue Datei an oder passe eine bestehende an).
4. Committe diese Änderung auf deinen Feature-Branch.

11 <https://github.com/marketplace/travis-ci>

12 <https://github.com/marketplace/circleci>

Wenn du jetzt noch einmal zu deinem Pull-Request von eben wechselst, fällt dir vielleicht auf, dass dort plötzlich ein Commit mehr ist als noch vor wenigen Minuten. Beispielsweise habe ich meinen zweiten Commit in Abbildung 4-16 gemacht, nachdem ich den Pull-Request bereits angelegt hatte. Ein Pull-Request ist also kein eingefrorener Zustand, in dem ich ein fest verschnürtes Paket jemandem vor die Tür stelle. Dieses Paket wird immer weiter befüllt, solange der Pull-Request noch offen ist und ich auf dem entsprechenden Branch munter weitercommitte.<sup>13</sup>

Woran liegt das? Wenn du dir in Abbildung 4-16 den Pull-Request noch mal anschaugst, siehst du in ①, welche beiden Branches betroffen sind (hier *master* zu *fix-login*). GitHub macht jedes Mal beim Auswählen des Pull-Requests einen Abgleich der getätigten Commits zwischen diesen beiden Branches und nicht zwischen einem verschnürten Paket und einem Branch. Abbildung 4-17 versucht, dies zu verdeutlichen: Sobald ein Pull-Request ausgewählt wird (Schritt 1), erfolgt ein Abgleich der betroffenen Branches (Schritt 2). *Commit1* und *Commit2* sind die ursprünglichen Commits auf dem Pull-Request. Wenn jetzt noch *Commit3* auf dem entsprechenden Branch ausgeführt wird, ist er ebenfalls Teil des Pull-Requests und wird entsprechend dargestellt (Schritt ③) – unabhängig davon, wann der Pull-Request erstellt wurde. Deswegen tauchen neue Commits auf unserem Feature-Branch ebenfalls im Pull-Request auf.

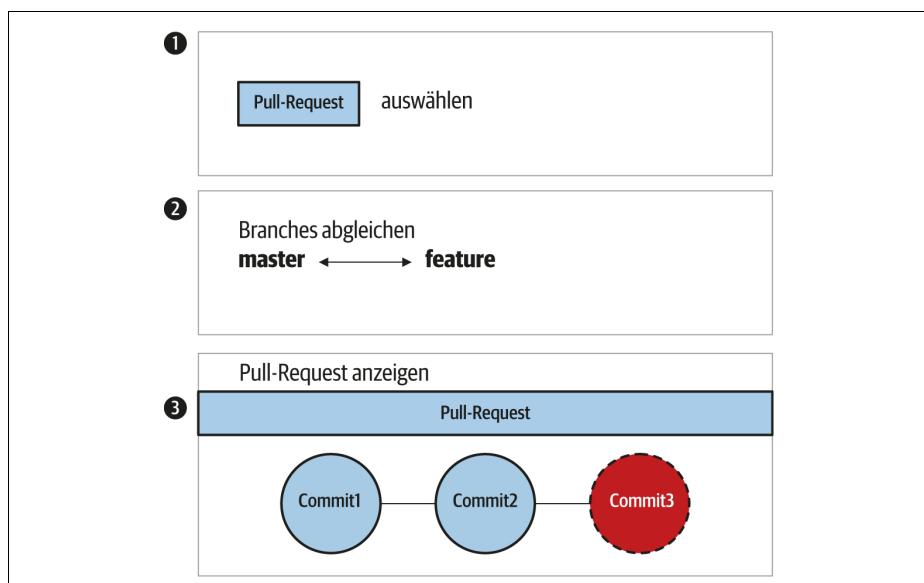


Abbildung 4-17: Beim Auswählen eines Pull-Requests erfolgt der Abgleich der beiden betroffenen Branches, spätere Commits (Commit3) auf einem Branch sind dann ebenfalls Teil des Pull-Requests.

13 Ich stelle mir dabei immer eine Art Schlauch vor, der an dem Paket befestigt ist und auch noch nach Ablieferung Sachen ins Paket »pusten« kann.

Das ist ähnlich wie die Metrik zu *commits ahead*, die du im Abschnitt »Änderungen auf einem Branch vornehmen« auf Seite 48 bereits kennengelernt hast.

GitHub gibt uns in Abbildung 4-16 zwischen ❸ und ❹ über dieses Verhalten auch Auskunft: *Add more commits by pushing to the fix-login branch on user/testprojekt*. – zu Deutsch etwa »Füge weitere Commits hinzu, indem du weitere Änderungen auf dem fix-login-Branch des Projekts user/testprojekt machst.«).



#### Tipp: Weiterarbeiten bei offenen Pull-Requests

Wenn du einen Pull-Request erstellst und auf dem entsprechenden Branch weitercommittest, landen diese Commits ebenfalls im offenen Pull-Request, und zwar so lange, bis der Pull-Request gemerget bzw. geschlossen ist.

Möchtest du weiterarbeiten, ohne den Pull-Request noch mehr aufzublähen, solltest du einen neuen Branch aufmachen. Beispielsweise könnte der Branch von deinem bisherigen Feature-Branch abzweigen, falls du Daten aus dem Branch zum Weiterarbeiten benötigst.

Wir – in der Rolle einer Maintainerin – haben jetzt der Projekteignerin in Form eines Pull-Requests zu erkennen gegeben, dass wir Änderungen in einem Branch vorgenommen haben. Mit dem Pull-Request bitten wir nun darum, sich unsere Änderungen anzuschauen und diese bestenfalls in den master-Branch zu übernehmen. Daher stammt auch der Begriff »Pull« (deutsch »ziehen«): Die Projekteignerin soll sich die Änderungen »ziehen«.

## Merge – Änderungen aus Pull-Requests übernehmen

Waren wir im vorherigen Abschnitt die Maintainerin, so wechseln wir jetzt in die Rolle der Projekteignerin, die einen neuen Pull-Request in ihrem Projekt bekommen hat. Zum Glück müssen wir uns dafür nicht neu einloggen, da beide Rollen von ein und derselben Person übernommen werden.

Mergen an sich ist erst einmal ziemlich einfach. Wähle den zu mergenden Pull-Request aus und klicke *Merge pull request* an, wie in Abbildung 4-16 zu sehen. Du wirst danach noch einmal aufgefordert, den Merge zu bestätigen und – tada – der Merge ist vollbracht und der Pull-Request wird automatisch geschlossen. Nach dem erfolgreichen Mergen kannst du den dazugehörigen Branch löschen (siehe hierzu auch Abbildung 4-18). Grundsätzlich ist es eine gute Idee, solche Branches zu löschen, da sie zum einen ihren Zweck erfüllt haben und damit überflüssig sind. Zum anderen kann ein Repository auch schnell unübersichtlich werden, wenn es viele – insbesondere viele unnötige – Branches beinhaltet.

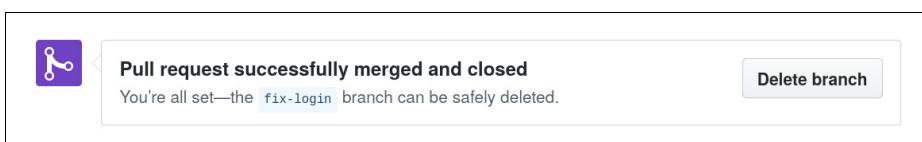


Abbildung 4-18: Nach einem Merge kann der dazugehörigen Branch gelöscht werden.

## Branches im Blick behalten – der Network Graph

Wenn du das Gefühl hast, den Überblick über deine Branches und Merges zu verlieren, hilft ein Blick in den *Network graph* (deutsch »Netzwerk-Graph«), den du in deinem Repository unter *Insights* (deutsch »Einblicke«) und *Network* findest (siehe auch Abbildung 4-19).

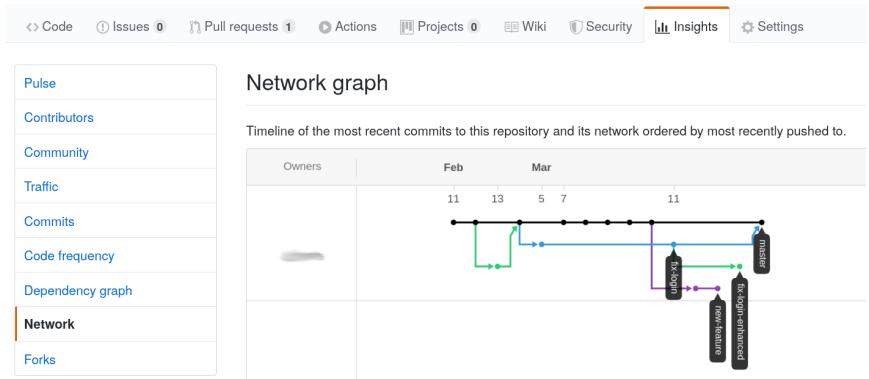


Abbildung 4-19: Unter Network kann man sich die aktuelle Branchsituation ansehen.

In meinem Beispiel gab es im Februar einen Feature-Branch mit einem Commit, der wenige Tage später gemerget wurde. Da der Branch danach gelöscht wurde, taucht hier auch kein Name auf. Viel interessanter ist die Situation danach. Direkt im Anschluss an den Merge vorher wird ein neuer Branch namens *fix-login* erstellt, zwei Commits werden darauf ausgeführt und schlussendlich in *master* gemerget. Hier steht noch der Name, was bedeutet, dass der Branch bisher nicht gelöscht wurde.

Abgehend von *fix-login* ist ein neuer, bisher ungemergter Branch namens *fix-login-enhanced* mit einem Commit entstanden. Das ist die Situation, die ich vorhin beschrieben habe. Du hast beispielsweise einen offenen Pull-Request zu *fix-login* am Start und möchtest noch weiter an dem Thema arbeiten, ohne dass diese weiteren Änderungen in den Pull-Request kommen. Dafür ist der Branch *fix-login-enhanced* entstanden, auf dem du jetzt »Pull-Request-frei« arbeiten kannst.

Parallel zu *fix-login* haben wir noch einen ungemergten Branch namens *new-feature* mit zwei Commits, der die parallele Entwicklung an einem weiteren Feature repräsentiert.

Jetzt haben wir einmal komplett den zweiten Ablauf durchgespielt. Du kannst Branches und Pull-Requests anlegen und weißt, wie man Pull-Requests mergt. Im realen Leben würde man Pull-Requests nicht einfach unbesehen mergen, weswegen wir uns im nächsten Abschnitt damit beschäftigen, welche Werkzeuge es dafür gibt.

## Standardarbeitsablauf für den GitHub Flow

Eine Kurzzusammenfassung dessen, was du für den GitHub Flow machen musst:

1. Branch anlegen.
2. Auf Branch (gegebenenfalls mehrmals) committen.
3. Pull-Request erstellen (Achtung, weitere Commits auf dem Branch verändern den Pull-Request entsprechend).
4. Pull-Request mergen.
5. Gegebenenfalls Branch löschen.

## Reviews durchführen

Im vorherigen Abschnitt haben wir den Pull-Request »einfach so« übernommen, ohne uns darum zu kümmern, ob die Änderung überhaupt gut oder gar sinnvoll ist. In einem realen Projekt würde man so etwas natürlich nicht tun. Du würdest dir mindestens anschauen wollen, was deine Maintainerin gemacht hat. Das nennt man einen »Review« (deutsch »Überprüfung«) durchführen. GitHub bietet uns eine Reihe von Werkzeugen an, um einen solchen Review durchführen zu können. Diese schauen wir uns jetzt mal an.

Um das auszuprobieren, benötigen wir einen neuen Pull-Request und dazu wieder einen von *master* separaten Branch, der am besten ein paar inhaltliche Änderungen in einer Datei hat (z.B. zwei neue Zeilen, siehe zur Erinnerung den Kasten »Standardarbeitsablauf für den GitHub Flow«). Wir werden diesen Pull-Request reviewen, indem wir im Pull-Request das Register *Files changed* (deutsch »veränderte Dateien«) anklicken (siehe Abbildung 4-20).



Abbildung 4-20: Der Start eines Reviews beginnt bei Files changed.

Hier gibt es gleich eine ganze Menge zu sehen (siehe Abbildung 4-21). Bei ❶ sind die beiden Zeilen zu sehen, die ich in diesem Pull-Request gerne gemerkt haben möchte (# *new fresh feature* und *lately added*). Das kleine Pluszeichen vor der Zeile ❷ bedeutet, dass dies eine Zeile ist, die hinzugefügt werden soll. Zeilen, die entfernt werden sollen, bekommen ein Minuszeichen. GitHub hinterlegt die Zeilen auch farblich entsprechend in Grün (hinzugefügt) und Rot (entfernt).

Die beiden Zeilen liegen übrigens deswegen so weit auseinander, weil ich bereits auf das (blaue) Kreuz bei ❸ geklickt habe, wodurch sich das Textfeld darunter zum Kommentieren aufklappte. Dieses Kreuz siehst du erst, wenn du die Maus vor eine Textzeile bewegst. Du kannst auch mehrere Zeilen markieren, indem du die Maustaste gedrückt hältst und so lange nach unten ziehst, bis du alle gewünschten Zeilen markiert hast.

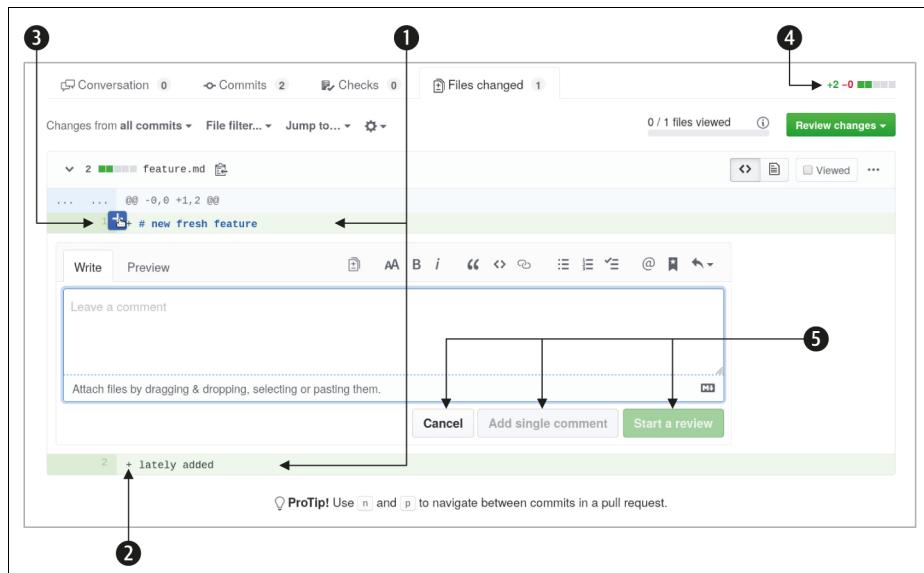


Abbildung 4-21: Um Änderungen in Pull-Requests zu reviewen, bietet GitHub einige Werkzeuge an.

Bei ❹ siehst du in einer Kurzübersicht, wie viel sich durch diesen Pull-Request ändern würde. In meinem Fall würden zwei Zeilen hinzukommen (+2) und keine Zeilen entfernt werden (-0). Die kleinen Kästchen rechts daneben drücken das Ganze noch mal grafisch aus. Bei dir kann das natürlich jetzt anders aussehen, je nachdem, was und wie viel du geändert hast.

Mit den Buttons in ❺ kannst du entweder den von mir bereits aufgeklappten Kommentar stornieren (*Cancel*) – falls du beispielsweise das blaue Kreuz versehentlich angeklickt hast –, oder du kannst einen Review starten (*Start a review*). Das bedeutet, solange du noch kommentierst, werden alle deine Kommentare mit *Pending* (deutsch »noch ausstehend«) markiert und sind erst einmal nur für dich sichtbar (siehe hierzu Abbildung 4-22 ❶). Erst wenn du den Review abschließt, werden alle Kommentare sichtbar für andere. Falls du nur einen einzigen Kommentar zu den Änderungen abgeben und nicht gleich einen ganzen Review starten möchtest, nimmst du den mittleren Button (*Add single comment*).

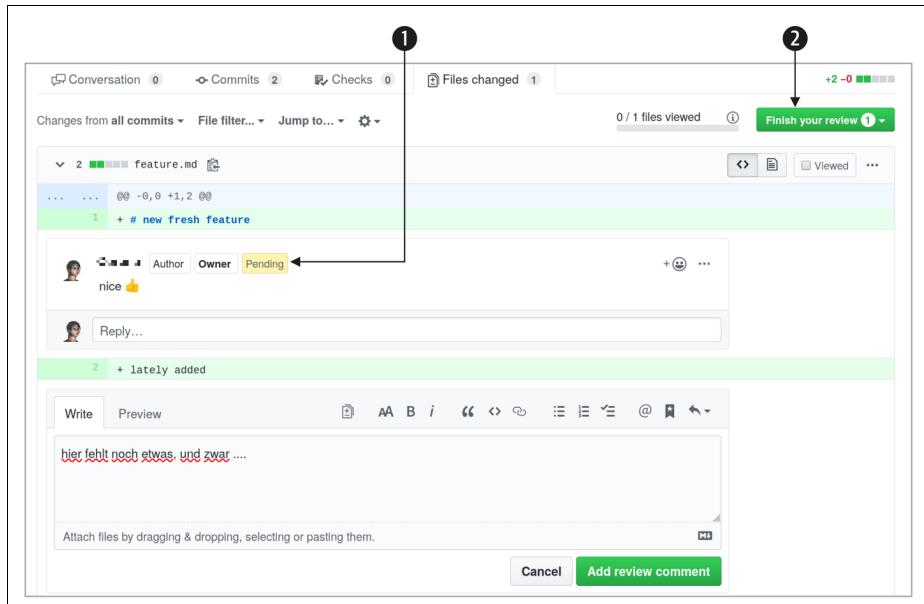


Abbildung 4-22: Ein Pull-Request mitten im Review-Prozess, Kommentare werden zunächst mit Pending markiert.

Falls du dich für einen Review und nicht für einen einzelnen Kommentar entschieden hast, kannst du ihn über den Button *Finish your review* (deutsch »Überprüfung beenden« ②) abschließen und damit deine Kommentare für alle sichtbar machen (siehe Abbildung 4-22). Die Zahl auf dem Button bedeutet übrigens die Anzahl der Kommentare, die du gemacht hast. Wenn du den Button anklickst, hast du mehrere Möglichkeiten, den Review zu beenden (siehe Abbildung 4-23). In meinem Beispiel kann ich nur *Comment* (deutsch »Kommentieren«) auswählen und damit ausschließlich ein Feedback ohne weitere prozessuale Auswirkungen hinterlassen. Das liegt daran, dass ich sowohl Erstellerin als auch Reviewerin bin. GitHub verhindert, dass ein und derselbe Account die anderen beiden Optionen auswählen kann. Im Abschnitt »Gutes schützen – Protected Branches« auf Seite 67 werden wir noch sehen, dass man als Schutzmechanismus eine bestimmte Anzahl an Reviews vorschreiben kann, bevor ein Pull-Request überhaupt gemerkt werden darf. Gäbe es die Möglichkeit, dass die Erstellerin auch gleichzeitig Reviewerin sein kann, könnte dieser Schutzmechanismus ausgehebelt werden, deswegen darf ich hier nur kommentieren.

Die anderen beiden Optionen werden primär für den bereits angesprochenen Schutzmechanismus verwendet. Die Option *Approve* (deutsch »freigeben« oder »genehmigen«) gibt den Pull-Request zum Mergen frei. Hat der Pull-Request die geforderte Anzahl an Genehmigungen, kann er gemerkt werden. Mit der Option *Request changes* (deutsch »Änderungen anfordern«) wird dagegen zunächst die Genehmigung verweigert, bis die angeforderten Änderungen adressiert worden sind.

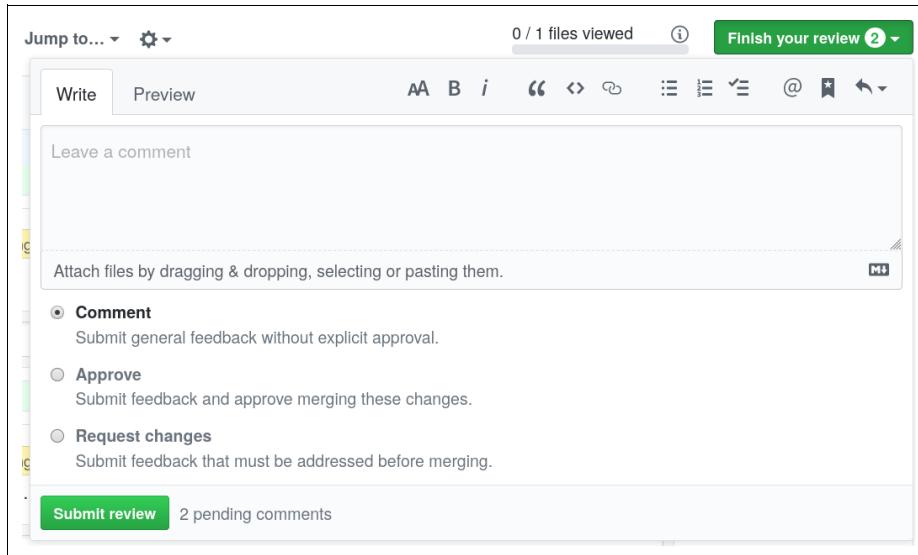


Abbildung 4-23: Um einen Review abzuschließen, gibt es mehrere Optionen. Ist es der eigene Pull-Request, ist nur kommentieren erlaubt.



#### Tipp: Fremde Pull-Requests anpassen

Manchmal ist es schneller und praktischer, während eines Reviews fehlende Kleinigkeiten in einem Pull-Request eben selbst anzupassen und nicht auf die ursprüngliche Entwicklerin zu warten (siehe auch Abbildung 4-24). Das geht relativ einfach, indem du im Pull-Request über *Files changed* ① die zu ändernde Datei anwählst und dann die drei Punkte ganz rechts neben dem Dateinamen anklickst ②. Dort kannst du *Edit file* (deutsch »Datei anpassen«) wählen ③, deine Änderungen vornehmen und mit einem Commit speichern. GitHub wird diesen Commit zum Pull-Request hinzufügen, und sobald dieser gemitigt ist, sind auch deine Änderungen aktiv.

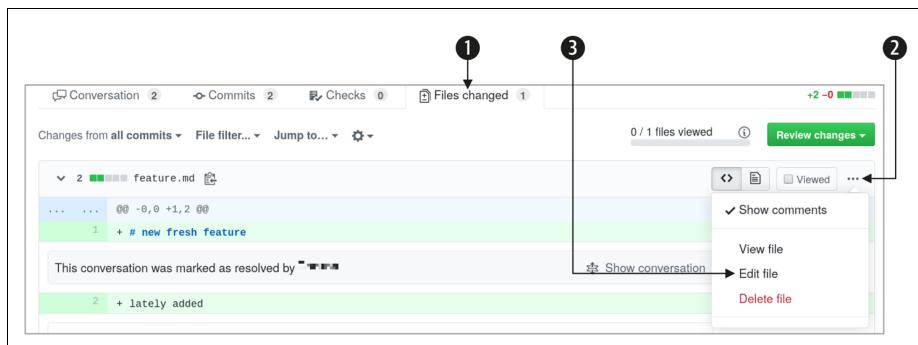


Abbildung 4-24: Fremde Pull-Requests lassen sich mit einem weiteren Commit anreichern.

## Reviewer manuell anfordern

Vielleicht möchtest du beim Erstellen eines Pull-Requests aber auch explizit einen speziellen Reviewer anfordern, der deine Änderungen unter die Lupe nimmt? Auch das geht mit GitHub relativ einfach (siehe Abbildung 4-25). Erstelle einen Pull-Request und klicke auf der rechten Seite auf *Reviewers* ①. Es erscheint dann ein Such- und Eingabefeld mit einer Liste an möglichen Accounts, die du als Reviewer auswählen kannst ②. Du kannst nur Accounts als Reviewer auswählen, die entweder Projekteigner oder Maintainer des Projekts sind.

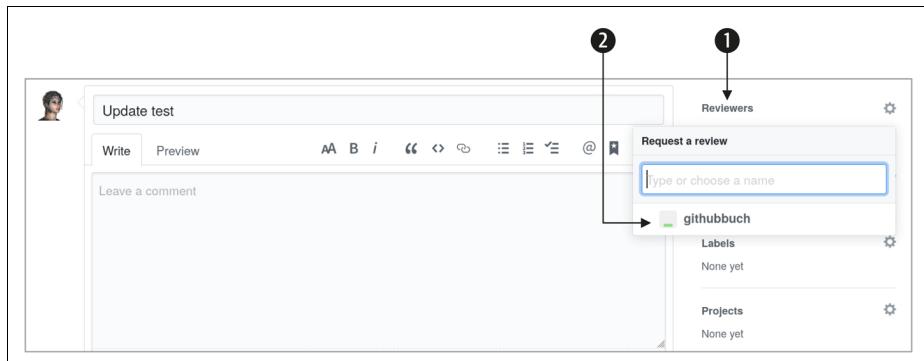


Abbildung 4-25: Beim Erstellen eines Pull-Requests kann man auch explizit Reviewer anfordern, diese werden dann per Mail informiert.

Das Ganze kannst du auch automatisieren. Das schauen wir uns im nächsten Abschnitt an.

## Reviews automatisch zuweisen – CODEOWNERS

In Projekten kommt es häufig vor, dass die Verantwortung für bestimmte Bereiche auf unterschiedliche Personen aufgeteilt wird. So könnte beispielsweise bei einer App die Entwicklerin verantwortlich für die Funktionalitäten der App sein und der Designer für deren Grafiken und Layout. GitHub bietet einen einfachen Weg an, um bei einem Pull-Request automatisch die richtigen Verantwortlichen für einen Review heranzuziehen. Das Ganze funktioniert mit einer Datei namens *CODEOWNERS* (deutsch »Codeeigentümer«), und die richten wir in diesem Abschnitt ein.

In deinem Projekt – entweder im Hauptverzeichnis oder unter *docs* oder *.github*<sup>14</sup> – legst du eine Datei namens *CODEOWNERS* an. Die Datei wird abhängig vom Branch angelegt, d.h., du kannst für ein und dieselbe Datei auf dem master-Branch andere Verantwortlichkeiten festlegen als auf einem Feature-Branch. Wir wählen erst einmal den master-Branch aus.

14 Diese Ablageorte kennen wir von der *README.md*.



### Tipp: Ablageort für Dateien mit Spezialfunktion

Viele Dateien mit Spezialfunktionen wie `README.md` oder `CODE OWNERS` können entweder im Hauptverzeichnis, im Verzeichnis `docs` oder unter `.github` abgelegt werden. Im Abschnitt »Dein Projekt anschaulich beschreiben« auf Seite 104 in Kapitel 6 sehen wir noch mehr solcher Dateien.

Der Aufbau der Datei `CODEOWNERS` ist relativ simpel. Kommentare werden mit einer Raute eingeleitet und dienen häufig der Strukturierung und Dokumentation. Als Erstes steht der Teil, der einem *Owner* zugewiesen werden soll, z.B. eine oder mehrere Dateien, Dateiendungen oder Verzeichnisse. Mit einem Leerzeichen getrennt werden danach die Personen (bzw. Accounts) notiert, die dafür verantwortlich sind. Mehrere Personen trennst du ebenfalls mit einem Leerzeichen. Hier mal als Beispiel eine Datei:

```
# Ich bin ein Kommentar

# Zwei Standard-Code-Owner für alles
# man beachte das * am Anfang der Zeile
* @projekteignerin @superwoman

# Zuweisen einer einzelnen Datei
README.md @schlumpfine

# Falls nur Dateien mit der Endung .html in einem Pull-Request auftauchen,
# bekommt dieser User den Aufruf (aber nicht unsere Standard-Owner oben).
# Es können auch die bei GitHub verwendeten Mailadressen genutzt werden.
*.html webmaster@website.de

# Für jede Datei und jedes Unterverzeichnis
# im Verzeichnis "website" ist der User Eigentümer,
# egal wo sich dieses Verzeichnis befindet.
website/ @webmaster

# Für jede Datei und jedes Unterverzeichnis
# im Verzeichnis "website", das direkt
# im root-Verzeichnis ist, ist der User Eigentümer,
# aber nicht /doc/website oder /doc/sub/website.
/websit/ @webmaster
```

**Wichtig zu wissen:** Die Reihenfolge zählt. Alles, was weiter unten in der Datei steht, kann alles, was weiter oben steht, überschreiben. So könnte nachfolgendes Beispiel `Schlumpfine` davon abhalten, dass sie etwas zu tun bekommt:

```
# Reihenfolge ist wichtig, Einträge weiter unten
# überschreiben die Einträge weiter oben.
# Zuweisen einer einzelnen Datei.
README.md @schlumpfine

# Zuweisen einer Dateiendung - überschreibt die obige.
*.md @schlaubi_schlumpf
```

Sobald jetzt ein Pull-Request angelegt wird, auf den eine der oben definierten Regeln zutrifft, wird die verantwortliche Person in *CODEOWNERS* automatisch als Reviewer zugewiesen und bekommt per Mail eine entsprechende Benachrichtigung (siehe auch Abbildung 4-26).

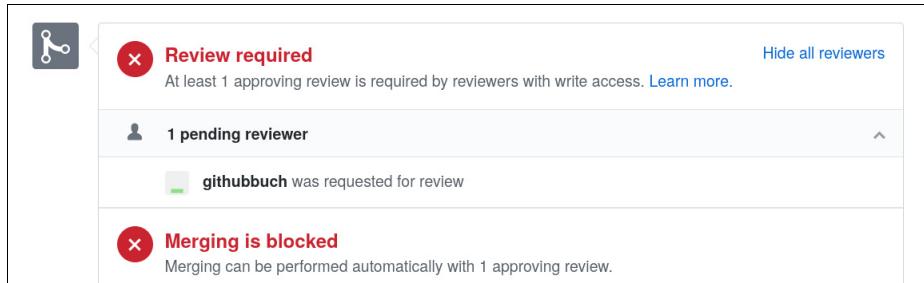


Abbildung 4-26: Mit der Datei *CODEOWNERS* können Reviewer automatisch Pull-Requests zugewiesen werden.

## Gutes schützen – Protected Branches

Standardmäßig kann jeder Pull-Request jederzeit gemergt werden – die entsprechenden Berechtigungen vorausgesetzt. Es sei denn, es gibt einen Konflikt beim Mergen.

Manchmal ist es aber wünschenswert, wenn noch ein Zweiter (oder Dritter oder Vierter) mit draufschaut,<sup>15</sup> bevor irgendein Unheil entsteht. Häufig will jemand nur »mal eben den Code aufräumen« und löst damit eine ungewollte Kettenreaktion aus, weil er eine Abhängigkeit übersehen hat, und am Ende funktioniert gar nichts mehr.<sup>16</sup> So ein Unheil lässt sich im besten Fall mit viel Zeit und Aufwand beheben, im schlechtesten Fall macht es etwas unwiederbringlich kaputt (und sei es nur die Reputation ...).

GitHub bietet uns dazu einen Schutzmechanismus namens *Protected Branches* (deutsch etwa »geschützte Branches«) an, den ich bereits weiter oben schon erwähnt hatte. Protected Branches beschreiben ein Regelwerk, das man einem oder mehreren Branches zuweisen kann. Zum Beispiel könnte eine Regel für einen speziellen Branch lauten, dass die Commits signiert werden müssen oder dass »Pull-Requests erst dann in diesen Branch gemergt werden dürfen, wenn mindestens eine Maintainerin diesen reviewt hat«.

Das Letztere nennt man auch *Required Reviews* (deutsch »erforderliche Reviews«), und das wollen wir jetzt mal ausprobieren. Alle anderen Optionen für Protected

<sup>15</sup> Auch bekannt als »Mehraugenprinzip«.

<sup>16</sup> Ich habe im Kopf da immer das Bild vom Elefanten im Porzellanladen, vorne am Kopf ist er sehr vorsichtig, aber sein Hinterteil stößt ständig irgendwelche Regale um.

Branches sind eher für Fortgeschrittene gedacht, und daher werden wir uns diese im Detail nicht weiter anschauen.

## Genehmigung vorschreiben – Required Reviews

Gehe auf dein Repository und wähle dort unter *Settings* auf der linken Seite *Branches* aus. Unter der Überschrift *Branch protection rules* gibt es den Button *Add rule*, den wir anklicken und der uns zu den Einstellungsmöglichkeiten bringt (siehe Abbildung 4-27).

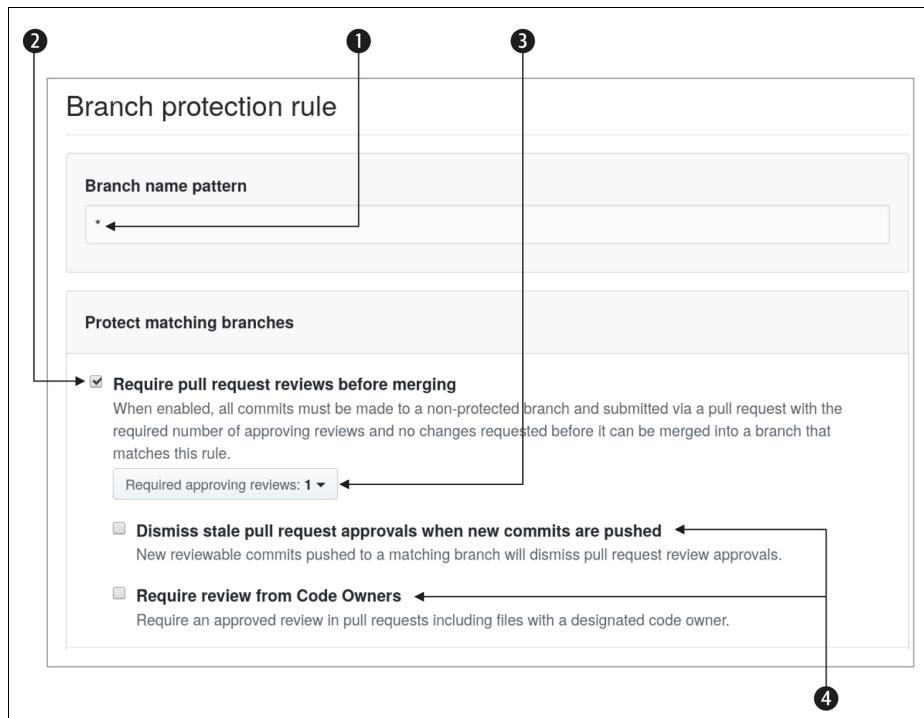


Abbildung 4-27: GitHub bietet für Protected Branches verschiedene Regeln an.

Bei *Branch name pattern* **1** (deutsch etwa »Muster für Branchnamen«) kannst du jetzt ein Muster eingeben, das alle Branches definiert, für die diese Regel gelten sollen. In der Abbildung habe ich ein \* als Platzhalter – eine sogenannte Wildcard – eingetragen, was »alles« bedeutet. Möchtest du beispielsweise, dass die Regel nur für Branches mit dem Wort »feature« im Namen gilt, müsste dort \*feature\* stehen (beachte die Sternchen vor und hinter dem Wort). Wir wählen, wie in der Abbildung, erst einmal alles.

Als konkrete Regel wählen wir die oberste aus **2** (*Require pull request reviews before merging*, deutsch etwa »Erfordert Pull-Request-Reviews vor dem Mergen«). Hier könnten wir noch die Anzahl der erforderlichen Reviews auswählen **3**. Wir

belassen die Zahl bei 1. Die beiden anderen Unteroptionen bei ④ schauen wir uns gleich noch an. Wir lassen sie daher erst einmal deaktiviert, wie auf dem Bild zu sehen. Wenn wir diese Regel in Kraft setzen, muss ab sofort jeder Pull-Request auf allen Branches mindestens einmal reviewt werden, bevor ein Merge erfolgen kann. Das probieren wir jetzt aus. Speichern nicht vergessen, also ganz nach unten scrollen und auf *Create* (deutsch »erstellen«) klicken! Du bekommst dann angezeigt, für wie viele Branches deine Regel aktuell gilt (siehe Abbildung 4-28).

**Branch protection rule**

**Branch name pattern**

- \*

Applies to **3** branches

- fix-login-enhanced
- master
- new-feature

Abbildung 4-28: GitHub zeigt uns an, welche Branches von einer Regel betroffen sind.

Nachdem wir alle unsere Branches mit der obigen Regel versehen haben, schauen wir uns an, welche Auswirkungen das auf einen neuen Pull-Request hat. Wenn du einen neuen Pull-Request anlegst, sollte es eine ähnliche Rückmeldung geben wie die in Abbildung 4-29, nämlich einen deutlichen Hinweis darauf, dass hier nicht »einfach so« gemerged werden kann (*Merging is blocked*, deutsch »das Mergen ist blockiert«), sondern noch mindestens ein Review notwendig ist (*Review required*, so haben wir es ja auch festgelegt).

**Review required**  
At least 1 approving review is required by reviewers with write access. [Learn more](#).

**Merging is blocked**  
Merging can be performed automatically with 1 approving review.

As an administrator, you may still merge this pull request. ←

**Merge pull request** or view command line instructions.

Abbildung 4-29: Aus Sicht der Projektleiterin: Die festgelegten Regeln greifen, sie kann trotzdem mergen.

Wie du auch siehst, hat die Projekteignerin (hier mit *administrator* bezeichnet) trotzdem noch die Möglichkeit, den Merge auch ohne Review durchzuführen. Schaut man sich denselben Pull-Request aus dem Blickwinkel einer Maintainerin an, entfällt diese Option (siehe Abbildung 4-30). Wenn du verhindern möchtest, dass die Projekteignerin (also du) die Regeln einfach so umgehen kann, aktiviere die Option *Include administrators* (deutsch »Administratoren einschließen«). Das ist im Menü aus Abbildung 4-27 weiter unten einstellbar. Jetzt kannst du nicht mehr aus Versehen den Merge durchführen, sondern musst dich ebenfalls an die Regeln halten.

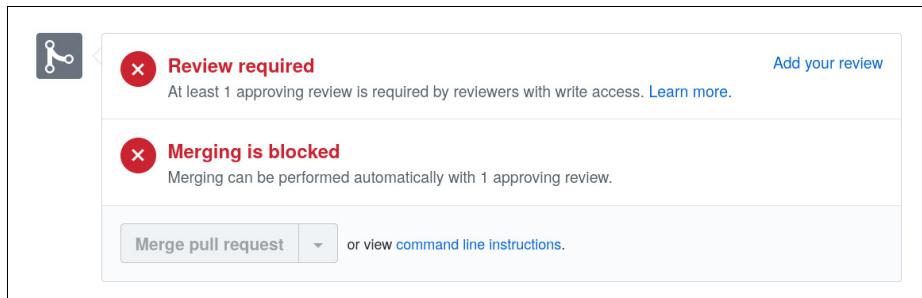


Abbildung 4-30: Aus Sicht der Maintainerin: Die festgelegten Regeln greifen, sie muss sich an die Regeln halten.

Wenn du dir jetzt die Übersicht der Pull-Requests anschaugst, siehst du, dass bei allen Pull-Requests auch der Status der geforderten Reviews dargestellt wird (siehe Abbildung 4-31). Du hast zudem die Möglichkeit, nach Review-Status zu filtern, was bei einer langen Liste an Pull-Requests praktisch ist.

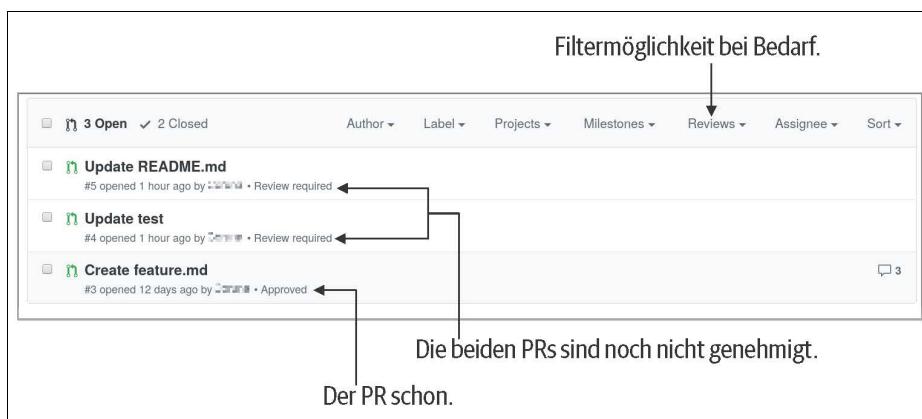


Abbildung 4-31: Den Status bei notwendigen Reviews kann man direkt in der Pull-Requests-Liste sehen.

Falls du den nächsten Prozessschritt selbst durchspielen möchtest, brauchst du jetzt einen zweiten Account. Aus meiner Sicht ist das aber nicht unbedingt notwendig. Für das Verständnis reicht es, wenn du dir die Abbildungen anschaust und meinen Erläuterungen folgst.



### Tipp: Mehrere Mailadressen

GitHub überprüft beim Anlegen eines neuen Accounts, ob eine Mailadresse auf der Plattform bereits registriert ist. Wenn du zwei oder mehr Accounts bei GitHub anlegen möchtest, aber dir nicht ständig neue Mailadressen zulegen willst, geht – je nach Mailanbieter – auch das Anlegen eines Alias.

Viele Mailanbieter bieten sogenannte *FunDomains* an, bei denen bis zu 20 neue Mailadressen angelegt werden können. Die Mails an diese FunDomains landen aber alle beim Hauptaccount.

In Abbildung 4-23 hatten wir gesehen, dass derselbe Account, der den Pull-Request anlegt, in einem Review nur kommentieren kann. Schauen wir mit einem anderen Account darauf, sieht die Welt anders aus (siehe Abbildung 4-32). Hier kann jetzt (endlich) der Pull-Request freigegeben (*Approve*) werden. Sobald die Anzahl der von uns festgelegten notwendigen Freigaben erreicht ist, ändert sich auch das Aussehen des Pull-Requests (siehe Abbildung 4-33). Jetzt kann er entweder von der Maintainerin oder der Projekteignerin gemerkt werden.

The screenshot shows a GitHub pull request review interface. At the top, there's a navigation bar with 'Jump to...', a gear icon, and a status message '0 / 1 files viewed'. To the right is a green 'Review changes' button. Below the bar, there's a text input field with the placeholder 'well done!'. Underneath the input field is a section for attachments with the text 'Attach files by dragging & dropping, selecting or pasting them.' followed by a file selection icon. At the bottom of the interface, there are three radio button options: 'Comment' (disabled), 'Approve' (selected), and 'Request changes'. Each option has a descriptive subtitle. Finally, at the very bottom is a large green 'Submit review' button.

Abbildung 4-32: Mit einem anderen Account funktioniert das Freigeben (*Approve*).

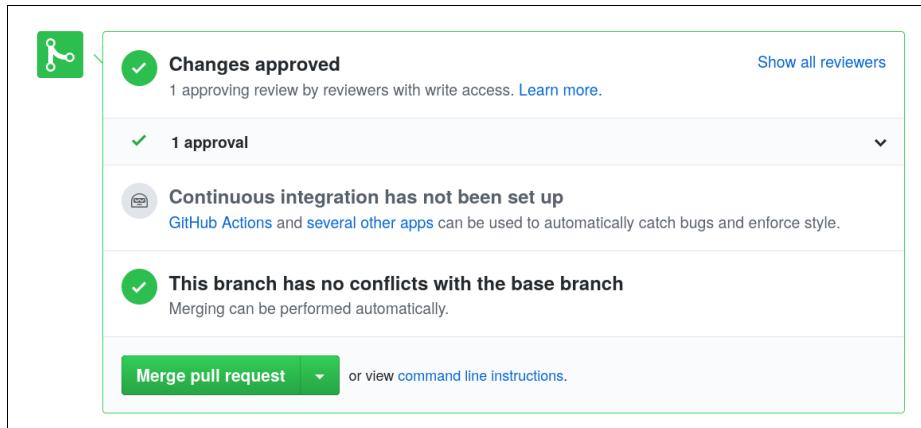


Abbildung 4-33: Der Pull-Request hat alle festgelegten Regeln erfüllt und kann jetzt gemerget werden.

In einem Review kannst du auch Veränderungen an dem bestehenden Pull-Request anfordern (siehe Button *Request changes* in Abbildung 4-32). Im entsprechenden Pull-Request wird das dann vermerkt, wie in Abbildung 4-34) gezeigt. Ist das der Fall, muss die Entwicklerin des Pull-Requests noch mal ran und die gewünschten Änderungen umsetzen.

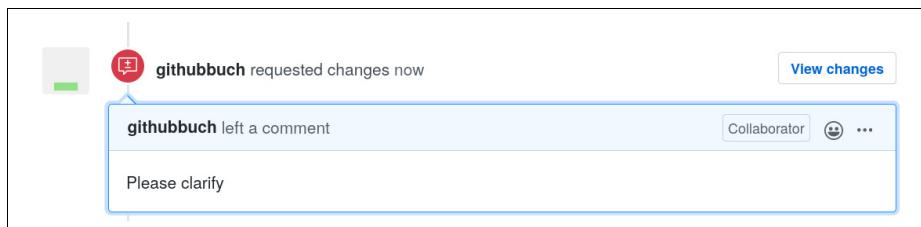


Abbildung 4-34: Eine gewünschte Änderung an einem Pull-Request

## Genehmigung automatisch zurückziehen

Eine interessante Sache, die du bei den Protected Branches noch einstellen kannst, ist die Regel *Dismiss stale pull request approvals when new commits are pushed* (deutsch etwa »Genehmigungen für Pull-Requests zurückziehen, wenn neue Commits auf dem Pull-Request erfolgen«). Du findest sie als Unterpunkt bei den Required Reviews (siehe Abbildung 4-27).

Wenn diese Regel nicht aktiviert ist, behalten Pull-Requests ihre Genehmigungen bei, selbst wenn im Nachgang noch weitere Commits hinzukommen – schau dir hierzu Abbildung 4-35 an. Commit1 und Commit2 sind Teil eines Pull-Requests, der genehmigt wurde. Commit3 kommt erst nach der Genehmigung dazu. Ohne *Dismiss stale pull request approvals ...* bleibt die Genehmigung erhalten (Pull-Re-

quest A), mit der Regel wird sie zurückgezogen, und der Pull-Request muss neu genehmigt werden (Pull-Request B).

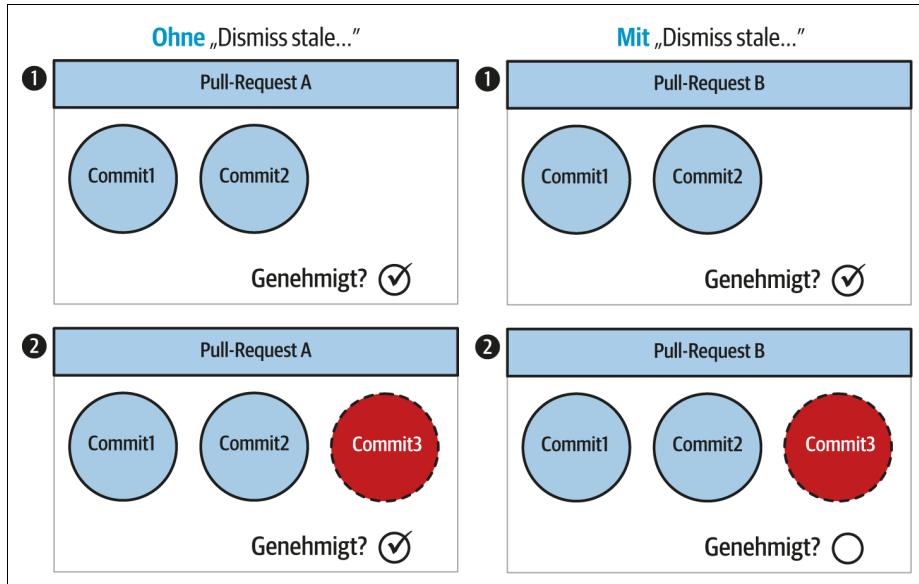


Abbildung 4-35: Einmal genehmigte Pull-Requests können weitere Commits bekommen und behalten ihre Genehmigung (Pull-Request A). Dismiss stale pull request approvals ... verhindert das (Pull-Request B).

Ist die Regel aktiv, sieht man direkt auf dem Pull-Request ihr Wirken (siehe Abbildung 4-36). Wenn du also verhindern möchtest, dass sich unerwünschte Commits über bereits genehmigte Pull-Requests »einschmuggeln« können, solltest du die Regel aktivieren.

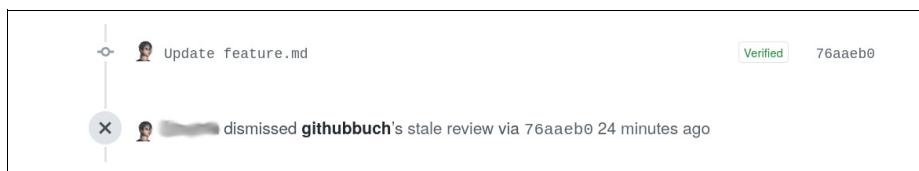


Abbildung 4-36: Dismiss stale pull request approvals ... ist aktiv und entfernt alle Genehmigungen.

## Genehmigung durch Eigentümer\*innen vorschreiben

Unsere letzte Option für Required Reviews ist *Require review from Code Owners* (deutsch »erfordert Überprüfung von Codeeigentümern«). Wir haben bereits im Abschnitt »Reviews automatisch zuweisen – CODEOWNERS« auf Seite 65 gesehen, wie wir Codeeigentümer als solche festlegen können.

Wenn wir jetzt mit geschützten Branches auch noch den Review durch einen Codeeigentümer einstellen, verschärfen wir die Einschränkungen (siehe hierzu zum Verständnis auch Abbildung 4-37). Ohne Einstellungen kann jeder Pull-Request »einfach so« gemerget werden. Mit der Datei `CODEOWNERS` können wir einen Review für einen Pull-Request automatisch einer Person zuweisen, der Pull-Request ist aber auch ohne diesen Review mergbar. Durch das Einrichten von geschützten Branches mit der Regel *Required Reviews* können wir vorgeben, dass eine Mindestanzahl an Personen (in unserem Beispiel eine Person) den Pull-Request reviewt, bevor dieser mergbar wird. Das muss aber nicht zwingend ein Codeeigentümer sein. Das »beheben« wir dadurch, dass wir schlussendlich die letzte Option einstellen, sodass Reviews erforderlich sind und diese auch noch vom entsprechenden Codeeigentümer durchgeführt werden.

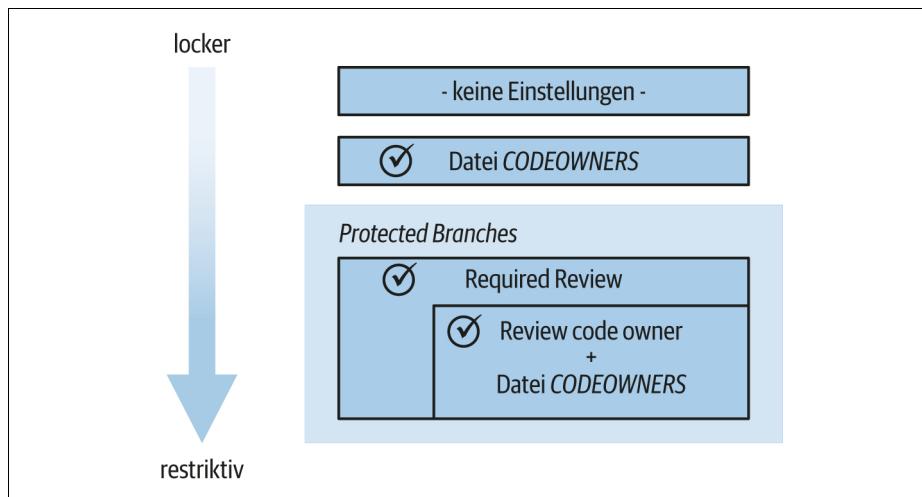


Abbildung 4-37: Die Voraussetzungen, um einen Pull-Request zu mergen, können durch verschiedene Einstellungen immer weiter verschärft werden.



#### Tipp: Löschung bei Protected Branches zulassen

Bei den Protected Branches kann es sinnvoll sein, noch die Option *Allow deletions* (deutsch »Löschen zulassen«) einzuschalten. Ansonsten ist es nicht möglich, geschützte Branches zu löschen. Insbesondere wenn du alle Branches schützt (z.B. über \*), kann das eine sinnvolle Einstellung sein.

Bist du die einzige Person, die Pull-Requests mergen kann, und ändert sich das auch auf absehbare Zeit erst einmal nicht, würde ich die Einschränkungen eher locker handhaben. Sobald du aber mindestens eine Maintainerin dabeihast, solltest du zumindest darüber nachdenken, ob und wie restriktiv das Mergen sein soll. Das hängt natürlich auch von der Art und dem Umfang des Projekts ab. Da Merges von GitHub nachvollziehbar dokumentiert und auch wieder rückgängig gemacht werden können (siehe Abbildung 4-38), ist eine lockere Einschränkung zu Beginn

durchaus sinnvoll. Verschärfen lassen sich die Einschränkungen später immer noch.

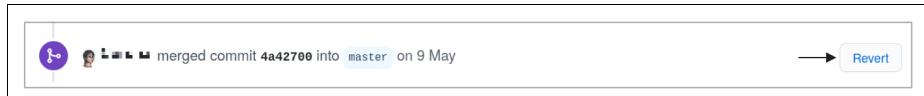


Abbildung 4-38: Der Merge eines Pull-Requests lässt sich wieder rückgängig machen.

Wir schauen uns jetzt noch an, mit welchen weiteren Mitteln du dein Repository organisieren kannst, um die Zusammenarbeit mit anderen möglichst gut zu gestalten.

## Den Laden sauber halten – Vorlagen, Diskussionen eingrenzen

Wenn du mit vielen Menschen am selben Projekt arbeitest, wirst du früher oder später feststellen, dass nicht jeder und jede die gleichen Qualitätsansprüche wie du selbst hat, was häufig zu Frust und Konflikten führen kann.

In diesem Abschnitt möchte ich dir daher zeigen, wie du mittels Vorlagen für Issues und Pull-Requests für eine gute einheitliche Qualität sorgst und wie du eine Diskussion, die sich schon etwas erhitzt hat, wieder (technisch) abkühlen lassen kannst.

### Vorlagen für Issues

Stell dir folgende Situation vor: Ein Contributor möchte einen Fehler mittels Issue melden, hat aber nur unzureichend Informationen beigesteuert.<sup>17</sup> Die Projekteigenerin kann aufgrund dieser fehlenden Informationen den Fehler nicht beheben und muss im schlimmsten Fall mehrmals nachfragen.<sup>18</sup> Das frustet beide Seiten und verzögert die Fehlerbehebung.

Durch das Erstellen einer Vorlage für Issues kannst du die Art und Weise, wie Issues erstellt werden, beeinflussen, z.B. welche Informationen ein Contributor beisteuern sollte, damit die Chance steigt, dass du den Fehler schneller verstehen und beheben kannst.

Da dieses Thema schon sehr viele Menschen vor uns gehabt haben, bietet GitHub einen recht komfortablen Weg an, solche Vorlagen zu erstellen, nämlich über einen sogenannten Builder. Klicke in deinem Repository die *Settings* an – achte darauf, dass links *Options* ausgewählt ist – und scrollle runter, bis du in den Abschnitt *Features* gelangst. Wähle dort *Issues* an und klicke auf den Button *Set up templates* (deutsch »Vorlagen einrichten«, siehe Abbildung 4-39).

17 Das passiert häufiger, als man denkt, der Klassiker ist: »Die Software ist kaputt!«

18 »Was genau geht nicht? Was passiert, wenn du sie startest?«

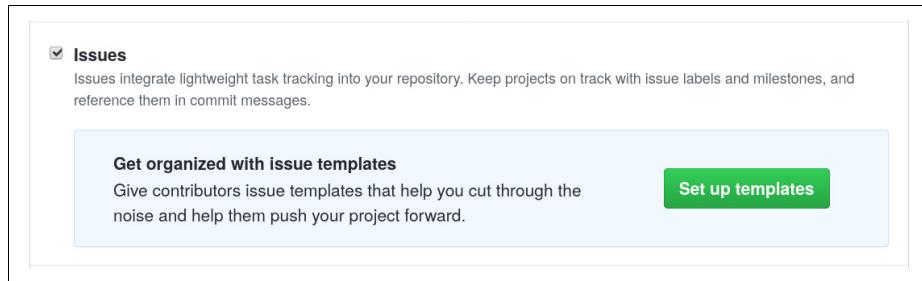


Abbildung 4-39: GitHub bietet einen Builder für mehrere Issue-Vorlagen an.

Du solltest mit dem Drop-down-Feld *Add template: select* begrüßt werden (siehe Abbildung 4-40). Sobald du das Drop-down anklickst, hast du die Wahl zwischen *Bug report* (»Fehlerbericht«), *Feature request* und *Custom template* (»benutzerdefinierte Vorlage«). Die ersten beiden sind die wohl häufigsten Anwendungsfälle bei Issues, der letzte Eintrag erlaubt dir, eine individuelle Vorlage zu erstellen.

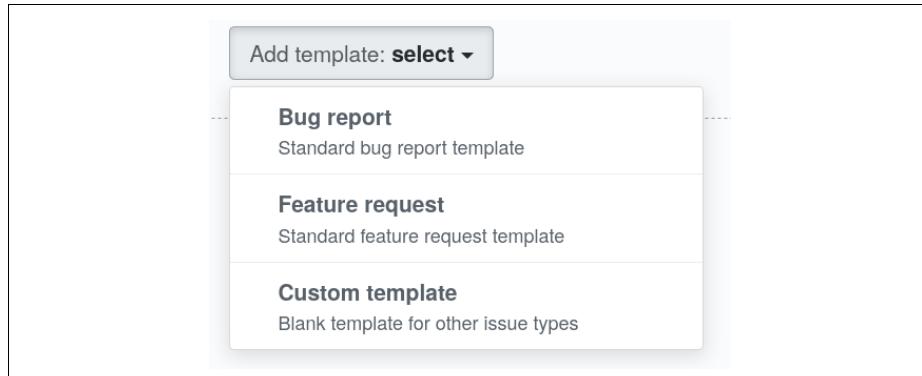


Abbildung 4-40: GitHub bietet bereits die beiden häufigsten Issue-Arten als Vorlage an.

Wir entscheiden uns für den *Bug report*. In der danach folgenden Übersicht (siehe Abbildung 4-41) kannst du das Template entweder löschen (über das Mülltonnen-Symbol) oder es durch Anklicken von *Preview and edit* (deutsch »Vorschau und editieren«) anpassen. Letzteres wollen wir tun.

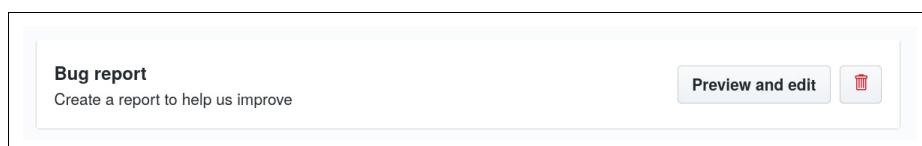


Abbildung 4-41: In der Übersicht der Templates lassen sich diese editieren oder löschen.

Du bekommst zunächst die Vorschau deiner Vorlage zu sehen (siehe Abbildung 4-42). Durch Anklicken des Stiftsymbols neben dem Namen *Issue: Bug report* kommen wir in den Editiermodus, den Abbildung 4-43 zeigt.



Abbildung 4-42: Den Editiermodus erreicht man über das Stiftsymbol in der Vorschau.

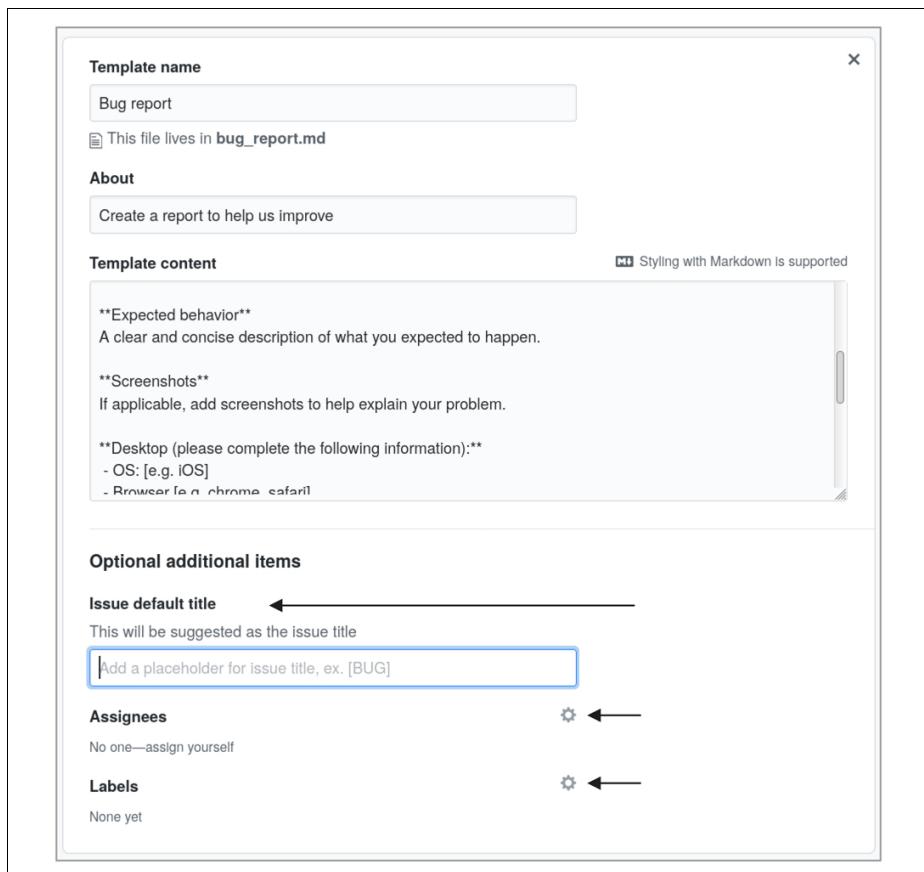


Abbildung 4-43: Im Editiermodus kann alles nach den eigenen Wünschen angepasst werden.

Wir passen das Template jetzt unseren Wünschen entsprechend an. Wir behalten die Vorschläge für Name und Inhalt bei und nehmen nur Anpassungen unterhalb des Punkts *Optional additional items* vor (deutsch etwa »optionale zusätzliche Einstellungsmöglichkeiten« – die mit den Pfeilen):

- Der Issue *default title* soll *[ERROR]* lauten.
- Als *Assignees* wählst du dich selbst.
- Und unter *Labels* wählen wir *bug*.

Nachdem du alles angepasst hast, musst du das Ganze noch speichern. Das erfolgt – etwas unscheinbar – über den Button *Propose changes* (deutsch »Änderungen vorschlagen«), siehe ❶ in Abbildung 4-44. Es öffnet sich ein Fenster mit der Möglichkeit, einen Commit durchzuführen ❷, wie wir es bereits kennen.

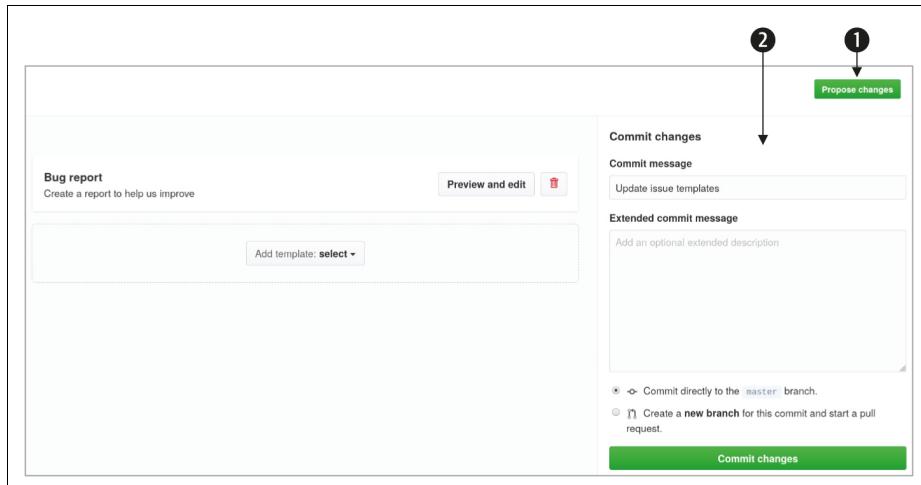


Abbildung 4-44: Der Speichern-Button bei den Templates befindet sich etwas unscheinbar rechts oben.

Wenn nun jemand in deinem Projekt einen Issue eröffnen möchte, bekommt er oder sie eine Auswahl dazu, welches Template verwendet werden soll (siehe Abbildung 4-45). Durch Anklicken von *Get started* (deutsch »loslegen«) wird die Maske für Issues angezeigt, die mit dem entsprechenden Template vorausgefüllt ist. In dem Bild siehst du auch, dass ich bereits ein zweites Template angelegt habe.



Abbildung 4-45: Wenn Issue-Templates aktiv sind, erhält der Contributor eine Auswahl beim Anlegen eines Issues.

Wähle das Template *Bug report* aus und schau dir an, wie ein Issue mit diesem Template jetzt aussieht (siehe Abbildung 4-46). Alle mit Pfeilen markierten Bereiche wurden vom Template vorausgefüllt. Ein Contributor weiß jetzt, was dir wichtig ist.

tig ist und wie du einen Issue markiert haben willst. Er kann alle diese Werte aber noch anpassen, wenn er es möchte.

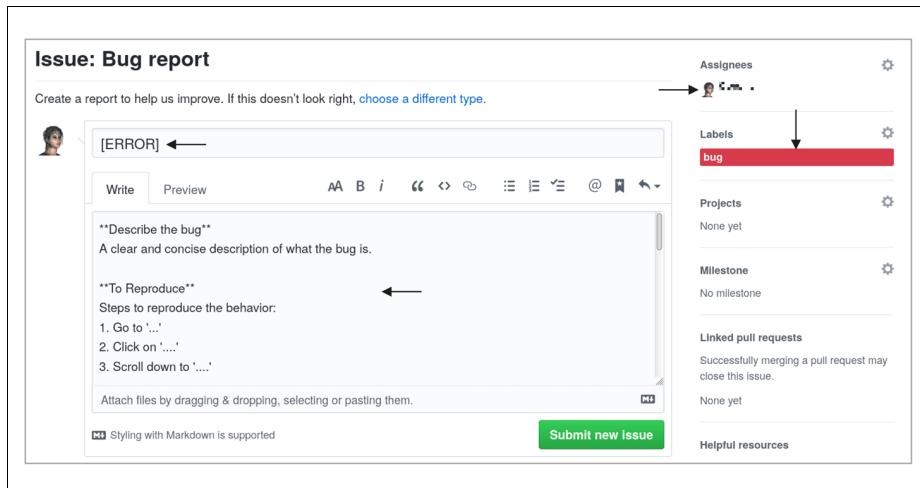


Abbildung 4-46: Alle vorher festgelegten Felder wurden automatisch befüllt.



### Issue-Vorlagen manuell anlegen

Es ist auch möglich, Issue-Vorlagen manuell anzulegen, indem du eine Datei namens *ISSUE\_TEMPLATE.md* (im Hauptverzeichnis oder unter *docs* oder *.github*) anlegst. Dafür gibt es auch Repositories, die Vorlagen anbieten.<sup>19</sup> GitHub empfiehlt aber den Weg über den Builder.<sup>20</sup>

So weit zu den Issue-Templates. Schauen wir uns die Templates für Pull-Requests an.

## Vorlagen für Pull-Requests

Was für Issues gilt, gilt auch für Pull-Requests. Du hast hier ebenfalls die Möglichkeit, eine Vorlage zu erstellen, allerdings ist das nicht ganz so komfortabel und mächtig wie bei Issues. Früher konnte man beide Vorlagen auf die gleiche Art und Weise erzeugen,<sup>21</sup> GitHub hat sich aber entschieden, für die Issues den Template-Builder zu bauen, den du im vorherigen Abschnitt benutzt hast. Eventuell wird es zukünftig ja auch noch einen Builder für Pull-Requests geben?

19 Beispielsweise <https://github.com/stevemao/github-issue-templates>.

20 <https://docs.github.com/en/github/building-a-strong-community/manually-creating-a-single-issue-template-for-your-repository>

21 Du findest heute noch viele Tutorials, die den »alten« Weg für Issue-Templates beschreiben.

Um ein Template für Pull-Requests anzulegen, erzeuge eine Datei namens *PULL\_REQUEST\_TEMPLATE.md*<sup>22</sup>, die entweder im Hauptverzeichnis, im Verzeichnis *docs* oder unter *.github* abgelegt wird. Diese Datei befüllen wir jetzt mit ein paar Informationen, die ein Contributor liefern sollte, damit der Pull-Request deinen Qualitätsansprüchen genügt. Ein nicht ganz ernst gemeinter Vorschlag für eine Checkliste:

- Pull-Requests werden nur angenommen, wenn
- [ ] du alle Tests durchgeführt hast.
  - [ ] du ganz lieb fragst.

Wenn jetzt ein Contributor einen Pull-Request erzeugt, wird dieser Text – wie bei den Issues auch – in den Inhalt kopiert. Mehr aber auch nicht. Immerhin kann der Contributor durch Anhaken der von dir vorgegebenen Checkliste dokumentieren, was er gemacht hat. Mehr kann dieses Template (noch) nicht. Beispielsweise kannst du (noch) keine Labels automatisch vergeben oder Ähnliches. Nur die Reviewer können wir über den Mechanismus der *CODEOWNERS* automatisch festlegen lassen (das hast du im Abschnitt »Reviews automatisch zuweisen – *CODEOWNERS*« auf Seite 65 schon kennengelernt).

So viel zu den Vorlagen. Wenn Issues und Pull-Requests in einem Repository angelegt werden, können diese aus den verschiedensten Gründen auch abgelehnt und einfach geschlossen werden. Manchmal entstehen dann hitzige Diskussionen über das Warum. In den letzten beiden Abschnitten schauen wir uns daher an, welche (technischen) Deeskalationsmechanismen du innerhalb deines Projekts nutzen kannst, um für Ruhe zu sorgen.

## Für Ruhe sorgen (Teil 1) – Locking Conversations

Wo Menschen und unterschiedliche Vorstellungen aufeinandertreffen, kommt es leicht zu hitzigen Debatten. Im besten Fall kühlt sich eine Diskussion von allein wieder ab, im schlechteren Fall fangen Menschen an, sich gegenseitig zu beleidigen. Das kann dir auch in deinem Projekt passieren. Dieser und der nächste Abschnitt zeigen dir daher zwei (technische) Mechanismen, um Diskussionen, die langsam aus dem Ruder laufen, erst einmal »auf Eis zu legen« und um dein Projekt temporär vor bestimmten Personengruppen abzuschotten.

Um eine Diskussion erst einmal auf Eis zu legen, gibt es die Möglichkeit, das Kommentieren bei einem Issue, einem Pull-Request oder einem Commit zu unterbinden. GitHub nennt das *Locking Conversations* (deutsch »Gespräche sperren« oder aus meiner Sicht treffender übersetzt mit: »Kommentarfunktion abschalten«). Diese Einschränkung gilt nur für den entsprechenden Issue/Pull-Request/Commit und auch nur für die Personen, die nicht Projekteignerin oder Maintainerin sind.

---

<sup>22</sup> Du kannst auch durchgängig kleinschreiben oder als Endung ».txt« nutzen, GitHub ist da nicht pingelig.

Das heißt, Maintainer können weiterhin Kommentare hinzufügen, ein Contributor nicht. Du findest diese Einstellung namens *Lock conversation* direkt im Issue/Pull-Request/Commit, wenn du ganz nach unten scrollst (siehe Abbildung 4-47).

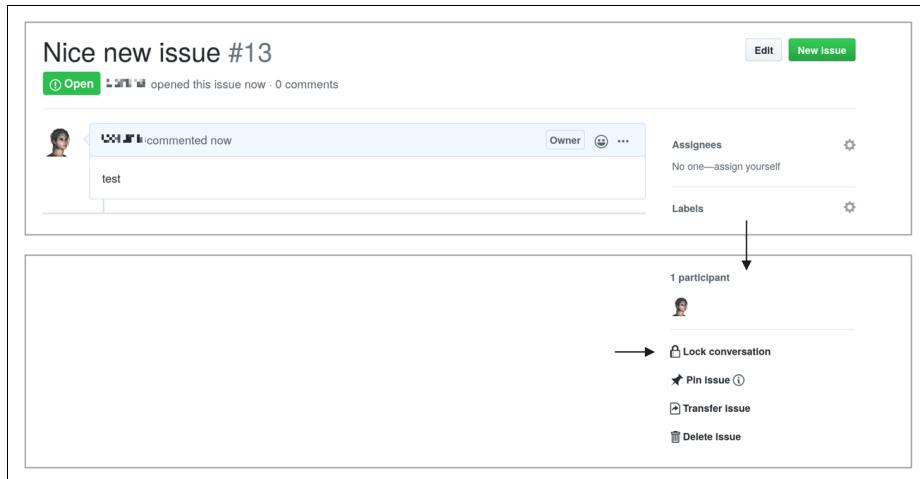


Abbildung 4-47: Der Button zum Abschalten der Kommentarfunktion ist gut versteckt.

Wir probieren das jetzt aus. Erstelle einen neuen Issue und wähle danach *Lock conversation* aus. Es erscheint ein neues Fenster (siehe Abbildung 4-48), das dir die Möglichkeit gibt, einen Grund für das Abschalten der Kommentarfunktion anzugeben. Das geht über das Drop-down *Choose a reason* (deutsch »Wähle einen Grund«).

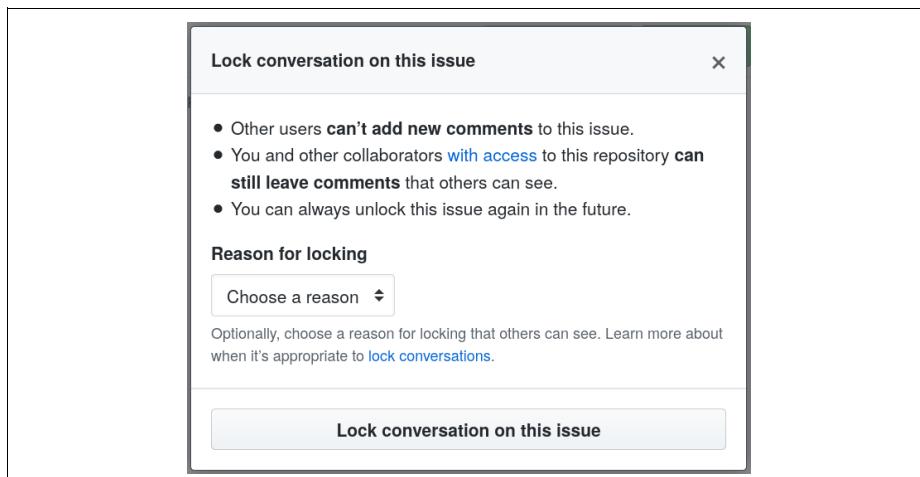


Abbildung 4-48: Es gibt die Möglichkeit, einen Grund für das Abschalten der Kommentarfunktion anzugeben.

Die möglichen Gründen sind fest vorgegeben und lauten wie folgt:

- *Off-topic* bedeutet so viel wie »am Thema vorbei« oder »themenfremd«, z.B. wenn Menschen anfangen, sich über die neuesten Kinofilme zu unterhalten (es sei denn natürlich, das Repository handelt davon).
- *Too heated* (deutsch »überhitzt«) ist das, was ich eingangs zu den hitzigen Debatten sagte.
- *Resolved* (deutsch »gelöst«) verwundert vielleicht auf den ersten Blick. Auf einem gelösten und geschlossenen Issue ist es weiterhin möglich, Kommentare hinzuzufügen. Falls man das nicht möchte, ist diese Begründung dafür gedacht.
- *Spam* (deutsch etwa »Müll«) ist, glaube ich, bekannt.

Falls keiner der vorgegebenen Gründe passt, kannst du auch die Standardeinstellung *Choose a reason* ausgewählt lassen. Wähle einen Grund (beispielsweise *Off-topic*) und klicke *Lock conversation on this issue* an. Schau dir danach deinen Issue an. Bei dir sollte jetzt so etwas Ähnliches zu sehen sein wie in Abbildung 4-49. Es ist angeben, wer die Kommentarfunktion abgeschaltet hat und mit welcher Begründung.

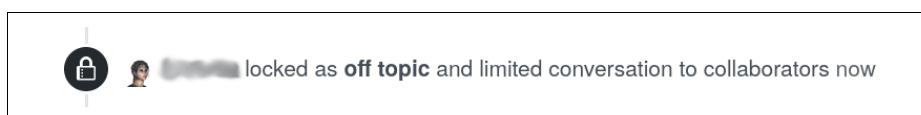


Abbildung 4-49: Aus Sicht der Projekteignerin: Die abschaltende Person und der Grund für die Abschaltung sind zu sehen.

Wenn du dir denselben Issue aus Contributor-Sicht anschaust (dafür musst du dich kurz ausloggen), sieht das Ganze etwas anders aus (siehe Abbildung 4-50).

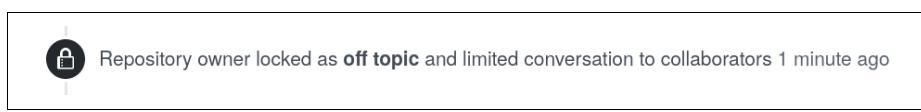


Abbildung 4-50: Aus Sicht des Contributors: Nur noch der Grund ist offen sichtbar, der Repository Owner könnte auch eine Maintainerin sein.

Die Begründung ist weiterhin da, nur die Person, die die Abschaltung vorgenommen hat, steht dort nicht mehr explizit, sondern nur noch *Repository owner*. Mit diesem ist aber nicht zwingend die Projekteignerin gemeint, es kann auch die Maintainerin gewesen sein. Nach meinem Verständnis ist das eine zusätzliche Deeskalationsmaßnahme, damit sich ein Konflikt nicht wieder daran entzündet, wer schlussendlich die Kommentarfunktion abgeschaltet hat.

Willst du die Kommentarfunktion wieder freischalten, geht das über denselben Weg:

1. Issue/Pull-Request/Commit auswählen.
2. Nach unten scrollen.
3. Link auswählen.

Der Link heißt in dem Fall *Unlock conversation* (deutsch etwa »Kommentarfunktion freischalten«) und hebt die Einschränkung wieder auf. Diese Einschränkung kann nur manuell durch Anklicken des Links aufgehoben werden. Solltest du vergessen haben, sie wieder aufzuheben, bleibt der Issue/Pull-Request/Commit für immer gesperrt. GitHub bietet aber noch einen anderen, einen automatischen Weg an, um für Ruhe zu sorgen.

## Für Ruhe sorgen (Teil 2) – Interaction Limits

Die andere Möglichkeit, für Ruhe zu sorgen, ist, die Interaktion auf dem Repository generell einzuschränken. Das betrifft zum einen, wer grundsätzlich kommentieren darf, und zum anderen, wer neue Issues oder Pull-Requests erstellen darf. Das nennt sich *Interaction Limits* (deutsch »Begrenzungen der Interaktion«), und du findest sie unter den *Settings* deines Repositorys (siehe auch Abbildung 4-51).

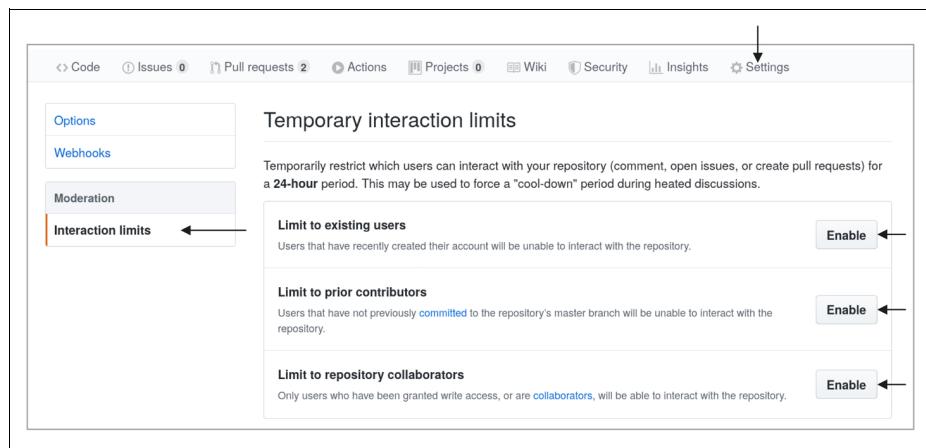


Abbildung 4-51: GitHub bietet die Möglichkeit, die Interaktion auf einem Repository temporär einzuschränken.

Du kannst dort festlegen, welche Nutzergruppen während eines festgelegten Zeitraums die genannten Interaktionen auf deinem Repository durchführen dürfen und welche nicht. Folgende Optionen stehen bereit:

- *Limit to existing users* (deutsch »Begrenze auf existierende Nutzer«) ist die schwächste Einschränkung. Ausgeschlossen werden alle Nutzer, deren Account jünger als 24 Stunden ist.<sup>23</sup> Nicht betroffen sind jüngere Accounts, die aber bereits Contributor waren oder Maintainerin sind.

<sup>23</sup> Es soll ja Menschen geben, die sich extra einen Account kurzfristig anlegen, um zu spammern oder zu trollen.

- *Limit to prior contributors* (deutsch »Begrenze auf frühere Contributoren«) ist die nächste Stufe der Einschränkung. Ausgeschlossen werden alle Nutzer, die nicht bereits vorher schon als Contributor tätig waren oder Maintainerin sind.<sup>24</sup>
- *Limit to repository collaborators* (deutsch etwa »Begrenze auf Maintainer«) ist die restriktivste Einschränkung. Ausgeschlossen werden alle Nutzer, die nicht Maintainer oder Maintainerinnen sind.

Du kannst nur eine der drei Optionen zeitgleich aktiv haben. Hast du dich für eine Option entschieden, wählst du sie durch einen Klick auf den entsprechenden Buttons *Enable* (deutsch »Aktivieren«) aus. Dann kannst du den Zeitraum auswählen, derzeit sind 24 Stunden, drei Tage, eine Woche, ein Monat oder sechs Monate einstellbar. Im Anschluss läuft eine Art Eieruhr, die die Option nach dem festgelegten Zeitraum wieder deaktiviert (siehe Abbildung 4-52). Solltest du die Beschränkung früher deaktivieren wollen, kannst du das durch Anklicken des Buttons *Disable* (deutsch »Deaktivieren«) herbeiführen.

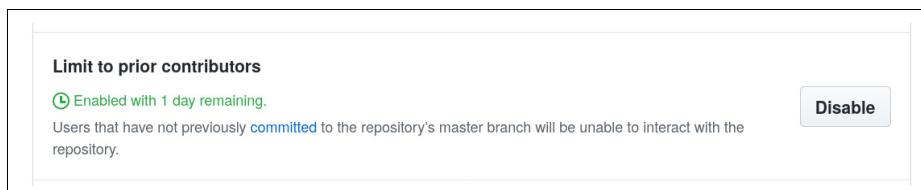


Abbildung 4-52: Nach Auswahl einer der Beschränkungen läuft automatisch eine Zeitschaltuhr ab, die nach Ablauf die Option wieder deaktiviert.

Der Vorteil hierbei ist, dass du nicht aktiv daran denken musst, die Einschränkungen wieder aufzuheben. Die eingebaute Zeitschaltuhr übernimmt das für dich. Andererseits musst du dafür diese Einschränkung jedes Mal neu aktivieren, falls es nötig sein sollte.

Jetzt haben wir die wichtigsten Grundlagen zu GitHub durch. Natürlich gibt es noch viel mehr zu entdecken – und das werden wir auch noch. Bevor wir aber tiefer einsteigen, möchte ich dich im nächsten Kapitel mit dem Thema Open-Source-Lizenzen vertraut machen. Das ist ein wichtiges Thema, wenn du mit anderen Menschen zusammen an Projekten arbeiten möchtest, und um dir einen Grundstock an Wissen zu vermitteln, habe ich das nächste Kapitel geschrieben.

<sup>24</sup> GitHub beschreibt die Einschränkung mit: »Wer noch nicht auf den master-Branch committet hat.«

## KAPITEL 5

# Rechtliches – Open-Source-Lizenzen

Lizenzen! Das klingt ja erst einmal ganz schön dröge, und Juristisches hat ja eher selten die Aura von etwas Aufregendem. Lizenzierung ist aber ein wichtiges Thema, denn ein Projekt wird erst dann zu Open Source, wenn eine entsprechenden Lizenz gesetzt wurde.

Die Wahl der richtigen Lizenz ist allerdings abhängig von vielen verschiedenen Faktoren, unter anderem davon, was du mit deinem Projekt erreichen willst. Ich persönlich habe keinen juristischen Hintergrund und werde dir daher auch keine (rechtsverbindlichen) Empfehlungen geben können. Die Informationen in diesem Kapitel sollten dir aber hoffentlich dabei helfen, herauszufinden, welche Lizenz für dich die passende ist.



### Am Ende des Kapitels weißt du ...

- warum du dich mit dem Thema Lizenzen überhaupt beschäftigen solltest.
- ob du dir eine Lizenz selber bauen musst und ob das überhaupt eine gute Idee ist.<sup>1</sup>
- welche Lizenzen beliebt und verbreitet sind (dafür wird es vermutlich Gründe geben).
- welche bekannten Open-Source-Produkte welche Lizenz nutzen (so als Referenz).
- wo es Hilfe zur Entscheidungsfindung gibt.
- wie du eine gewählte Lizenz in deinem GitHub-Repository veröffentlichen kannst.

## Warum Lizenzierung wichtig ist

Okay, fangen wir also erst einmal damit an, zu klären, warum Lizenzierung überhaupt wichtig ist. Wenn du ein Werk erzeugst, sei es Softwarecode, ein Gedicht oder ein Bild, greift erst einmal grundsätzlich das Urheberrecht<sup>2</sup>. Das passiert auto-

1 Spoiler: nein und nein!

2 Wer Lust auf die Details hat, findet sie hier: <https://www.gesetze-im-internet.de/urhg/>, insbesondere für Computerprogramme ist § 69a-g interessant.

matisch und kostenfrei – im Gegensatz zu Patenten. Diese muss man mit ein paar Geldscheinen in der Hand bei entsprechenden Behörden anmelden. Das Urheberrecht legt fest, dass nur du allein die Rechte an deinem Werk hast. Bei Software bedeutet das beispielsweise, dass niemand außer dir diese Software verwenden, verändern oder weitergeben darf.<sup>3</sup>

Vielleicht denkst du gerade: »Hoppla, aber dann kann ja keiner meine coole App nutzen!« oder: »Schade, kann mich dann niemand bei meinem Projekt unterstützen?« Bingo! Genau da kommen Lizenzen zum Einsatz. Lizenzen sind also dafür da, anderen Menschen entsprechende Rechte an deinem Werk einzuräumen. Sie schützen damit deine Nutzerinnen und Nutzer und ermöglichen überhaupt erst die Anwendung.



#### Tipp: Projekt unbedingt lizenzieren

Wenn du willst, dass dein Projekt von anderen genutzt werden kann, solltest du es lizenzieren und diese Lizenz auch auffindbar machen. Ein Projekt ohne Lizenz ist unbenutzbar.

Spätestens zu diesem Zeitpunkt ist dir vielleicht klar geworden, dass das reine Veröffentlichen deines Projekts auf GitHub – auch in einem öffentlichen Repository – nicht dasselbe ist wie eine Lizenzierung. Das ist vielleicht vergleichbar damit: Wenn eine fremde Person im Restaurant ihr Handy auf den Tisch legt, kannst du es zwar sehen, darfst es aber natürlich nicht einfach nehmen und benutzen. Die Nutzungsbedingungen von GitHub<sup>4</sup> decken einige Punkte ab, wie beispielsweise, dass andere Nutzerinnen dein Projekt ansehen und forken/teilen dürfen, aber nicht, dass sie es auch nutzen oder modifizieren dürfen (»Forken« lernst du in Kapitel 6 noch kennen).

Ohne Lizenz können sogar richtig skurrile Situationen entstehen, zum Beispiel: Jemand steuert ein paar Zeilen Code zu deinem Projekt bei – und schon darfst selbst du die Software theoretisch nicht mehr nutzen, obwohl dein Mitstreiter und du die Urheber seid.

## Lizenz Marke Eigenbau

Aus dem vorherigen Abschnitt hast du nun mitgenommen, dass eine Lizenz schon irgendwie wichtig ist, wenn du willst, dass andere dein Projekt nutzen oder dabei unterstützen wollen. Wie erstelle ich also so eine Lizenz? Um es ganz klar zu sagen: Ich rate dir dringend davon ab, dir eine eigene Lizenz zu bauen! Nicht nur, dass du eventuell eine entsprechende Fachanwältin hinzuziehen müsstest, es kann auch passieren, dass ungewollte Nebenwirkungen entstehen (siehe dazu weiter unten das Fallbeispiel »Lizenz selber bauen – amüsantes Fallbeispiel JSLint«).

3 Dass das trotzdem manche Menschen machen, steht auf einem anderen Blatt ...

4 <https://help.github.com/en/github/site-policy/github-terms-of-service>

## Lizenz selber bauen – amüsantes Fallbeispiel JSInt

Was passieren kann, wenn sich jemand eine eigene Open-Source-Lizenz baut, zeigt das nachfolgende Beispiel.<sup>5</sup> Douglas Crockford, Entwickler von JSInt (einer Software zur statischen Analyse von JavaScript-Code)<sup>6</sup>, war unzufrieden mit den existierenden Lizzenzen. Er ergänzte daher die bereits vorhandene MIT-Lizenz um den Satz »Die Software soll für Gutes, nicht für Böses verwendet werden.« (»The Software shall be used for Good, not Evil.«). Heraus kam die JSInt-Lizenz.

Dieser eine Satz hatte zur Folge, dass die Software als »unfrei« und auch »heikel« eingestuft und deswegen häufig nicht verwendet wurde. IBM hatte starkes Interesse an der Nutzung der Software, äußerte aber aufgrund dieses Satzes entsprechende Bedenken (sie hätten schließlich nicht verhindern können, dass eine ihrer Kundinnen nicht doch etwas Böses damit anstellen würde). Das führte dazu, dass Douglas Crockford der Firma einen Freibrief erteilte mit den Worten: »Ich erlaube IBM, seinen Kunden, Partnern und Untergebenen, JSInt für böse Zwecke zu verwenden.« (»I give permission for IBM, its customers, partners, and minions, to use JSInt for evil.«). Mit dieser Erlaubnis war es IBM jetzt möglich, die Software zu nutzen.

Zum Glück gibt es schon eine ganze Menge bestehender Lizzenzen, die man einfach kopieren und an geeigneter Stelle ins Repository einfügen kann. Weitere Gründe, die für den Einsatz einer bestehenden Lizenz sprechen, sind:

- Sie sind bekannt. Jemand, der sich bereits mit Lizzenzen beschäftigt hat, weiß, was die Lizenz GPLv3 bedeutet, und muss in der Regel nicht mehr groß nachschlagen. Eine neue Lizenz, wie die »Anke Lederer Special License (ASLS)«<sup>7</sup>, kennt niemand und erfordert erst einmal Zeit und Energie, um die Lizenzbedingungen zu studieren und zu verstehen.
- Es gibt viele leicht verfügbare Informationen über bereits bestehende Lizzenzen. Zur oben genannten GPLv3 gibt es im Internet allein Dutzende – wenn nicht gar Hunderte – Websites mit einer Fülle an Informationen zu Bedeutung, Kompatibilitäten, Erfahrungen etc.



### Empfehlung: Wähle eine bereits vorhandene Lizenz

Lizenzen der Marke Eigenbau benötigen in der Regel rechtliche Unterstützung und können unbeabsichtigte Nebenwirkungen haben.

Die Frage ist also: Welche bestehende Lizenz ist die passende für dein Projekt?

5 Quellen: <https://en.wikipedia.org/wiki/JSInt> und <https://wonko.com/post/jsmin-isnt-welcome-on-google-code/>.

6 <https://github.com/douglascrockford/JSInt/>

7 Für mich klingt die natürlich total cool, aber eine solche Lizenz dient vermutlich mehr meiner eigenen Eitelkeit ...

# Welche Lizenzen gibt es?

Die richtige Open-Source-Lizenz zu finden, ist für den Anfänger (und auch viele Fortgeschrittene) eine erst einmal undurchschaubare und vielleicht auch Angst einflößende Angelegenheit. Sobald du eine Lizenz für dein Projekt gewählt hast, kann es unter Umständen nämlich langwierig und auch schwierig sein, diese wieder zu ändern. Wenn du beispielsweise zuerst eine stark einschränkende Lizenz gewählt hast und sie dann abschwächen willst, musst du gegebenenfalls alle Menschen, die bei der Erstellung deines Werks beteiligt waren, um Erlaubnis bitten. Was für ein Aufwand!

Aus diesem Grund solltest du im Vorfeld gut abwägen, was dir eigentlich wichtig ist. Aber keine Panik, nach dem Lesen dieses Kapitels wirst du hoffentlich ein erstes Gefühl dafür bekommen haben, welche Lizenz für dich und dein Projekt die richtige sein könnte.

Die erste Frage, die wir klären sollten, ist, was genau du lizenziieren willst. Reden wir von Softwarecode? Oder reden wir von Musik, Bildern und Texten (beispielsweise Gedichten, Liedern oder Anleitungen)? Je nachdem, um was es sich handelt, ist eventuell ein anderes »Lizenzpaket« sinnvoller.

## Softwarecode

Lizenzen für Open-Source-Software gibt es Hunderte. Erschwerend kommt hinzu, dass es von einigen Lizenzen auch noch verschiedene Versionen gibt, die aber alle gleichzeitig weiterhin gültig sind. Eine Software mit der Lizenz GPL Version 2 wird also nicht automatisch durch die Lizenz GPL Version 3 ersetzt. Bei Softwareversionen ist das ja meist anders, Version 10 ersetzt in der Regel Version 9 (sofern die Anwenderin ein entsprechendes Update macht).

Zum Glück kann man die »Lizenzlandschaft« grob in drei Bereiche aufteilen: freizügige Lizenzen (*permissive*) und sogenannte Copyleft-Lizenzen, wobei Letztere noch in Lizenzen mit starkem und schwachem Copyleft unterteilt werden (siehe hierzu auch Tabelle 5-1).

Tabelle 5-1: Unterteilung verbreiteter Open-Source-Softwarelizenzen nach ihrem Copyleft

Art des Copyleft	Bedeutung	Beispiele
kein Copyleft (permissive)	Code darf auch in proprietärer Software weitergegeben werden.	MIT, Apache, BSD, ISC
schwaches (oder auch beschränktes) Copyleft	Nutzen von Codeteilen (z. B. bei Programmierbibliotheken) auch in proprietärer Software möglich.	LGPL, MPL
starkes Copyleft	Keine Einbindung des Codes in proprietäre Software möglich. Veränderter Code muss wieder unter dieselbe Lizenz gestellt werden.	GPL, AGPL

## Proprietäre Software

Der Begriff »proprietär« bedeutet so viel wie »im Eigentum befindlich« oder »firmeneigen« und wird häufig als *Closed Source* (deutsch »verschlossener« oder »abgeschlossener Quellcode«) bezeichnet. Das bedeutet, der Quellcode von proprietärer Software ist nicht zugänglich und stellt damit einen Gegenpol zu Open Source dar.

## Copyleft

Copyleft ist ein Wortspiel im Zusammenhang mit Copyright und soll die Freiheit von Software sicherstellen. Du erinnerst dich sicher an die Erklärbarbox zu Open Source (»Open Source«) in Kapitel 1. Das war die Freiheit, Software zu verwenden, zu verbreiten, zu verändern und den Quellcode zu verstehen.

Wenn du beispielsweise ein Werk – mit Erlaubnis! – weiterbearbeitest, bekommst du ein Mitspracherecht dazu, was mit dem bearbeiteten Werk passieren kann. Du bist jetzt (Mit-)Urheberin des Werks. Das könnte dazu führen, dass beispielsweise Softwarecode, der vorher Open Source war, nach deiner Bearbeitung zu Closed Source wird und die oben genannten Freiheiten eingeschränkt werden würden.

Um das zu verhindern, wurde das Konzept des Copylefts erfunden. Copyleft garantiert also, dass die Nutzer weiterhin diese Freiheiten haben. Geprägt wurde der Begriff von der *Free Software Foundation*<sup>8</sup> (FSF, deutsch »Stiftung für freie Software«), die sich als gemeinnützige Organisation der Unterstützung und Förderung freier Software verschrieben hat.

Puh, das erschlägt einen erst einmal, oder? Eigentlich ist es aber ziemlich einfach. Stellst du deinen Code unter eine *permissive Lizenz*, dürfen andere – beispielsweise Firmen – deinen Code nehmen und ihn in ihre (kostenpflichtigen oder kostenlosen) Produkte einbauen. Der Code für diese Produkte kann dabei Closed Source sein, d.h., im schlimmsten Fall wird deine brillante Idee noch mehr verbessert, aber du kannst dir das im Code nicht anschauen, wenn die Firma den Zugang zu deinem – jetzt verbesserten – Code verweigert. Rechtlich darf sie das, für die Allgemeinheit geht aber etwas verloren. Mit einer freizügigen Lizenz gibst du also relativ viel Kontrolle ab.

Das andere Extrem ist, deinen Code unter eine *starke Copyleft-Lizenz* zu stellen. Jetzt muss jede Person, die deinen Code verwenden möchte, ihre weitere Bearbeitung wieder unter die gleiche Lizenz stellen. Das verhindert, dass sich jemand einfach deine geniale Idee greift und der Allgemeinheit entzieht, wie es bei den permissive Lizzenzen der Fall sein kann. Diese Regelung wird häufig auch als »ansteckend« oder »viral« bezeichnet und kann im kommerziellen Umfeld zu Problemen führen,

<sup>8</sup> <https://www.fsf.org/>

da manche Unternehmen solche Lizenzen vermeiden. Du hast insofern Kontrolle über den Code, als dass du sicher sein kannst, dass von anderen verbesserter Code wieder frei zugänglich gemacht werden muss. Du kannst die Verbesserungen dann beispielsweise zurück in deinen Code übernehmen, wenn du das möchtest. Falls du jedoch irgendwann mal vorhast, mit Firmen zusammenzuarbeiten, könnte das eventuell ein Problem werden.

Lizenzen mit einem *schwachen Copyleft* stehen irgendwo dazwischen. Sie sind etwas restriktiver als die permissive Lizenzen, aber auch nicht so streng wie die mit starkem Copyleft und können damit einen guten Kompromiss darstellen.

## Open-Source-Lizenzen aus Deutschland

Die bisherigen Lizenzbeispiele kommen alle aus dem englischsprachigen Raum. Es gibt aber auch deutsche Open-Source-Lizenzen, z.B. die *Deutsche Freie Software Lizenz*<sup>9</sup> und die *Bremer Lizenz für freie Softwarebibliotheken*<sup>10</sup>, die beide kleinere rechtliche Inkompatibilitäten mit den bestehenden Nutzungsbedingungen US-amerikanischer Lizenzen ausbügeln. Einsortieren kann man die beiden Lizenzen unter dem *schwachen Copyleft*.

Ob diese Lizenzen für dein rein deutsches Projekt die bessere Wahl sind, musst du im Einzelfall entscheiden. In freier Wildbahn habe ich noch keine dieser Lizenzen beobachtet.

## Musik, Bilder und Texte

Für Erzeugnisse wie Bilder, Texte oder Musik gibt es eine andere Möglichkeit der Lizenzierung. Auch diese können Open Source sein (im Sinne von verwenden, verbreiten und verändern, den Quellcode zu verstehen, ist hier eher selten im Fokus). Dafür nutzt man in der Regel keine der weiter oben bereits erwähnten Lizenzen, sondern andere offene Lizenzen für Medien.

Zwei Lizenzen möchte ich dir dazu vorstellen, einmal die *GNU Free Documentation License* (GNU FDL oder auch GFDL),<sup>11</sup> die für Dokumentationen vorgesehen ist und aus der Familie der GPL-Lizenzen stammt. Sie besitzt damit ebenfalls ein entsprechendes Copyleft, was die Weiterveröffentlichung nach einer Veränderung sicherstellen soll.

Eine andere sehr verbreitete »Lizenzgruppe« ist *Creative Commons*<sup>12</sup>, die nicht nur für Dokumentationen, sondern auch für Bilder, Videos und Musik genutzt werden

9 <https://www.hbz-nrw.de/produkte/open-access/lizenzen/dfl>

10 [https://www.itzbund.de/SharedDocs/Downloads/DE/DVDV/DVDV\\_BremerLizenz.html](https://www.itzbund.de/SharedDocs/Downloads/DE/DVDV/DVDV_BremerLizenz.html)

11 <https://www.gnu.org/licenses/#FDL>

12 <https://creativecommons.org/>

kann. Creative Commons werden dabei nach einer Art Baukastenprinzip zusammengestellt, wobei nicht jede Kombination möglich ist. Es gibt sechs Hauptlizenzen, die aus vier Bausteinen bestehen können (siehe auch Abbildung 5-1), wobei CC0 eine Sonderstellung einnimmt. Diese Lizenz entspricht dem Verständnis von *Public Domain (PD)*: »Jeder darf alles mit dem Werk machen.« Eine zusammengebaute Lizenz könnte beispielsweise so aussehen: CC-BY-SA.

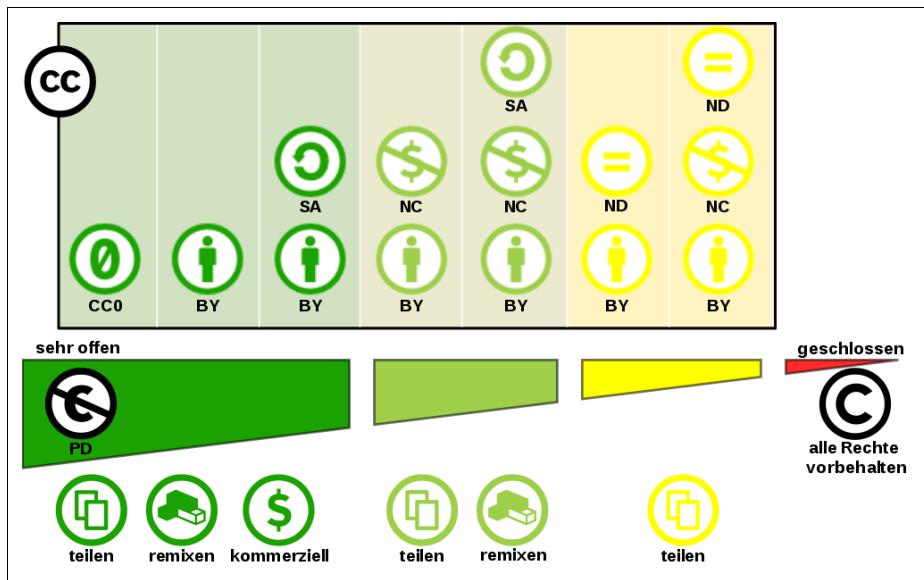


Abbildung 5-1: Der Baukasten von Creative Commons (erstellt von Napa<sup>13</sup>); Lizenz: Creative Commons Attribution 4.0 International Lizenz<sup>14</sup>

Die einzelnen Bestandteile bedeuten dabei Folgendes:

- **BY – Attribution** (deutsch »Namensnennung«): Legt fest, dass der Autor oder die Autorin des Werks genannt werden muss.
- **SA – Share Alike** (deutsch »gleichermaßen teilen«): Bedeutet, dass andere das Werk kopieren, verteilen, anzeigen, aufführen und verändern dürfen, solange sie die Veränderungen zu den gleichen Bedingungen verbreiten – das entspricht in etwa dem Copyleft bei Softwarelizenzen.
- **NC – Non Commercial** (deutsch »nicht kommerziell«): Bedeutet, dass andere das Werk kopieren, verteilen, anzeigen, aufführen und verändern dürfen – aber nicht in einem kommerziellen Kontext.
- **ND – Non Derivatives** (deutsch »keine Ableitungen«): Bedeutet, dass andere das Werk zwar kopieren, verteilen, anzeigen und vorführen dürfen, aber verändern ist nicht erlaubt.

13 Quelle: [https://commons.wikimedia.org/wiki/File:Creative\\_Commons\\_Lizenzspektrum\\_DE\\_quer.svg](https://commons.wikimedia.org/wiki/File:Creative_Commons_Lizenzspektrum_DE_quer.svg).

14 <https://creativecommons.org/licenses/by/4.0/deed.en>

Gerade der Einsatz von NC wird teilweise recht kritisch gesehen. Was vielleicht erst einmal ganz gut klingt (»niemand darf mit meinem Werk ungefragt Geld verdienen«), zeigt sich aber in der praktischen Umsetzung häufig ein Problem.

**Fallbeispiel 1:** Eine Schülerin betreibt ein kostenloses Blog, in dem sie über ihren Schulalltag schreibt. Sie benutzt dafür Bilder mit dem CC-Baustein NC. Alles kein Problem, oder? Was passiert, wenn sie jetzt über die Einblendung von Werbung ein bisschen Geld dazu verdient, um zumindest die Hosting-Gebühren ihres Blogs bezahlen zu können?

**Fallbeispiel 2:** Ein Wissenschaftler eines Forschungsbereichs an der Universität nutzt für eine kostenlose Publikation ebenfalls ein NC-Bild. Die Uni bekommt aber Fördergelder für diesen Forschungsbereich. Wie ist das zu bewerten?

Leider habe ich auf diese Fragen auch keine juristisch wasserichte Antwort. Vielleicht möchtest du der Schülerin oder dem Wissenschaftler die Nutzung deines Werks ermöglichen. Es könnte aber sein, dass diese aufgrund des NC-Bausteins lieber vorsichtig sind und eine Alternative ohne NC suchen. Wenn du das nicht möchtest, solltest du auf den Baustein verzichten. Einen Tipp, den ich irgendwo gelesen habe, war: »Wenn du bereit bist, dein Recht auf nicht kommerzielle Nutzung einzuklagen, dann verwende diesen Baustein, ansonsten lass es lieber bleiben.« (Quelle leider unbekannt)



#### Tipp: Creative Commons für alles andere nutzen

Ich persönlich würde dir bei Werken, die kein Softwarecode sind, zu Creative Commons raten. Im Vergleich zur GNU FDL kannst du damit mehr lizenziieren (Bilder!), und sie sind deutlich einfacher zu handhaben. Beispielsweise muss bei der GNU FDL immer die ganze Lizenz mit ausgedruckt werden,<sup>15</sup> während bei den Creative Commons normalerweise Namensnennung, Titel, Quelle und Lizenz ausreichen.

Aber Achtung: Die Creative Commons sind nicht für Softwarecode gedacht! Hier musst du eine andere Lizenz wählen.

## Was wählen andere als Lizenz?

Manchmal ist es hilfreich und auch entlastend, zu schauen, was andere so wählen – natürlich in der Hoffnung, dass die anderen schon nicht so falsch liegen. Lass dich aber bitte nicht dazu verleiten, eine bestimmte Lizenz zu nehmen, nur weil sie besonders oft genutzt wird.

Im Jahr 2015 hat GitHub eine Erhebung dazu gemacht, welche Lizenzen wie oft auf ihrer Plattform vertreten sind. Das Ergebnis zeigt Abbildung 5-2. Wie du schön sehen kannst, ist die MIT-Lizenz am weitesten verbreitet, mit Abstand gefolgt von den GPL-Lizenzen (GPLv2, GPLv3, LGPLv3 und AGPLv3) und der Apache-Lizenz.

<sup>15</sup> Das können mehrere DIN-A4-Seiten sein, siehe <https://www.gnu.org/licenses/fdl-1.3.html>.

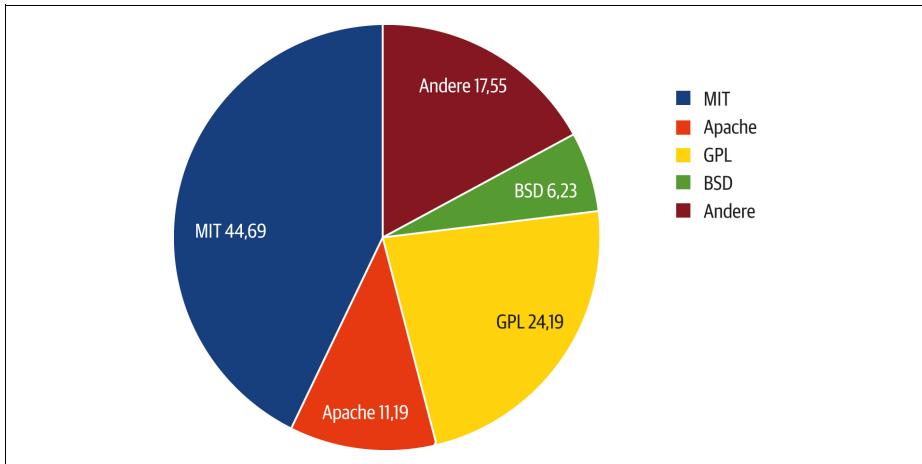


Abbildung 5-2: Welche Lizenzen sind auf GitHub vertreten (Stand 2015)?<sup>16</sup>

Die MIT-Lizenz hat als *permissive* Lizenz den großen Vorteil, dass sie sehr einfach und kurz ist. Man braucht keinen juristischen Abschluss, um sie zu verstehen, und die Implementierung ist einfach – womöglich auch der Grund dafür, dass sie so verbreitet ist.



#### Achtung: MIT-Lizenz und Softwarepatente

Eine Fragestellung gibt es bei der MIT-Lizenz im Bereich Softwarepatente. Wenn ein Patentinhaber entsprechend patentierten Code einem Open-Source-Projekt hinzufügt, gewährt er damit den Nutzern der Software auch automatisch die Patentrechte? Oder könnte es passieren, dass der Patentinhaber später Patentlizenzzgebühren für die Verwendung der Software verlangt? Für die (rechtssichere) Beantwortung dieser Frage müsste wieder unsere Fachanwältin ran.

Über diesen Punkt kann man viel im Netz nachlesen. Für den US-amerikanischen Markt gibt es wohl keine Hinweise darauf, dass nachträglich noch Patentlizenzzgebühren verlangt werden.<sup>17</sup> Wer aber auf Nummer sicher gehen möchte, wählt lieber eine Lizenz mit sogenannten Patentnutzungsklauseln, wie z.B. Apache 2.0, MPL 2.0 oder GPLv3.

Manchmal hilft es ja auch, zu wissen, welche Software welche Lizenz nutzt, um das Ganze für sich selbst etwas besser einsortieren zu können. Ich habe daher eine Auswahl von – mehr oder minder – bekannter Open-Source-Software getroffen und ihre jeweilige Lizenz dazu aufgelistet (siehe Tabelle 5-2). Bei den meisten Softwareprodukten habe ich dazugeschrieben, um was es sich handelt – nur für den Fall, dass dir die eine oder andere nicht geläufig sein sollte. Mein Orientierungspunkt bei der Auswahl der Lizenzen war die Relevanz aus meiner Sicht.

16 Quelle: <https://github.blog/2015-03-09-open-source-license-usage-on-github-com/>

17 <https://opensource.com/article/18/3/patent-grant-mit-license>

Tabelle 5-2: Übersicht ausgewählter Open-Source-Projekte, aufgeteilt nach Lizenzen

Lizenz	Auswahl entsprechender lizenziert Projekte
MIT License	Bootstrap (Framework für die Webentwicklung), Atom, Visual Studio Code, Brackets, GitHub Desktop (alles Editoren, die ich in Kapitel 11, Abschnitt »Editoren und Handy-Apps« auf Seite 250, noch näher vorstellen werde)
GNU General Public License v2.0, GPLv2	Linux Kernel, Git, Telegram (Messenger), VLC (Mediaplayer), LMMS (Musikproduktionssoftware), Joomla! (Content-Management-System)
GNU General Public License v3.0, GPLv3	Signal (Messenger), Darktable (Fotobearbeitungssoftware), uBlock Origin (Werbeblocker für Browser), Tutanota (Mailanbieter), Gimp (Grafikprogramm)
GNU Lesser General Public License v3.0, LGPLv3	Adafruit NeoPixel (Arduino-Bibliothek für LED-Steuerung), cryfs (kryptisches Dateisystem für die Cloud), Adguard Browser Extension (Werbeblocker für Browser)
GNU Affero General Public License v3.0 AGPLv3	F-Droid (App-Store), Mastodon (verteilter Mikroblogging-Dienst), Overleaf (webbasierter kollaborativer LaTeX-Editor), invidious (alternatives Frontend für YouTube), Povray (3-D-Grafikprogramm)
Apache License 2.0	TensorFlow (Machine-Learning-Framework), Docker (Containerlösung), Kubernetes (Containermanagement), moby (erinnere dich an Kapitel 2)



### Tipp: Lizenzschnittstelle

GitHub bietet eine API (*Application Programming Interface*, deutsch »Schnittstelle für die Programmierung von Anwendungen«), um Metadaten von beliebten Open-Source-Lizenzen und Informationen über die Lizenzdatei eines bestimmten Projekts abzufragen: <https://developer.github.com/v3/licenses/>.

Die GitHub-API schauen wir uns im Abschnitt »GitHub-API (für Fortgeschrittene)« auf Seite 247 in Kapitel 11 noch kurz an.

In einer von WhiteSource<sup>18</sup> durchgeführten Trendanalyse<sup>19</sup> sieht man deutlich, dass der Trend zu permissive Lizenzen geht. MIT License und Apache 2.0 License sind dabei klare Favoriten.

Ich hoffe, du hast jetzt einen ersten Überblick darüber, wie die Lizenzlandschaft aussieht. Im nächsten Abschnitt wollen wir uns der Frage widmen, welche Lizenz für dich wohl die richtige sein mag.

## Welche Lizenz ist die richtige für mich?

Ich möchte dir noch ein paar Gedanken mit auf den Weg geben, die bei der Lizenzentscheidung bezüglich Software hoffentlich helfen werden. Du solltest dir auf jeden Fall darüber klar werden, was du mit deinem Projekt erreichen willst und ob die gewählte Lizenz förderlich oder hinderlich für diese Zielerreichung ist.

<sup>18</sup> Eine Softwarefirma, die eine Lösung für Sicherheit, Compliance und Reporting zur Verwaltung von Open-Source-Komponenten anbietet: <https://www.whitesourcesoftware.com/>.

<sup>19</sup> Quelle: <https://resources.whitesourcesoftware.com/blog-whitesource/top-open-source-licenses-trends-and-predictions>.

- Möchtest du Unterstützer für dein Projekt anlocken, denen es wichtig ist, dass ihr Beitrag nicht irgendwo im Closed-Source-Bereich landet? Dann wären Lizenzen mit einem starken Copyleft wie GPLv3 oder AGPLv3 eine sinnvolle Entscheidung.
- Entwickelst du Projekte innerhalb einer Community, beispielsweise Softwarebibliotheken für eine bestimmte Programmiersprache? Dann ist es wahrscheinlich eine gute Idee, die beliebteste Lizenz dieser Community zu nutzen – allein schon um Inkompatibilitätsprobleme zu vermeiden.
- Möchtest du vielleicht Unternehmen ansprechen, damit sie deine Software nutzen oder auch weiterentwickeln (und eventuell deine Hilfe dazu einkaufen)? Dann könnte eine permissive Lizenz mit einer Patentnutzungsklausel das Mittel der Wahl sein, wie beispielsweise Apache 2.0.

Für welche Lizenz du dich schlussendlich auch entscheidest: Sie sollte den Zweck deines Projekts bestmöglich unterstützen.



#### Achtung bei Quellcode in Dokumentationen

Wenn du eine Dokumentation für deine Software schreibst und Software und Dokumentation unterschiedlich lizenziert sind (beispielsweise die Dokumentation unter Creative Commons und die Software unter der Lizenz Apache 2.0), achte darauf, die Quellcodebeispiele in der Dokumentation ebenfalls unter der Softwarelizenz zu lizenzieren.

Beispiel:

»Dieses Material steht unter der Creative-Commons-Lizenz Namensnennung – Weitergabe unter gleichen Bedingungen 4.0 International. Um diese Lizenz zu sehen, besuchen Sie <https://creativecommons.org/licenses/by-sa/4.0/>. Der darin enthaltene Quellcode steht unter der Apache-2.0-Lizenz.«

## Wo finde ich mehr Infos und Unterstützung zu Lizenzen?

Trotz all dieser Informationen herrschen bestimmt noch Unsicherheiten in Bezug auf Lizenzen, und du hast sicher noch Fragen dazu. Im Folgenden findest du einige Quellen, die aus meiner Sicht hilfreich sind, wenn es um die Wahl der passenden Lizenz geht oder auch darum, zu klären, ob Lizenzen kompatibel sind.

## Unterstützung bei der Wahl der richtigen Lizenz

Die Website <https://choosealicense.com/> bietet einen einfachen Einstieg in das Thema Lizenzwahl, indem sie gleich zu Anfang fragt, was genau du eigentlich willst. Die Seite ist übrigens von GitHub selbst aus der Taufe gehoben worden und wird auch auf der Plattform selber weiterentwickelt.<sup>20</sup>

20 <https://github.com/github/choosealicense.com>

Wer Texte und Bilder – keinen Softwarecode – erzeugt bzw. veröffentlichten will, kann sich auf der Website <https://creativecommons.org/choose/> die einzelnen Creative-Commons-Lizenzen anschauen bzw. bekommt Hilfestellung für die Wahl der richtigen Lizenz.

Wer schon weiß, dass er eine Copyleft-Lizenz aus dem GPL-Universum nutzen möchte, findet auf der Seite <https://www.gnu.org/licenses/license-recommendations.html> Empfehlungen dazu, welche Lizenz für welchen Zweck am besten geeignet ist.

GNU AGPLv3	Permissions	Conditions	Limitations
<p>Permissions of this strongest copyleft license are conditioned on making available complete source code of licensed works and modifications, which include larger works using a licensed work, under the same license. Copyright and license notices must be preserved. Contributors provide an express grant of patent rights. When a modified version is used to provide a service over a network, the complete source code of the modified version must be made available.</p>	<ul style="list-style-type: none"><li>● Commercial use</li><li>● Distribution</li><li>● Modification</li><li>● Patent use</li><li>● Private use</li></ul>	<ul style="list-style-type: none"><li>● Disclose source</li><li>● License and copyright notice</li><li>● Network use is distribution</li><li>● Same license</li><li>● State changes</li></ul>	<ul style="list-style-type: none"><li>● Liability</li><li>● Warranty</li></ul>
<a href="#">View full GNU Affero General Public License v3.0 »</a>			
GNU GPLv3	Permissions	Conditions	Limitations
<p>Permissions of this strong copyleft license are conditioned on making available complete source code of licensed works and modifications, which include larger works using a licensed work, under the same license. Copyright and license notices must be preserved. Contributors provide an express grant of patent rights.</p>	<ul style="list-style-type: none"><li>● Commercial use</li><li>● Distribution</li><li>● Modification</li><li>● Patent use</li><li>● Private use</li></ul>	<ul style="list-style-type: none"><li>● Disclose source</li><li>● License and copyright notice</li><li>● Same license</li><li>● State changes</li></ul>	<ul style="list-style-type: none"><li>● Liability</li><li>● Warranty</li></ul>
<a href="#">View full GNU General Public License v3.0 »</a>			

Abbildung 5-3: Die Website <https://choosealicense.com> listet sehr übersichtlich die einzelnen Lizenzen und ihre Eigenschaften auf.

## Tools und Informationen zu/über Lizenzen

Die Website <https://opensource.org/licenses> bietet neben zahlreichen Informationen zum Thema Lizenzen auch eine Frage-und-Antwort-Sektion sowie alle derzeit zugelassen Open-Source-Lizenzen im Detail. Sucht man eine etwas ausgefältere Lizenz, ist man hier an der richtigen Adresse.

Die Website <https://opensource.guide/legal/><sup>21</sup> ist ein Tutorial zum Thema Open-Source-Lizenzierung, erstellt und kuratiert direkt von GitHub. Wen das Thema Open-Source-Lizenzierung im Firmenkontext interessiert, sollte hier mal rein-schauen.

Du verwendest Quellcode und/oder Bibliotheken von anderen und hast Sorge, dass du Lizenzen eventuell falsch benutzt? Der Dienst FOSSA bietet nach eigenen Angaben für Privatpersonen kostenlos die Möglichkeit, sich in den Entwicklungs-

21 Auch diese Seite wird auf GitHub gehostet: <https://github.com/github/opensource.guide>.

Workflow einzuklinken und bei jedem Commit auf mögliche Lizenzprobleme zu prüfen (<https://www.fossa.com>).<sup>22</sup>

Die Macher\*innen von FOSSA haben zudem die Seite <https://tldrlegal.com> erstellt, auf der viele populäre Softwarelizenzen in einfachem Englisch dargestellt werden. Hier findet man nicht nur Open-Source-Lizenzen, sondern beispielsweise auch die Nutzungsbedingungen von YouTube.

Wenn du Code von anderen Projekten einbindest (z.B. in Form von Bibliotheken) oder ein neues Projekt aus einem bestehenden Projekt erzeugst, solltest du auf die Kompatibilität der Lizenzen achten. Jede größere Lizenz bietet dazu in der Regel entsprechende Übersichten an (z.B. für GNU GPL<sup>23</sup> oder Apache<sup>24</sup>). In der englischsprachigen Wikipedia gibt es darüber hinaus eine Gesamtübersicht über viele gängige Lizenzen und ihren Kompatibilitäten: [https://en.wikipedia.org/wiki/Comparison\\_of\\_free\\_and\\_open-source\\_software\\_licenses](https://en.wikipedia.org/wiki/Comparison_of_free_and_open-source_software_licenses).

Wenn du bei Material, das unter Creative Commons lizenziert ist, korrekt die Namensnennung vornehmen willst, bietet die Seite [https://wiki.creativecommons.org/wiki/Best\\_practices\\_for\\_attribution](https://wiki.creativecommons.org/wiki/Best_practices_for_attribution) einige Beispiele für gutes, mittelmäßiges und auch falsches Benennen an.



#### Tipp: Richtige Namensnennung bei CC-Lizenzen

Wer die Namensnennung bei Werken unter einer CC-Lizenz richtig vornehmen will, sollte sich das Akronym TASL merken. Dieses steht für *Titel*, *Autor\*in*, *Source* (Quelle) und *Lizenz*. Wer diese vier Elemente angibt, hat schon einmal so gut wie alles richtig gemacht.

## Eine Lizenz zu einem Repository hinzufügen

Du hast jetzt eine Lizenz für dein Projekt ausgesucht und stellst dir die nicht ganz unerhebliche Frage: Wie kommt die Lizenz in das entsprechende Repository?

Ein Weg, eine Lizenz für ein Repository hinzuzufügen, ist, das gleich beim Erstellen vorzunehmen. Beim Anlegen eines neuen Repositorys kann man mittels Drop-down (siehe Abbildung 3-7 in Kapitel 3, Drop-down-Feld *Add a license*) aus einer Reihe an vorgefertigten Lizenzen auswählen.

Falls du bereits ein Repository ohne Lizenz erstellt hast, funktioniert das nachträglich ebenfalls sehr einfach. Erstelle in deinem Repository eine neue Datei und nenne sie *LICENSE* (*LICENSE.md* geht auch). Sobald du den Namen der Datei eingetippt hast, sollte rechts ein neuer Button *Choose a license template* erscheinen (siehe Abbildung 5-4).

---

22 Auch hierzu findet man etwas auf GitHub: <https://github.com/fossas/fossa-cli>.

23 <https://www.gnu.org/licenses/license-list.html#SoftwareLicenses>

24 <https://www.apache.org/legal/resolved.html>

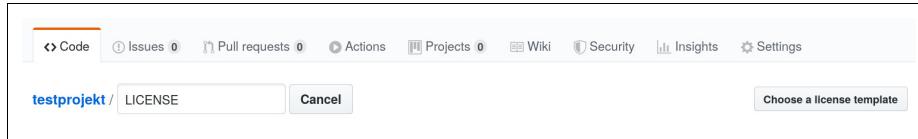


Abbildung 5-4: Sobald das Wort »LICENSE« beim Anlegen einer neuen Datei auftaucht, bietet GitHub einen neuen Button an.

Wenn du darauf klickst, siehst du die Übersicht aus Abbildung 5-5. Dort kannst du die für dich passende Lizenz auswählen. Ich schätze an diesem Weg besonders, dass man hier die Lizenzen noch mal im Detail sieht – welche Berechtigungen (*Permissions*) und welche Einschränkungen (*Limitations*) sie vergeben und unter welchen Konditionen (*Conditions*) das Werk genutzt werden darf.

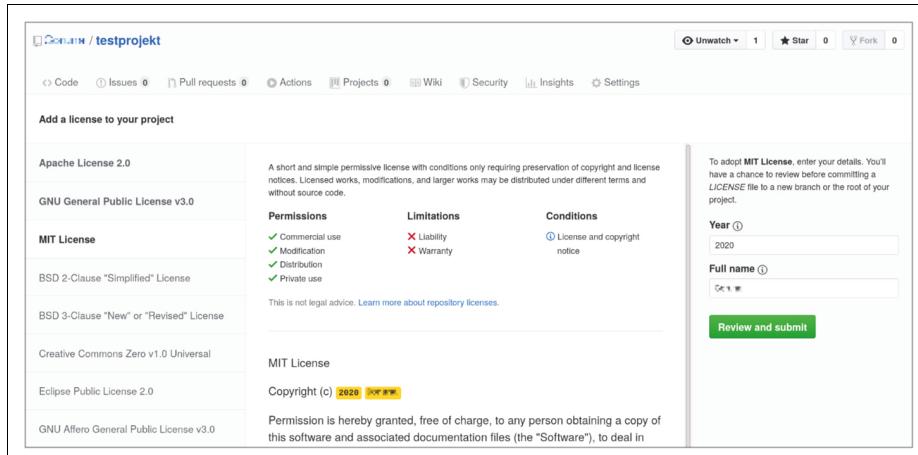


Abbildung 5-5: Man kann sich die einzelnen Lizenz-Templates vorher genau anschauen.

Sobald du eine Lizenz eingerichtet hast – egal auf welchem Weg –, ist auf der Startseite deines Projekts die gewählte Lizenz sicht- und anklickbar (siehe Abbildung 5-6).

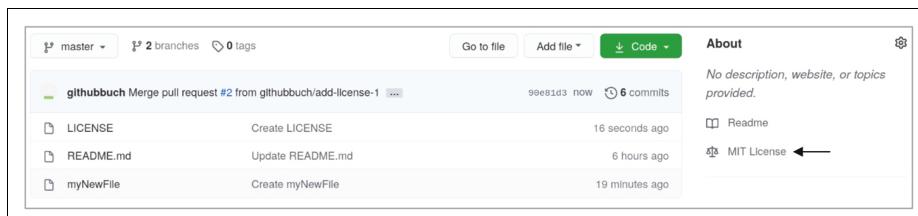


Abbildung 5-6: Die gewählte Lizenz ist auf der Startseite des Projekts zu sehen und anklickbar.

Ein Sache möchte ich zwar nicht vertiefen, aber auch nicht unerwähnt lassen: Wir haben bisher nur von *einer* Lizenz pro Projekt gesprochen, was suggerieren könnte, dass eine Software auch nur eine haben kann. Das ist so nicht korrekt, man kann eine Software auch mehrfach lizenziieren. Das Wieso, Weshalb und Wie würde hier aber zu weit gehen. Daher möchte ich dir hier eine Webrecherche nahelegen, falls das Thema für dich überhaupt relevant ist.

Glücklicherweise haben wir den rechtlichen Teil endlich geschafft. Im nächsten Kapitel wollen wir schauen, wie wir Menschen dazu motivieren, uns bei unserem Projekt zu unterstützen.



## KAPITEL 6

# Unterstützung für GitHub-Projekte finden

Wenn du dich auf GitHub bewegst, ist die Wahrscheinlichkeit recht hoch, dass du entweder Unterstützer für dein eigenes Projekt suchst, z.B. Conributoren, die dich auf Fehler aufmerksam machen, oder Maintainerinnen, die dich bei der Weiterentwicklung und Administration unterstützen. Oder du suchst ein fremdes Projekt, in das du dich selbst einbringen kannst, z.B. für den Einstieg als Contributor oder mittelfristig als Maintainer für ein Projekt. Hier lernen wir den Begriff *Fork* (deutsch »Gabelung«) kennen. Dieses Kapitel wird beide Themenbereiche adressieren.



### Am Ende des Kapitels kannst du ...

- dein Projekt auf GitHub besser auffindbar machen.
- dein Projekt attraktiver für potenzielle Mitstreiter machen.
- einem Maintainer Zugriff auf dein Repository gewähren.
- fremde Projekte zum Mitmachen besser finden.
- fremde Projekte besser dahin gehend beurteilen, ob ein Mitmachen für dich das Richtige ist.
- bei einem fremden Projekt einen Änderungsvorschlag (Pull-Request) einreichen.
- ein geforktes Repository aktuell halten.

## Wie bringt man Leute dazu, beim eigenen Projekt mitzumachen?

Das Wichtigste, um Unterstützung von anderen zu bekommen, ist, dass Menschen dein Projekt finden, verstehen, wofür es gut ist, und optimalerweise auch nutzen (etwa eine Smartphone-App, die installiert und eingesetzt wird). Nutzerinnen und Nutzer werden dann häufiger Fehler finden, sich neue Features wünschen und so viel eher Conributoren werden. Es gibt aber auch Menschen, die unterstützen

wollen, auch ohne dass sie dein Projekt nutzen.<sup>1</sup> Um Menschen dafür zu gewinnen, dass sie dein Projekt nutzen und/oder unterstützen, sind diese Aspekte wichtig:

1. **Gefunden werden** – Dein Projekt sollte für Interessierte leicht auffindbar sein.
2. **Verstanden werden** – Dein Projekt sollte so anschaulich beschrieben sein, dass man leicht erkennen kann, wozu es dient.
3. **Bekannt werden** – Dein Projekt sollte in allen relevanten Kanälen bekannt gemacht werden.

Natürlich sollte dein Projekt auch aktiv betreut werden, damit eingehende Fehlermeldungen oder Wünsche auch bearbeitet werden.

## Dein Projekt auffindbar machen

Da wir uns in einem GitHub-Buch befinden, gehe ich davon aus, dass du dein Projekt dort weiterentwickelst und auch dort veröffentlichtst. GitHub bietet ein paar Möglichkeiten, dein Projekt leichter auffindbar zu machen.

Du solltest darüber hinaus aber auch über zusätzliche Optionen nachdenken, da die breite Masse GitHub nicht nutzt, weil es ihnen zu unübersichtlich erscheint<sup>2</sup> und/oder die Nutzung bzw. Installation ein gewisses Know-how voraussetzt, was viele abschrecken könnte.<sup>3</sup> Daher schauen wir uns an, wo eine weitere Veröffentlichung gegebenenfalls noch Sinn ergeben kann.

### Besser gefunden werden auf GitHub: Topics

Indem du deine Repositories mit entsprechenden Etiketten versiehst – den sogenannten Topics –, können sie leichter gefunden werden. Topics können etwa der Zweck deines Projekts, das Fachgebiet, wichtige Eigenschaften, eingesetzte Technologien und vieles mehr sein. Andere Menschen, die sich beispielsweise ebenso wie du mit einer »App für ein Star-Trek-Rollenspiel« oder »Widgets für KDE-Plasma« beschäftigen, können über entsprechende Topics dann das Projekt finden. Ich zeige dir jetzt, wie du Topics erstellst und wie du nach Topics suchen kannst.

Das Erstellen von Topics ist einfach. Auf der rechten Seite des Repositorys macht GitHub kenntlich, dass das Projekt noch keine Topics hat (siehe Abbildung 6-1). Wähle das Zahnradsymbol aus, und durch die Eingabe von TOPIC, Leertaste, TOPIC, Leertaste, TOPIC, Leertaste kannst du so lange Topics eingeben, bis dir keine mehr einfallen. GitHub schlägt dir gegebenenfalls auch Topics vor, die auf einer Analyse des Repository-Inhalts basieren.<sup>4</sup> Du bist komplett frei in der Wahl deiner Topic-Namen.

---

1 Eine meiner ersten Unterstützungen auf GitHub war Hilfe bei der Übersetzung einer Software, die ich selber nicht nutze.

2 Wie bei uns am Anfang ;).

3 Schon einmal versucht, eine Smartphone-App von GitHub manuell herunterzuladen und auf das eigene Smartphone zu installieren?

4 Gilt aber nur für öffentliche Repositories, private werden nicht analysiert.



### Tipp: (Auch) Bestehende Topic-Namen verwenden

Was nutzen einem die kreativsten Topics, wenn niemand danach sucht? Insofern solltest du wenigstens ein bis zwei gängige und natürlich zu deinem Projekt passende Topics wählen.

Dafür kannst du dir die Topics von verwandten Projekten abkupfern und/oder zur Inspiration die kuratierte Topic-Liste auf GitHub anschauen.<sup>5</sup>

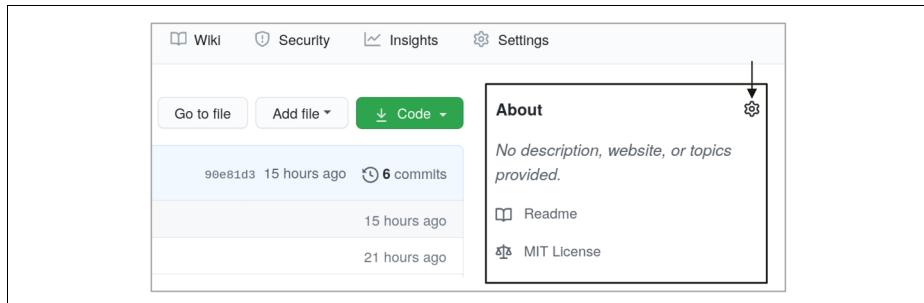


Abbildung 6-1: GitHub gibt einen Hinweis, wenn Topics und/oder die Beschreibung eines Projekts fehlen. Über das Zahnradsymbol lassen sich Anpassungen vornehmen.

Falls du selbst nach Topics suchen möchtest, geht das entweder über *Explore* im GitHub-Menüband oder direkt auf <https://github.com/topics>. Eine andere Möglichkeit besteht darin, bei einem Projekt mit Topics einfach auf den entsprechenden Topic-Namen zu klicken.

Bei unserem bisherigen Beispielprojekt *moby* kannst du sehen, wie die Topics später aussehen (siehe Abbildung 6-2). Dabei ist *docker* eine Containersoftware, *containers* soll uns ebenfalls auf das Thema Container hinweisen, und *go* ist eine Programmiersprache. Solltest du für dein Projekt ebenfalls *go* gewählt haben, könnte jetzt jemand durch Anklicken des Topics dein Projekt und die 1.000 anderen Projekte mit demselben Topic finden.

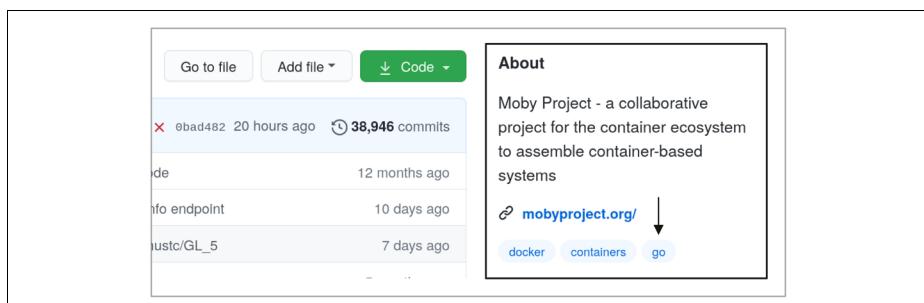


Abbildung 6-2: Mit ein paar aussagekräftigen Topics kann ein Projekt leichter gefunden werden.

5 <https://github.com/github/explore>

## Andere Plattformen

Vielleicht gibt es aber für dein Projekt auch noch andere Plattformen als GitHub, die für eine Veröffentlichung infrage kommen? Für eine Smartphone-App gibt es App-Stores, über die entsprechende Apps verbreitet werden können, wie beispielsweise F-Droid<sup>6</sup> oder Google Play Store für Android-basierte Smartphones.

Manche Zeitschriftenverlage bieten die Möglichkeit, Software herunterzuladen, beispielsweise Heise<sup>7</sup> oder Chip<sup>8</sup>. Es lohnt sich eventuell, hier mal nachzufragen, ob sie deine Software mit ins Portfolio nehmen.

Für reine Textveröffentlichungen gibt es ebenfalls geeignete Plattformen<sup>9</sup> oder auch das eigene Blog. Eine Plattform muss aber nicht unbedingt im Internet sein. Eine Software auf CD, DVD, USB-Stick, SD-Karte etc. zu kopieren und weiterzugeben, kann im Einzelfall genauso seinen Zweck erfüllen.

Einige Projekte werden zusätzlich auf den Konkurrenzprodukten von GitHub veröffentlicht, beispielsweise GitLab oder BitBucket (wird auch *Mirroring* genannt). Sollte das für dich infrage kommen, lautet meine Empfehlung: Nutze eine der Plattformen als »Hauptstandort« und unterbinde auf den gespiegelten »Nebenstandorten« – wenn möglich – das Anlegen von Issues und Pull-Requests. Mach auf allen Plattformen sehr deutlich, auf welcher Plattform die aktuelle Version deines Projekts liegt und du Unterstützung im Sinne von Issues und Pull-Requests erwartest. Ansonsten verwirrst du unterstützungswillige Menschen nur unnötig und hast zudem das Risiko, dass dir etwas durchrutschen könnte.

## Eigene Website

Für die Veröffentlichung eines Projekts empfiehlt es sich, eine eigene Website einzurichten. Diese sollte idealerweise die »Heimatbasis« für das Projekt sein, sodass du bei Veröffentlichungen immer wieder darauf verweisen kannst und Menschen eine Anlaufstelle haben. Optimalerweise ist die Website so ausgerichtet, dass sie auch die nicht ganz so technikaffinen Menschen anspricht.

Im Abschnitt »Websites aus GitHub generieren (GitHub Pages)« auf Seite 225 in Kapitel 10 zeige ich dir, wie du eine Website basierend auf deinem GitHub-Repository anlegen kannst, auch wenn du vorher noch nie eine Website selbst gebaut hast.

## Dein Projekt anschaulich beschreiben

Das A und O, um Menschen von deinem Projekt zu überzeugen, ist, ihnen klarzumachen, was dein Projekt eigentlich genau macht und wofür es gut ist, und ihnen

6 <https://f-droid.org/>

7 <https://www.heise.de/download/>

8 [https://www\(chip.de/download/](https://www(chip.de/download/)

9 Beispielsweise <https://medium.com> für Artikel.

dann am besten gleich auch zu sagen, wie sie es unterstützen können. In diesem Abschnitt werden wir uns dafür ausschließlich auf GitHub bewegen. Natürlich kannst (und solltest) du dein Projekt auch auf deiner Homepage oder im App-Store verständlich beschreiben, dabei hilft dir hoffentlich mein kleiner Impuls in der Erklärbärbox »Über das sinnvolle Beschreiben von Dingen« weiter unten. GitHub bietet aber noch eine Reihe an Möglichkeiten, um Projekte für andere zu verdeutlichen (insbesondere für Conributoren), und das wollen wir uns hier ansehen.

Die *README.md* als ersten Einstiegspunkt für dein GitHub-Repository hast du schon kennengelernt. Darüber hinaus unterstützt dich GitHub durch eine Checkliste darin, den Überblick über die wichtigsten Dokumente und Vorlagen zu behalten. Diese Dokumente nennen sich *Community Health Files* (deutsch etwa »Community-Gesundheitsdateien«). Klicke hierfür in deinem Repo auf *Insights* und dann auf *Community*. Dort bekommst du eine Übersicht der Dokumente und siehst auch, inwieweit du diese Möglichkeiten bereits genutzt hast (siehe Abbildung 6-3). Das sind keine Vorgaben, eher Best Practices. Du kannst dich auch dazu entschließen, das eine oder andere Dokument nicht in dein Projekt zu integrieren.

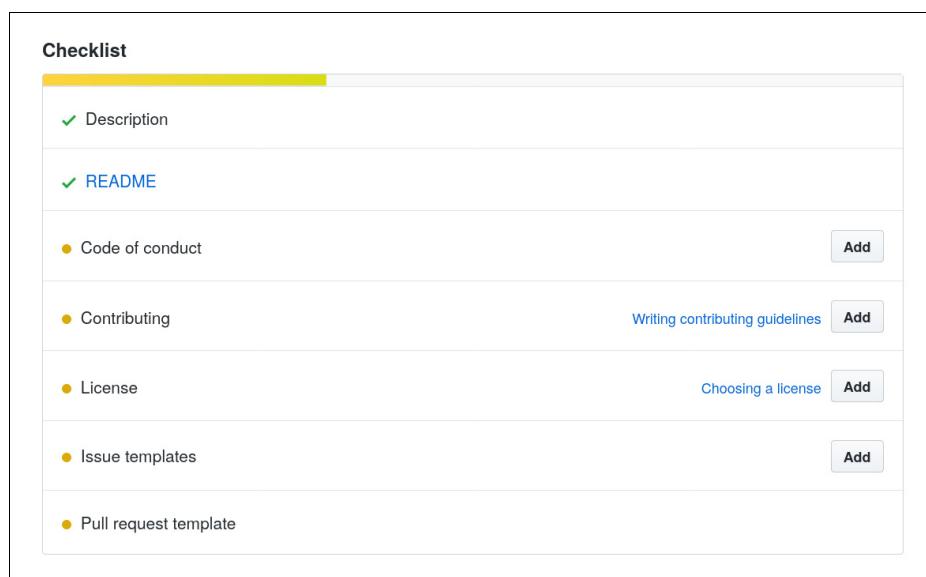


Abbildung 6-3: Eine Checkliste hilft, an die wichtigsten Dokumente zu denken.

## Description

Die *Description* (deutsch »Beschreibung«) ist eine kurze Beschreibung deines Repositorys mit der Möglichkeit, optional eine Website anzugeben. Dargestellt werden Beschreibung und Website in der Zeile direkt über den Topics (siehe auch Abbildung 6-2). Einrichten kannst du beides, indem du auf das Zahnradsymbol klickst, wie es in Abbildung 6-1 ganz rechts zu sehen ist.

Auf den ersten Blick wirkt die *Description* innerhalb des Repositorys nicht sehr überzeugend. Sie ist sehr klein, und man kann sie leicht übersehen. Interessant wird es aber, wenn zu einem Projekt bereits mehrere Topics angegeben wurden und jemand über diese Begriffe sucht (siehe Abbildung 6-4 und auch den Abschnitt »Besser gefunden werden auf GitHub: Topics« auf Seite 102). Durch die Beschreibung des dort abgebildeten Projekts (deutsch »Lernspiel über Wirtschaft und Inflation im Smartphone-Messenger-Stil«) lässt sich gut erahnen, worum es in dem Projekt geht, oder? Wäre die Beschreibung nicht da, hätte uns der Repository-Name *IN.flation* vermutlich nicht viel geholfen.

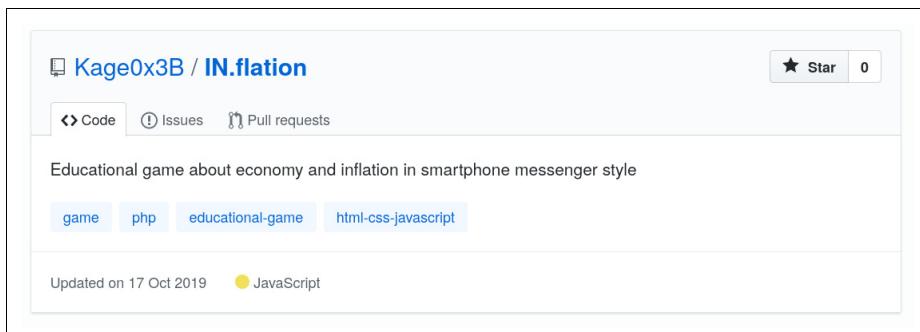


Abbildung 6-4: Die Wichtigkeit für die Beschreibung eines Repositorys sieht man bei der Suche über Topics.

## README.md

Die *README.md* haben wir schon ausführlich besprochen, und zwar in Kapitel 2 in der Erklärbärbox »*README.md*« und im Abschnitt »Das erste eigene Repository anlegen« auf Seite 24 in Kapitel 3. Ich werde hier daher nicht weiter auf sie eingehen.

## Über das sinnvolle Beschreiben von Dingen

In der Regel möchte man, wenn man eine App, ein Add-on oder Ähnliches veröffentlicht, dass andere Menschen die Sache gut finden und auch nutzen. Häufig genug kommt es aber vor, dass die Beschreibung so schlecht ist, dass die potenziellen Nutzer sie schulterzuckend links liegen lassen und nur die absoluten Experten<sup>10</sup> damit etwas anfangen können. Spricht mein Kunstwerk denn nicht schon für sich selber? Nein, meistens nicht, und um dir das klarzumachen, entführe ich dich kurz in eine andere Welt.

10 Also vermutlich du und deine zwei Mitentwickler\*innen.

Velleicht erinnerst du dich an Momente, in denen du ein Gerät oder Werkzeug gesehen und dich gefragt hast: »Was ist denn das?« Sieh dir hierzu Abbildung 6-5 an. Weißt du, wozu man es benutzt? Würdest du es kaufen? Wenn der Hersteller jetzt noch etwas wie »ergonomischer Griff«, »aus rostfreiem Edelstahl« oder »großzügige Variantenauswahl« dazu schreiben würde, wärest du dann schlauer? Würdest du es jetzt kaufen? Vermutlich nicht. Wie viel hilfreicher wäre es, zu wissen, dass dies ein Gerät zum Entschuppen von Fischen ist? Du magst keinen Fisch oder musst Fische nicht entschuppen? Großartig, dann weißt du, dass du dieses Werkzeug nicht brauchst, egal wie ergonomisch der Griff ist.<sup>11</sup>

Was will uns die Autorin damit sagen? Bei der Recherche für dieses Buch habe ich mir unzählige Repositories, Actions und Apps (die lernst du in Kapitel 9 noch kennen) auf GitHub angesehen. Bei (viel zu) vielen davon wurde mir erzählt, wie ergonomisch der Griff sei. Ich wusste jedoch selten oder nur mit großer Mühe, ob es etwas zum Dinosaurierhäuten oder zum Unkrautjäten ist. In der Regel interessieren sich Menschen aber weniger für das »coole Feature XY«, sondern eher für: »Welches Problem kann ich damit lösen?«

**Wir merken uns:** Wenn ich will, dass meine Arbeit von vielen genutzt wird, versuche ich, zu beschreiben, welche Probleme damit gelöst werden können.



Abbildung 6-5: Ein seltsames Gerät, wofür das wohl sein mag? Lösung siehe Text! Mit freundlicher Genehmigung der Firma G.W.P. Manufacturing Services AG, [www.gwp-ag.de](http://www.gwp-ag.de).

### Code of Conduct

*Code of Conduct* bedeutet im Deutschen »Verhaltenskodex« und beschreibt, wie sich die Menschen, die etwas zu deinem Projekt beitragen möchten, verhalten sollten. Mit einem solchen Kodex hast du die Chance, eine offene und freundliche Atmosphäre zu schaffen, die jede Person willkommen heißt. Dadurch kannst du dein Projekt attraktiver für Conributoren machen.

<sup>11</sup> Falls du das Gerät unbedingt dein Eigen nennen möchtest, am Ende des Abschnitts sage ich dir, wo du es herbekommst.

Dein Verhaltenskodex sollte idealerweise folgende Fragen beantworten:

- Für welche Bereiche ist der Verhaltenskodex gültig? Geht es um Issues und Pull-Requests, oder zählen auch Veranstaltungen im Rahmen des Projekts dazu?
- Für wen gilt der Verhaltenskodex? Beispielsweise nur für Maintainerinnen oder auch für Conributoren? Was ist mit Anwenderinnen und Sponsoren?
- Was passiert, wenn jemand gegen den Verhaltenskodex verstößt?
- Wie kann jemand Verstöße melden? Wie wird damit umgegangen, wenn jemand einen Verstoß melden möchte, der von der Meldestelle begangen wurde?



#### Achtung: Durchsetzung des Kodex

Falls du einen Verhaltenskodex etablieren möchtest, sei dir bewusst, dass du diesen auch konsequent, transparent und für alle gleichermaßen durchsetzen musst. Ansonsten schadet er mehr, als er nützt.

Nichts ist schlimmer, als Regeln vorzugeben und diese bei erstbester Gelegenheit, oder weil es den besten Kumpel betrifft, zu ignorieren. So geht Vertrauen – und damit auch die Beteiligung an deinem Projekt – schneller verloren, als du »Expelliarmus«<sup>12</sup> sagen kannst.

Der Verhaltenskodex lässt sich ähnlich leicht wie eine Lizenz anlegen (siehe Kapitel 5 im Abschnitt »Eine Lizenz zu einem Repository hinzufügen« auf Seite 97): Erstelle eine Datei namens *CODE\_OF\_CONDUCT.md* im Haupt-, *docs-* oder *.github*-Verzeichnis, und GitHub zeigt dir wieder einen Button für die Wahl eines Templates an (siehe Abbildung 6-6).



Abbildung 6-6: Sobald der Begriff »CODE\_OF\_CONDUCT« beim Anlegen einer neuen Datei auftaucht, bietet GitHub einen neuen Button an.

Wenn du auf den Button klickst, bekommst du (derzeit) zwei Verhaltenskodizes angezeigt (siehe Abbildung 6-7), den *Contributor Covenant Code of Conduct*<sup>13</sup> und den *Citizen Code of Conduct*<sup>14</sup>.

Falls du einen der beiden Kodizes wählst, kannst du diesen noch anpassen, im Beispiel mit einer E-Mail-Adresse beim *Contributor Covenant Code of Conduct* (rechts im Bild). Du kannst aber auch einen Verhaltenskodex von anderer Stelle übernehmen oder deinen ganz eigenen entwickeln. Dann lässt du den Template-Button einfach links liegen, erstellst die leere Datei *CODE\_OF\_CONDUCT.md* und füllst sie mit einem eigenen Text.

12 Entwaffnungszauberspruch bei Harry Potter.

13 <https://www.contributor-covenant.org/>

14 <http://citzencodeofconduct.org/>

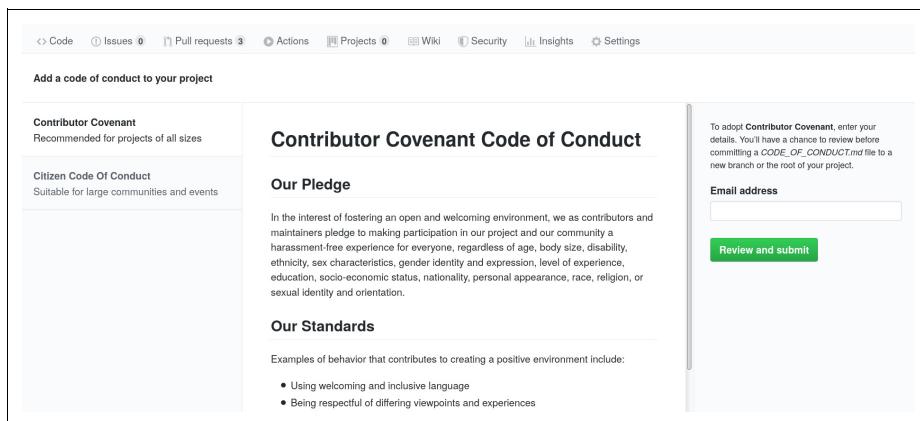


Abbildung 6-7: GitHub bietet zwei Vorlagen für einen Verhaltenskodex an.

Die von GitHub bereitgestellten Kodizes gibt es nur auf Englisch. Um die meisten Menschen zu erreichen, ist der Einsatz dieser Sprache auch sinnvoll. Je nach Projekt kann es aber gewünscht sein, den Verhaltenskodex auch in einer anderen Sprache anzubieten. Hierfür kannst du entweder selbst die Übersetzung vornehmen oder sie von anderen Repositories übernehmen.<sup>15</sup>

Als letzten Schritt solltest du deinen Verhaltenskodex leichter auffindbar machen, z.B. indem du ihn in der *README.md* oder anderen Dokumenten wie beispielsweise den Vorlagen für Issues und Pull-Requests verlinkst. Eine weitere Möglichkeit ist die Veröffentlichung auf deiner Website.

## Contributing

*Contributing* bedeutet übersetzt »beitragen« oder »mitmachen«. Um Menschen wissen zu lassen, wie sie dein Projekt unterstützen können, kannst du eine spezielle Datei *CONTRIBUTING.md* anlegen, in der alles Wichtige festgehalten ist.<sup>16</sup> Wichtig ist, dass sich diese Datei entweder im Hauptverzeichnis oder in einem der Unterordner *docs* oder *.github* befindet.

Während die *README.md* eine breite Zielgruppe anspricht, wie beispielsweise Anwenderinnen oder generell am Projekt Interessierte, adressiert *CONTRIBUTING.md* vor allem potenzielle Contributoren.

Immer wenn jemand einen Issue oder Pull-Request aufmacht, wird er oder sie auf die Datei hingewiesen (siehe auch Abbildung 6-8). Optimalerweise sollte die Datei folgende Informationen enthalten:

- Die Information, ob Unterstützung erwünscht ist und, wenn ja, in welchen Bereichen.

<sup>15</sup> Beispiel für ein Repository mit zweisprachigem Verhaltenskodex: <https://github.com/zauberware/code-of-conduct>.

<sup>16</sup> GitHub ist bei der Benennung der Datei nicht zimperlich, *contributing.txt* würde auch funktionieren.

- Optimalerweise Beispiele dazu, wie unterstützt werden kann.
- Informationen darüber, wie ein guter Issue oder Pull-Request aussehen sollte, und die dafür notwendigen Schritte (oder du verwendest dafür Templates, siehe Abschnitte »Vorlagen für Issues« auf Seite 75 und »Vorlagen für Pull-Requests« auf Seite 79 in Kapitel 4).
- Verweise auf weitere Dokumentationen und Kommunikationsmöglichkeiten (beispielsweise einen Chatkanal, der genutzt wird).
- Wie sich Unterstützer\*innen verhalten sollten (z.B. über eine Verlinkung mit dem Verhaltenskodex, siehe Abschnitt oben).

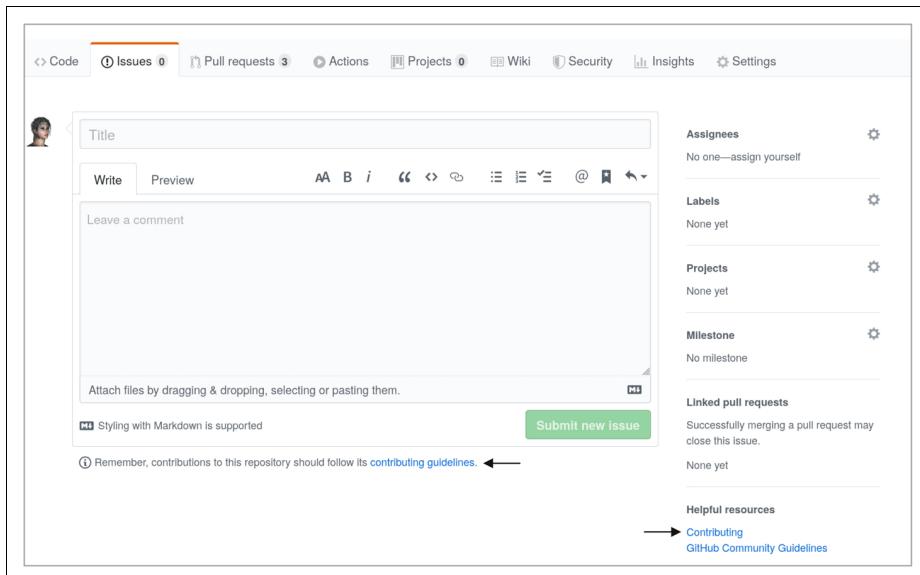


Abbildung 6-8: Existiert eine CONTRIBUTING.md, wird die Datei in Issues und Pull-Requests verlinkt.

GitHub bietet für die CONTRIBUTING.md keine Vorlagen an. Ich empfehle dir, einfach mal bei anderen Projekten zu schauen, wie diese Datei dort aufgebaut ist.

## License, Issue-Template und Pull-Request-Template

Das Hinzufügen von Lizzenzen sowie Issue- und Pull-Request-Templates haben wir bereits besprochen:

- Hinzufügen einer Lizenz zum Projekt im Abschnitt »Eine Lizenz zu einem Repository hinzufügen« auf Seite 97 in Kapitel 5
- *Issue-Templates* im Abschnitt »Vorlagen für Issues« auf Seite 75 in Kapitel 4
- *Pull-Request-Templates* im Abschnitt »Vorlagen für Pull-Requests« auf Seite 79 in Kapitel 4

## Weitere Tipps, um dein Projekt attraktiv zu machen

Um Menschen zum Mitmachen zu animieren, ist es wichtig, deutlich zu machen, dass du dich über Hilfe freust. Dabei unterstützt dich beispielsweise das Projekt »Make a Pull Request«<sup>17</sup>. Das Projekt stellt zum einen eine Art Aufkleber oder Banner (auch »Badge« genannt) namens *PRs welcome* für die *README.md* deines Repositorys zur Verfügung, um deinen Wunsch nach Unterstützung deutlich zu machen (Badges lernst du im Abschnitt »Zeigen, wo man steht – Badges« auf Seite 256 in Kapitel 11 noch etwas genauer kennen). Zum anderen bietet das Projekt eine Reihe von kostenlosen Tutorials an, beispielsweise wie man einen Pull-Request macht oder wie man Git in Verbindung mit GitHub nutzt – insbesondere für noch unerfahrene potenzielle Conributoren eine gute Einstiegshilfe. Den Link auf diese Tutorials kannst du in deiner *CONTRIBUTING.md* veröffentlichen.

Eine gute Möglichkeit, Menschen direkt zum Mitmachen zu bewegen, ist das Anlegen von Issues. Hier kannst du anderen gezielt zeigen, wo noch etwas gemacht werden kann. Es empfiehlt sich daher, die Issues so gut zu beschreiben, dass jemand anderer auch genau versteht, was getan werden soll. Zusätzlich kannst du den Issue mit einem ansprechenden Label versehen (das haben wir im Abschnitt »Den ersten Ablauf üben – Issue anlegen und bearbeiten« auf Seite 31 bereits ausprobiert). Zum Beispiel können die beiden Labels *good first issue* (»guter erster Issue«) und *help wanted* (»Hilfe benötigt«), die GitHub bereits von Haus aus bereitstellt, eine geeignete Beschriftung sein, um Interessierte anzulocken (siehe auch Abbildung 3-16 in Kapitel 3).

Bevor wir zum letzten Schritt kommen – dein Projekt bekannt zu machen –, halte ich noch mein Versprechen, dir die Bezugsadresse des Fischschuppers<sup>18</sup> zu nennen.

## Dein Projekt bekannt machen

Dein Projekt bekannt zu machen, ist aus meiner Sicht der schwierigste Teil. Du hast vermutlich kein millionenschweres Werbebudget, und wenn du nur Eigenwerbung betreibst, ist der Grat schmal zwischen »Werbung« und »Spam«. Ich kann dir daher auch nur ein paar Denkanstöße dazu geben.

Auf GitHub habe ich ein Repository gefunden, das ein »Cheat Sheet«<sup>19</sup> (deutsch »Spickzettel«) bereitstellt, in dem du finden kannst, wie man Open-Source-Projekte am besten bewirbt. Ich empfehle dir, dort mal vorbeizuschauen. Zwei Punkte daraus möchte ich in diesem Abschnitt hervorheben.

---

17 <http://makeapullrequest.com/>

18 <https://edelstahlregale.jimdofree.com/shop/fischschupper/>

19 <https://github.com/zenika-open-source/promote-open-source-project>

## Namenswahl

Gib deinem Projekt einen aussagekräftigen Namen. Optimal wäre es, wenn man direkt am Namen erkennt, wofür das Projekt steht.

- Fiktives Beispiel: Du würdest unter dem Namen »Booom!« vermutlich keine Software für die Steuererklärung erwarten, sondern eher einen billigen Ego-shooter aus den 1990ern.
- Reales Beispiel: Eine E-Mail-App namens »FairEmail«<sup>20</sup>, die besonderen Wert auf Datenschutzfreundlichkeit legt, kann man schon am Namen erkennen.

Achte aber darauf, dass der Name nicht zu trivial ist, beispielsweise Alltagsgegenstände (egal in welcher Sprache) enthält. Sonst ist die Websuche keine große Freude. Wenn der Name unbedingt einen Alltagsgegenstand wie z.B. Buch enthalten soll, ist es sinnvoll, ihn etwas abzuwandeln, z.B. in »Rent\_A\_Book« oder »BuchVorleser«.

Ist dein Projekt in irgendeiner Fachdomäne angesiedelt, z.B. Fantasy-Rollenspiel, Energiewirtschaft oder Karate? Dann ist es gut, Fachbegriffe aus dieser Domäne mit einzubinden, eventuell in abgewandelter Form, z.B. »Würfelhelfer«, »Energie-wende-Kalkulator« oder »Practice-your-Kata«. Fachanwender\*innen werden dein Projekt dann leichter finden und auch zuordnen können.

## Projekt aktiv bewerben

Um dein Projekt aktiv zu bewerben, bieten sich Foren und Social-Media-Kanäle an. Du könntest beispielsweise in einem Forum die Website deines Projekts in deine Signatur schreiben. Oder in einem Social-Media-Kanal regelmäßig informieren, wenn es ein neues Update für dein Projekt gibt – aber natürlich niemals wahl- und zusammenhanglos.

Wo und wie du dein Projekt bewerben kannst, zeigen dir die nachfolgenden Möglichkeiten. Es ist eine Ideensammlung und natürlich keine vollständige Liste:

- Foren: Stackoverflow<sup>21</sup>, Reddit<sup>22</sup> etc.
- Social Media: Mastodon<sup>23</sup>, Twitter<sup>24</sup>, Facebook<sup>25</sup> etc.
- Dokumentationen und/oder Demonstrationen erstellen und auf Videoplattformen hochladen: Vimeo<sup>26</sup>, YouTube<sup>27</sup> etc.
- Artikel schreiben in Blogs: im eigenen Blog oder in Blogs, die sich mit verwandten Themen beschäftigen

20 <https://email.faircode.eu/>

21 <https://stackoverflow.com/>

22 <https://www.reddit.com/>

23 <https://joinmastodon.org/>

24 <https://twitter.com/>

25 <https://www.facebook.com/>

26 <https://vimeo.com/de/>

27 <https://youtube.de/>

Eine zusätzliche Möglichkeit besteht darin, sich eine entsprechende Community zu suchen und dort dein Projekt bekannt zu machen. Das muss nicht zwingend eine technisch ausgerichtete Community sein. Wenn du beispielsweise eine Smartphone-App für Theaterbesucher\*innen baust, könnte der »Verein der Theaterfreund\*innen« die richtige Anlaufstelle sein<sup>28</sup>. Eine gute Gelegenheit, Kontakt zu einer solchen Community aufzubauen, sind Veranstaltungen wie Konferenzen oder Kongresse. Eine gute Anlaufstelle ist etwa ein Vortrag über das Projekt. Im Abschnitt »Suche über direkten Kontakt mit Maintainerinnen« auf Seite 123 weiter unten habe ich ein paar Veranstaltungen aufgeführt, die eventuell für dein Projekt infrage kommen könnten.



#### Tipp: Insights – Traffic

Um festzustellen, inwieweit deine Werbemaßnahmen erfolgreich sind, solltest du schauen, wie und wo (und ob überhaupt) Menschen auf deinem Projekt unterwegs sind. Das geht im entsprechenden Repository unter dem Menüpunkt *Insights* und dann *Traffic* (deutsch etwa »Datenverkehr«). Dort gibt es unterschiedliche Auswertungen, unter anderem wie viele Besucher\*innen wann dein Projekt besucht haben (siehe Abbildung 6-9).



Abbildung 6-9: Der Traffic eines Repositorys zeigt unter anderem die Anzahl an Besucher\*innen auf einer Zeitachse an.

Dein Projekt ist jetzt auffindbar, verständlich und auch (hoffentlich) bekannt. Super! Vielleicht hast du jetzt schon viele Nutzerinnen und auch einige Contributoren. Im letzten Abschnitt zum Thema »eigenes Projekt« möchte ich dir zeigen, wie du jemanden zur Maintainerin »befördern« kannst, falls du einen Contributor hast, dem du vertraust und der dich umfassender unterstützen soll, als er es bisher getan hat.

28 Falls du meinst, diese Personengruppe traue sich nicht, dich auf GitHub zu unterstützen, kannst du ihnen ja dieses Buch empfehlen ;-).

## Dein Projekt (gegebenenfalls) zugänglich machen (Rechtevergabe)

Du hast einen bis mehrere Conributoren auf deinem Projekt und möchtest jetzt eine zur Maintainerin befördern und ihr damit mehr Rechte einräumen?

### Worum es geht – die Rechte

Ein Repository, wie wir es bisher angelegt haben,<sup>29</sup> hat in der Regel zwei Berechtigungsstufen (in Klammern der Name, den GitHub dafür verwendet):

1. Projekteignerin (*Repository Owner*) – darf alles innerhalb des Repositorys
2. Maintainerin (*Collaborator*) – darf viel, aber nicht alles innerhalb des Repositorys

Maintainer\*innen haben auf deinem Repository (sei es öffentlich oder privat), nachdem du sie freigeschaltet hast, weitreichende Rechte. Sie können unter anderem:

- In das Repository schreiben (*push*), lesen (*pull*) und es kopieren (*fork*).
- Labels erstellen, zuweisen und löschen.
- Issues erstellen, schließen, wieder öffnen und jemandem zuweisen.
- Kommentare auf Commits, Pull-Requests und Issues editieren oder löschen.
- Pull-Requests öffnen, mergen und schließen.
- Wikis erstellen und editieren.
- Die Kommentarfunktion sperren (siehe Abschnitt »Für Ruhe sorgen (Teil 1) – Locking Conversations« auf Seite 80 in Kapitel 4).
- Und Weiteres.<sup>30</sup>

Eine stärkere Detaillierung ist nicht möglich. Du kannst also nicht eine Maintainerin nur auf die Issues loslassen, und eine andere darf nur das Wiki anpassen. Eine Maintainerin darf entweder alles, was oben beschrieben ist, oder sie ist keine Maintainerin, sondern eine Contributorin. Sie kann aber nicht die Einstellungen des Projekts ändern und dort beispielsweise *Required Reviews* ausschalten (das hast du im Abschnitt »Genehmigung vorschreiben – Required Reviews« auf Seite 68 in Kapitel 4 kennengelernt).

Rechte granularer zu vergeben, ist nur über Einrichtung einer Organisation (*Organization*) möglich. Wenn du hier näher einsteigen möchtest, empfehle ich dir einen Blick in die GitHub-Hilfeseite.<sup>31</sup>

<sup>29</sup> Gemeint ist damit: innerhalb eines User-Accounts – also wie wir bisher gearbeitet haben. Als Abgrenzung dazu gibt es noch Organisationsaccounts. Die schauen wir uns hier aber nicht näher an.

<sup>30</sup> Zum Nachlesen:  
<https://help.github.com/en/github/setting-up-and-managing-your-github-user-account/permission-levels-for-a-user-account-repository#collaborator-access-on-a-repository-owned-by-a-user-account>.

<sup>31</sup> <https://help.github.com/en/github/setting-up-and-managing-organizations-and-teams/about-organizations>

## Wie es geht – Rechtevergabe

Um die nachfolgenden Aktivitäten »nachzuspielen«, brauchst du entweder einen zweiten Account oder eine Freundin, die du einladen möchtest. Alternativ kannst du auch wieder nur theoretisch meinen Erklärungen und Bildern folgen.

Um einer neuen Maintainerin Zugriff auf dein Repository zu geben, kannst du unter *Settings* auf der Einstellungsseite deines Repositorys *Manage access* (deutsch »Zugriff verwalten«) auswählen (siehe auch ❶ und ❷ in Abbildung 6-10).

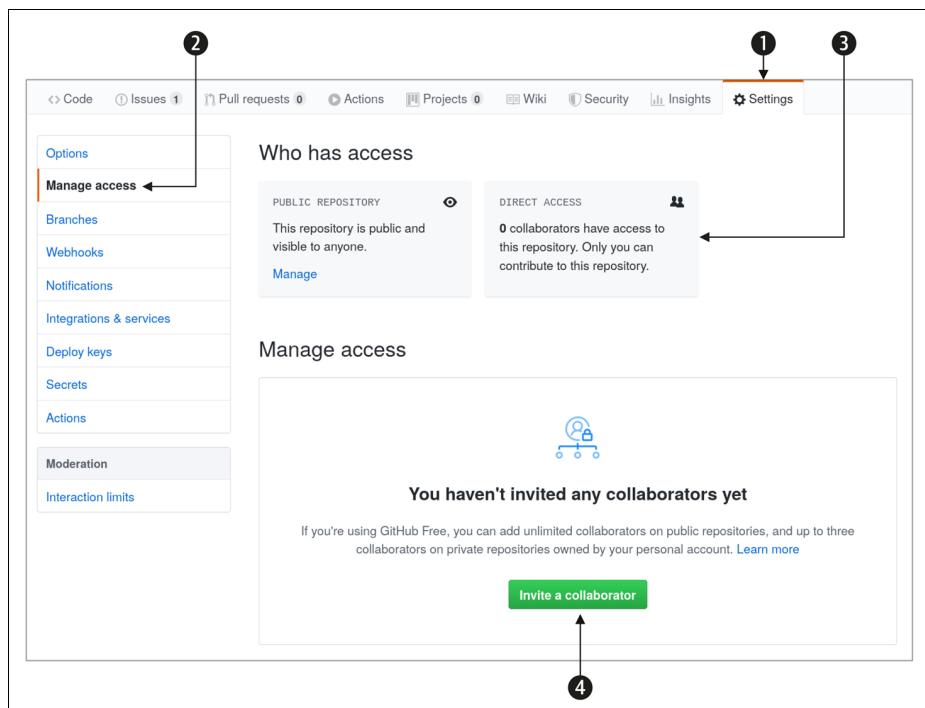


Abbildung 6-10: Alles zum Thema Rechteverwaltung findet man unter dem Menüpunkt *Manage access*.

Dort siehst du zunächst, dass in dem rechten Kasten mit dem Titel *Direct access* (deutsch »direkter Zugriff«) bisher 0 Collaborators stehen ❸. Das ändern wir jetzt. Etwas weiter unten auf der Seite klicke den Button *Invite a collaborator* (deutsch etwa »Lade jemanden zum Mitarbeiten ein«) an ❹ und gib den Account an, den du einladen möchtest.

Sobald du einen Account eingeladen hast, erscheint dieser auf der Rechteverwaltungsseite (siehe Abbildung 6-11). Hier kannst du die Einladung auch wieder löschen, und zwar über das kleine Müllimersymbol auf der rechten Seite. Der Button zum Einladen ist immer noch da, aber weiter nach rechts oben gerutscht.

The screenshot shows the 'Manage access' section of a GitHub repository. At the top right is a green button labeled 'Invite a collaborator'. Below it is a search bar with the placeholder 'Find a collaborator...'. A table lists a single entry: 'githubbuch' with a small profile picture, status 'Awaiting githubbuch's response', and a 'Pending Invite' button with a trash icon. To the left of the table is a checkbox labeled 'Select all' and a dropdown menu labeled 'Type'.

Abbildung 6-11: Der eingeladene Account erscheint auf der Rechteverwaltungsseite.

Vielleicht ist dir aufgefallen, dass dort bei dem eingeladenen Account noch *Awaiting githubbuch's response* (deutsch etwa »Warte auf Rückmeldung von User githubbuch«) und *Pending invite* (deutsch »ausstehende Einladung«) angezeigt wird. Das bedeutet, eine neue Maintainerin muss der Einladung erst zustimmen, bevor sie offiziell dazugehört.

Ich wechsle jetzt den Account, um zu zeigen, wie das auf der anderen Seite aussieht. Nach dem Log-in in den anderen Account sieht es so aus wie in Abbildung 6-12.

The screenshot shows an invitation email from GitHub. It starts with 'Einladende' pointing to a user icon and 'Eingeladene' pointing to another user icon. The main message reads: 'CUIUM invited you to collaborate'. Below it are two buttons: 'Accept invitation' (green) and 'Decline' (grey). To the left is 'Welches Projekt' (testprojekt) and to the right are 'Handlungsoptionen'. A detailed list of what the owner can see is shown: 'Owners of testprojekt will be able to see:' followed by a bulleted list: 'Your public profile information', 'Certain activity within this repository', 'Country of request origin', 'Your access level for this repository', and 'Your IP address'. At the bottom is a question 'Is this user sending spam or malicious content?' with a 'Block' button.

Abbildung 6-12: Angehende Maintainerinnen erhalten eine Einladung, um an einem Projekt mitzuwirken.

Man sieht, wer wen eingeladen hat, um welches Projekt es sich handelt (*testprojekt* in meinem Fall), und die Maintainerin bekommt Handlungsoptionen angezeigt zum Annehmen oder Ablehnen der Einladung. Sobald sie die Einladung durch Anklicken von *Accept invitation* (deutsch »Einladung akzeptieren«) annimmt, ist sie bei deinem Projekt dabei, erscheint nicht mehr als *Pending invite*, und ihr könnt gemeinsam loslegen.

# Ein Projekt finden, das du unterstützen möchtest

In diesem Abschnitt schauen wir uns wichtige Begriffe und Prozesse an für den Fall, dass du für ein fremdes Projekt in die Rolle des Contributors schlüpfen möchtest. Falls du direkt hier einsteigst, weil du andere Projekte unterstützen, aber kein eigenes aufbauen möchtest, empfehle ich dir, vorher einen Blick in Kapitel 4 zu werfen – es sei denn, Begriffe wie Branch und Pull-Request sind dir bereits vertraut.

## Wer bin ich und, wenn ja, wie viele?<sup>32</sup>

Wie finde ich fremde Projekte, an denen ich mich beteiligen kann? Die Beantwortung dieser Frage hängt maßgeblich davon ab, was du willst und was du kannst. Eine C#-Entwicklerin, die ihre Fähigkeiten erweitern will, sucht nach anderen Projekten als der Open-Source-Enthusiast, der gar nicht programmieren kann.

Es ist also erst einmal wichtig, sich die eigenen Kenntnisse, Ressourcen und Motivationen klarzumachen.

- **Was suche ich?** Ein spezielles Projekt, eine bestimmte Programmiersprache oder überhaupt irgendwas zum Mitmachen?
- **Was kann ich?** C# programmieren, zeichnen, gute Texte schreiben oder fließend Polnisch?<sup>33</sup>
- **Wie viel Zeit will ich aufwenden?** Nur mal für eine Stunde reinschnuppern oder ganze Wochen und Monate investieren?
- **Wie langfristig möchte ich bei einem Projekt unterstützen?** Lieber ein einziges Projekt intensiv voranbringen oder gern bei vielen kleineren unterstützen?
- **Wie viel Zeit brauche ich, um etwas beizutragen?** Bist du Anfängerin und brauchst für jede Zeile Code zehnmal so lang, oder atmest du die Issues so weg?
- **Wie wichtig ist mir schnelles Feedback?** Kannst du damit leben, dass du erst nach Wochen (oder gar Monaten) Rückmeldung auf deinen Beitrag bekommst, oder sollte das optimalerweise sofort passieren?

Je nachdem, wie deine Antworten ausfallen, werden andere Projekte infrage kommen. Es gibt einige Suchwerkzeuge, um diese Projekte zu finden.

## Fremdes Projekt suchen

Um ein fremdes Projekt zu suchen, gibt es mehrere Möglichkeiten. Da ich nicht weiß, was dir persönlich wichtig ist und was du genau suchst, werde ich die folgenden verschiedenen Möglichkeiten beschreiben:

---

<sup>32</sup> Wer sich mit weiteren philosophischen Grundsatzfragen des Lebens beschäftigen möchte, dem sei Richard David Prechts gleichnamiges Sachbuch empfohlen.

<sup>33</sup> Das sind alles Fertigkeiten, mit denen du auf GitHub gut unterstützen kannst.

- Suche nach konkretem Thema – Suche und Topics
- Suche über Stöbern – Explore und Trending
- Suche über Labels
- Suche in einem konkreten Projekt – good first issues
- Suche über andere Anbieter
- Suche über direkten Kontakt mit Maintainerinnen



#### Tipp: Selbst genutzte Software

Vielleicht nutzt du aktuell selbst eine App, bei der du gerne unterstützen würdest? Schau in der App unter den Einstellungen oder im Hilfemenü nach Menüpunkten wie *Contributing*, *Fehler melden* oder *Über diese App*. Die Wahrscheinlichkeit ist recht hoch, dass du auf der GitHub-Seite der App landest.

### Suche nach konkretem Thema – Suche und Topics

Du suchst nach einem konkreten Thema oder einer konkreten Programmiersprache? Erinnerst du dich an die Topics aus Abschnitt »Besser gefunden werden auf GitHub: Topics« auf Seite 102? Dort haben wir sie eingesetzt, damit dein Projekt besser gefunden werden kann. Was in die eine Richtung geht, geht aber auch in die andere.

A screenshot of the GitHub Topics page. At the top, there is a navigation bar with links: Explore, Topics (which is underlined), Trending, Collections, Events, and GitHub Sponsors. To the right of the navigation bar is a blue button labeled "Get email updates". Below the navigation bar, the word "Topics" is centered in a large, bold font. Underneath it, the text "Browse popular topics on GitHub." is displayed. The main content area shows three cards, each representing a popular topic:

- Sass**: Shows the Sass logo (a pink "S" with a tail) and the text "Sass is a stable extension to classic CSS."
- Ratchet**: Shows the Ratchet logo (a purple and orange gear-like icon) and the text "Ratchet is a set of libraries to handle WebSockets asynchronously in PHP."
- Mastodon**: Shows the Mastodon logo (a blue lowercase "m" inside a circle) and the text "Mastodon is a free, decentralized, open-source microblogging network."

Abbildung 6-13: Auf der Topics-Startseite sind alle Topics aufgeführt – das Durchsuchen ist aber recht mühsam.

Auf der Startseite der Topics<sup>34</sup> werden populäre Topics angezeigt (siehe Abbildung 6-13.) Wenn du weiter nach unten scrollst, siehst du noch mehr davon.

34 <https://github.com/topics>

Leider gibt es auf dieser Seite keine Suchfunktion, um die Topics sinnvoll zu durchsuchen. Eine Suche lässt sich jedoch leicht anders durchführen. Tippe das gesuchte Topic entweder in der Suchfunktion im GitHub-Menü oben auf der Seite ein oder nutze die dedizierte Suche<sup>35</sup> (z.B. für die Suche nach der Programmiersprache java). Drücke dann auf Enter. Der gesuchte Text kann allerdings überall im Repository auftauchen, z.B. in der *README.md* oder im Titel des Repos. Um jetzt dediziert nach dem Topic zu suchen, kannst du die Suchfunktion darauf eingrenzen, siehe Abbildung 6-14.

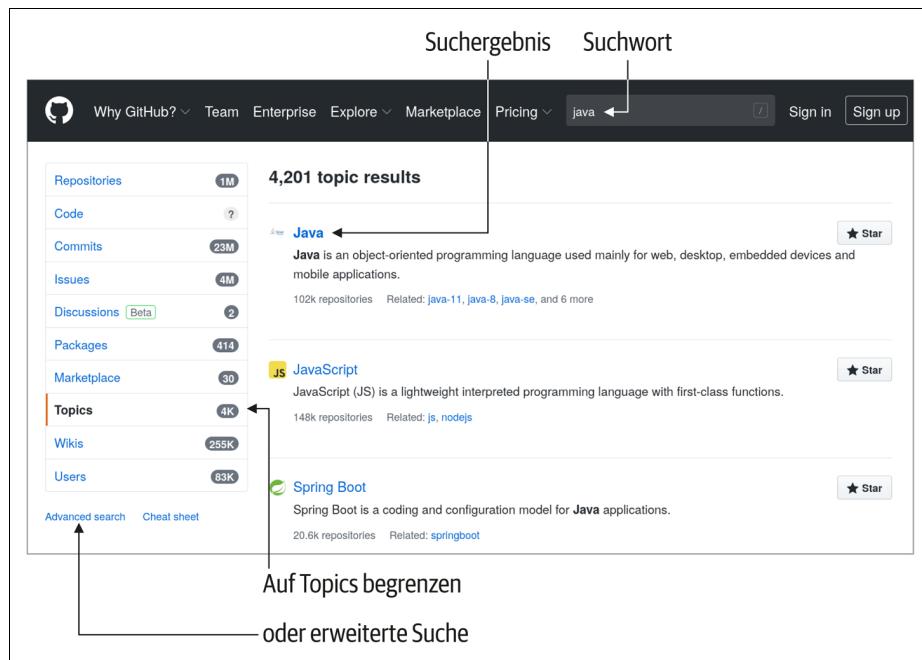


Abbildung 6-14: Jede Suche kann auf verschiedene Ebenen eingegrenzt werden, hier auf Topics.

Über *Advanced search* hast du noch weitere Suchmöglichkeiten, um beispielsweise Repositories mit einer bestimmten Lizenz, mit den noch offenen Issues insgesamt oder einer bestimmten Programmiersprache auszuwählen (oder auch alles zusammen). So erwischst du alle Projekte mit dieser Sprache, auch die, die kein entsprechendes Topic gewählt haben.

## Suche über Stöbern – Explore und Trending

Vielleicht hast du aber noch kein für dich interessantes Topic identifiziert und möchtest dich bei deiner Suche erst einmal etwas treiben lassen? Dann ist ein guter

35 <https://github.com/search>

Startpunkt *Explore*<sup>36</sup> (siehe Abbildung 6-15). Dort werden dir – basierend auf deinen bisherigen Vorlieben – diverse Repositories vorgestellt.

The screenshot shows the GitHub Explore page. On the left, there's a 'Join GitHub' sidebar with a 'Sign up for free' button and a cartoon character. The main content area has a header 'Here's what's popular on GitHub today...'. It lists two trending repositories: 'natewong1313 / bird-bot' (Python, 206 stars) and 'fastai / fastbook' (Python, 4.6k stars). Below these are sections for 'Trending repository' and 'Draft of the fastai book'. To the right, there's a sidebar titled 'Trending developers' featuring profiles for Hang Zhang, Hopsoft, Kamil Kisielka, and Felix Angelov, each with their GitHub icons and names. A 'See more Trending developers >' link is at the bottom.

Abbildung 6-15: *Explore* dient dem Stöbern.

The screenshot shows the GitHub Trending page. At the top, there's a header 'Trending' and a sub-header 'See what the GitHub community is most excited about today.' Below this is a search bar with tabs for 'Repositories' (selected) and 'Developers'. There are filters for 'Spoken Language: Any', 'Language: Any', and 'Date range: Today'. The main content area lists two repositories: 'natewong1313 / bird-bot' (Python, 206 stars, 48 issues, built by Docker, 76 stars today) and 'fastai / fastbook' (Jupyter Notebook, 4,616 stars, 866 issues, built by Docker, 56 stars today).

Abbildung 6-16: *Trending* zeigt die gerade angesagten Repositories und Entwickler\*innen.

Alternativ bietet sich *Trending*<sup>37</sup> an (siehe Abbildung 6-16). Hier werden alle Repos und Entwickler\*innen angezeigt, die aktuell »trenden« – also gerade in sind. Diese Ansicht hat den Vorteil, dass du darin noch weitere Filtermöglichkeiten hast.

36 <https://github.com/explore>

37 <https://github.com/trending>

## Suche über Labels

Erinnerst du dich an die Labels, die ich im Abschnitt »Einen Issue anlegen« auf Seite 32 in Kapitel 3 eingeführt habe? Hier können sie für die Suche genutzt werden. Zwei der von GitHub bereits von Haus aus bereitgestellten Labels kennen wir schon, nämlich *good first issue* und *help wanted*. Über die erweiterte Suche, die du in Abbildung 6-14 bereits kennengelernt hast, kannst du alle Issues finden, die mit dem einen oder dem anderen (oder beiden) Label versehen sind (siehe Abbildung 6-17).

The screenshot shows the GitHub search interface with the title 'Issues options'. It includes several search fields:

- 'In the state': A dropdown menu set to 'open/closed'.
- 'With this many comments': A text input field containing '0..100, >442'.
- 'With the labels': A text input field containing 'good first issue'.
- 'Opened by the author': A text input field containing 'hubot, octocat'.
- 'Mentioning the users': A text input field containing 'tptope, mattt'.
- 'Assigned to the users': A text input field containing 'twp, jim'.
- 'Updated before the date': A text input field containing '<YYYY-MM-DD'.

Abbildung 6-17: Eine gute Möglichkeit, Projekte zum Unterstützen zu finden, bietet die erweiterte Suche über Labels.

Viele Projekte haben eigene Labels entwickelt, die in eine ähnliche Richtung zielen, beispielsweise *easy*, *first-timers-only* oder *contribution*.<sup>38</sup> Nach diesen Labels zu suchen, ist jedoch schwieriger, da sie nicht einheitlich und eventuell nur projektspezifisch ausgeprägt sind.

Ein weiterer Knackpunkt, der bei allen bisherigen Labels auftreten kann, ist, dass die Einschätzung, ob etwas *easy* oder ein *good first issue* ist, von dem Projekteigner oder der Maintainerin vorgenommen wurde. Auch wenn etwas als einfach eingestuft wird und vielleicht nur eine Zeile Code geändert werden muss, ist es unter Umständen ausgesprochen aufwendig, sich erst einmal in das Projekt hineinzudenken. Das kann auch schnell zu Frustration führen.

Ein etwas anderer Einstieg besteht darin, gezielt nach Projekten Ausschau zu halten, die Unterstützung bei der Dokumentation oder bei der Übersetzung benötigen. Für die Suche danach kann das hauseigene Label *documentation* gute Dienste leisten. Aber auch eigene Labels wie *pending documentation* oder *translation* kann man wieder ausprobieren.

<sup>38</sup> Manchmal kann auch *question* zum Ziel führen.

Dokumentationen und Übersetzungen sind vielleicht keine Disziplinen, für die man mit Lorbeeren überschüttet wird, sie bieten aber einen wunderbaren Einstieg, da sie relativ leicht und schnell umzusetzen sind und man in der Regel nicht übermäßig viel Konkurrenz hat.

### **Suche in einem konkreten Projekt – good first issues**

Falls du schon konkret weißt, in welchem Projekt du unterstützen möchtest, und nach entsprechenden Issues suchst, bietet GitHub eine Funktion namens *good first issues* an. Du kannst diese Funktion aufrufen, indem du die Website des entsprechenden Projekts nach dem Schema *github.com/USERNAME/REPONAME/contribute* im Browser öffnest, z.B. <https://github.com/moby/moby/contribute>.

Der Name klingt nicht ganz zufällig wie das bereits im vorherigen Abschnitt erwähnte Label. GitHub nutzt für die Bewertung, ob etwas ein guter erster Issue ist, genau dieses und andere Labels. Es werden aber ebenfalls Verfahren des maschinellen Lernens eingesetzt, um auch die guten ersten Issues zu finden, die nicht gut oder ausreichend etikettiert sind.<sup>39</sup>

### **Suche über andere Anbieter**

Es gibt mittlerweile eine Reihe von Websites, die Neulinge (und alte Hasen) bei der Suche nach Projekten unterstützen – nicht nur für Programmierer!

- <https://24pullrequests.com/> ist eine Art Adventskalender und wird auch nur um die Weihnachtszeit aktiviert. Die Idee dahinter ist, dass Entwicklerinnen jeden Tag im Dezember Code verschenken (sprich: Code zu Open-Source-Projekten beisteuern).
- <https://www.codetriage.com/> ist ein Dienst, der zum einen Issues von Open-Source-Projekten auflistet und nach Programmiersprachen filterbar macht. Zum anderen können sich Interessierte dort anmelden und bekommen regelmäßig Issues zur Bearbeitung zugesandt.
- <https://gauger.io/contrib/> ist eine Übersicht über anfängerfreundliche Issues. Du kannst die Programmiersprache auswählen und bekommst eine Übersicht aller Issues pro Projekt, die ein Label haben, das auf Anfänger hindeutet (beispielsweise *good first issue*).
- <https://github.com/szabgab/awesome-for-non-programmers> ist ein Repository, das Unterstützungsmöglichkeiten für Menschen sammelt, die nicht programmieren können oder wollen.
- <https://goodfirstissue.dev/> kuratiert eine Liste von Issues, die als Einstieg gut geeignet sind. Projekte lassen sich nach Programmiersprache sortieren.

---

<sup>39</sup> Wer das Verfahren genauer nachlesen möchte: <https://github.blog/2020-01-22-how-we-built-good-first-issues/>.

- <http://issuehub.io/> sammelt Issues, die über Labels und Programmiersprache auswählbar sind.
- <https://up-for-grabs.net/> hat ebenfalls eine Liste von kuratierten Aufgaben, insbesondere für neue Conributoren. Projekte lassen sich unter anderem nach Namen und Labels durchsuchen.

## **Suche über direkten Kontakt mit Maintainerinnen**

Eine weitere gute Möglichkeit, ein geeignetes Projekt zu finden, besteht darin, auf Veranstaltungen zu fahren, auf denen (traditionell) viele Maintainer\*innen unterwegs sind. Entweder halten solche Personen Vorträge auf der Veranstaltung, und/oder man kann sie direkt ansprechen. Du bekommst dann ein erstes Gefühl für das Projekt und das Miteinander. Im direkten Gespräch kannst du zudem Fragen stellen, die dir auf der Seele brennen und die du eventuell nicht per Mail gestellt hättest.

Meiner Erfahrung nach suchen viele Maintainerinnen nach Unterstützern und sind froh über jede Person, die Lust hat, mitzumachen. Und wenn du der Maintainerin persönlich bekannt bist, kann das nur von Vorteil sein.

Und wo finde ich solche Veranstaltungen? Meine kleine – absolut nicht vollständige – Liste in alphabetischer Reihenfolge:

- **Chaos Communication Congress**, eine derzeit in Leipzig stattfindende Konferenz des *Chaos Computer Club* (CCC) über Technologie, Gesellschaft und Utopie (<https://events.ccc.de>). Über das Jahr verteilt, finden auch kleinere Veranstaltungen vom CCC statt.
- **Chemnitzer Linux-Tage** (CLT), eine in Chemnitz (wer hätte das gedacht?) stattfindende Veranstaltung für kleines Geld zum Thema Linux und Open Source (<https://chemnitzer.linux-tage.de>).
- **FOSDEM** (*Free and Open Source Software Developers' European Meeting*), eine in Brüssel stattfindende kostenlose Veranstaltung mit dem Ziel, Entwicklerinnen und Entwicklern von freier und Open-Source-Software sowie Communitys einen Treffpunkt zu bieten (<https://fosdem.org>).
- **FrOSCon** (*Free and Open Source Software Conference*), eine jährlich in Sankt Augustin (nahe Köln) stattfindende kostenlose Konferenz zum Thema Open Source (<https://www.froscon.de>).
- **Maker-Faire**, eine in vielen größeren Städten angebotene Do-it-yourself-Messe (<https://maker-faire.de>).
- **Open Source Summit Europe** (OSS EU), eine jährliche Veranstaltung für die Open-Source-Gemeinschaft, die jedes Jahr in einer anderen europäischen Stadt stattfindet (<https://events.linuxfoundation.org/open-source-summit-europe>).

Eine weitere Möglichkeit für direkten Kontakt ist das Aufsuchen von sogenannten Hackspaces oder FabLabs. Das sind in der Regel von einem Verein getragene Orte, an denen Menschen mit Interesse an Technik zusammenkommen und gemeinsam Projekte vorantreiben. Hackspaces gehen tendenziell eher in Richtung Arbeiten

mit Software/Hardware, während FabLabs (oder auch Makerspaces genannt) häufig einen größeren Maschinenpark mit Bandsäge, Lasercutter oder CNC-Fräse besitzen. Die Übergänge sind hier aber fließend.

Meine Empfehlung ist, eine Internetsuchmaschine deiner Wahl mit den Begriffen »Hackspace« und/oder »FabLab« und eventuell deinem gewünschten Ort zu füttern. Vielleicht helfen auch Suchwörter wie »IT-Stammtisch«, »IT-Meetup« und/oder »Linux User Groups«. In jeder größeren Stadt solltest du dabei fündig werden. Auf der Website des Hackspace findest du deren Kontaktdaten und kannst im Vorfeld anfragen, ob es dort jemanden gibt, der Unterstützung bei einem Projekt benötigt.

## Fremdes Projekt begutachten

Wenn du ein interessantes Projekt gefunden hast, ist es eine sehr gute Idee, zunächst zu überprüfen, ob es noch lebt. Nichts ist frustrierender, als viel Liebe und Energie in die Lösung eines Issues gelegt zu haben, und dann ist niemand da, um den Pull-Request zu mergen.

### Pulse – Lebt das Projekt noch?

Eine erste gute Anlaufstelle, um herauszufinden, wie aktiv ein Projekt betreut wird, ist der sogenannte *Pulse*, der bei jedem gefundenen Projekt unter *Insights* versteckt ist. Hier kann man sehen, wie viele Pull-Requests und Issues in einer gewählten Zeitperiode neu erstellt bzw. geschlossen/gemerget wurden (siehe Abbildung 6-18). Dabei wird nicht die Gesamtanzahl aller Pull-Requests und Issues dargestellt, sondern wirklich nur die, die innerhalb dieser Zeitperiode erstellt bzw. geschlossen/gemerget wurden.

Welche Bedeutung hat das für dich? Wenn du eine Person bist, die schnell Veränderungen reinbringen und ebenso schnell Rückmeldung zu ihren Beiträgen erhalten möchte, könnte ein Projekt mit einem niedrigen *Pulse* deine Nerven strapazieren. Wenn du jemand bist, der gern über einen längeren Zeitraum und mit viel Ruhe eine Veränderung erarbeitet, könnte dich ein Projekt mit hohem *Pulse* ebenso nerven, da du dadurch gegebenenfalls gezwungen wirst, regelmäßig gewisse »Wartungsarbeiten« vorzunehmen (das nennt man auch »den eigenen Fork aktuell halten«, siehe hierzu den Abschnitt »Ein geforktes Projekt aktuell halten« auf Seite 133 weiter unten).

Ein Projekt mit einem kaum spürbaren *Pulse* ist eventuell tot, und es will gut überlegt sein, ob die eigene Zeit und Kraft anderswo nicht sinnvoller investiert ist. Andererseits könnte das auch der Anfang eines neuen Aufsatzes für das Projekt werden mit dir als Projekteignerin. Ich habe es aber auch schon erlebt, dass vermeintlich tote Projekte trotzdem relativ zügig meine Pull-Requests akzeptiert haben. Der *Pulse* gibt dir also einen ersten Ansatz, das entsprechende Projekt für dich zu bewerten, ist aber kein Allheilmittel.

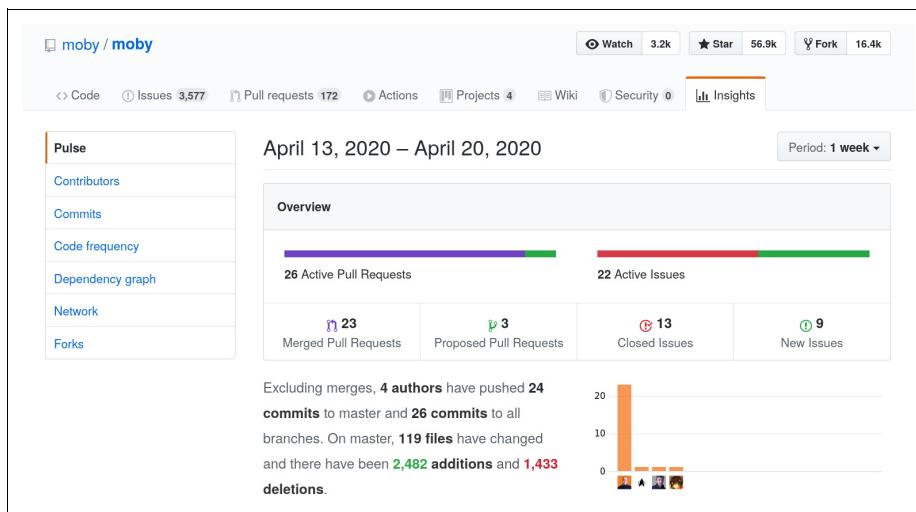


Abbildung 6-18: Der Pulse gibt Aufschluss darüber, wie aktiv ein Projekt derzeit ist.



#### Tipp: Projektlebenszeichen über letztes Änderungsdatum

Für eine schnelle erste Begutachtung kann man in der Dateiliste eines Projekts feststellen, wann die letzte Änderung stattgefunden hat. Das gibt meist schon ein erstes Indiz über die Lebendigkeit des Projekts.

### Projekt über Health Files bewerten

Im Abschnitt »Dein Projekt anschaulich beschreiben« auf Seite 104 haben wir die *Community Health Files* wie die Datei *CONTRIBUTING.md* bereits kennengelernt. Jetzt ist es an der Zeit, sie für die Bewertung eines Projekts zu nutzen. Wir schauen uns noch mal die relevanten Dateien an. Bei jeder Datei solltest du dich fragen:

- Ist die Datei im Projekt vorhanden? Was sagt das über das Projekt aus, wenn sie fehlt bzw. wenn sie da ist?
- Was steht drin? Gefällt mir, was ich lese?

Die relevanten Dateien sind:

1. *README.md* – hier sollte dein erster Blick hingehen. Viele Repos nutzen nicht die *CONTRIBUTING.md*, um potenziellen Conributoren Hilfestellung zu geben, sondern die *README.md*.
2. *CONTRIBUTING.md* – alleine das Vorhandensein einer solchen Datei drückt schon aus, dass sich der Projekteigner mit dem Thema befasst hat und auch Unterstützung wünscht.
3. *Code of Conduct* – falls es einen Verhaltenskodex gibt, solltest du dich damit identifizieren können.

4. *License* – gibt es keine Lizenz, kannst du rein rechtlich das Projekt nicht unterstützen (wir erinnern uns an Kapitel 5). Falls es eine Lizenz gibt, musst du natürlich entscheiden, wie wichtig oder hinderlich für dich beispielsweise ein starkes Copyleft ist.
5. *Issue- und Pull-Request-Templates* – gibt es Vorlagen, könnte das ein Indiz für ein strukturiertes Projekt sein. Je nachdem, was für ein Typ du bist, kann das entweder toll oder abschreckend sein.

## Projekt über Issues und Pull-Requests bewerten

Ein weiterer wichtiger Bewertungsindikator sind die Issues und Pull-Requests eines Projekts. Hier gibt es eine ganze Reihe an Fragen, die zu stellen sich lohnt:

- Gibt es viele offene Issues und Pull-Requests? Und wenn ja, wie alt sind sie? Wann war die letzte Aktivität?
- Gibt es (relativ) schnelle Antworten der Maintainer auf neue Issues oder Pull-Requests?
- Wird in Issues und Pull-Requests diskutiert? Oder werden sie kommentarlos geschlossen?
- Bedanken sich Maintainerinnen für die Unterstützung?
- Wie ist generell der Umgangston bei Diskussionen?



### Wer ist Maintainer\*in eines Projekts?

Die Frage ist leider nicht einfach zu beantworten. Die einzige Möglichkeit, die ich bisher gefunden habe, ist, sich die Issues und Pull-Requests anzuschauen. Wer beispielsweise einen Pull-Request mergen kann, ist Maintainerin (siehe auch Abbildung 6-19).



Abbildung 6-19: Dieser User kann in dem Projekt mergen, er ist Maintainer (oder Projektleiter).

## Fremdes Projekt unterstützen – Fork

Bevor du auf dem fremden Projekt deiner Wahl loslegst und die ersten Änderungen durchführst, möchte ich dir eine Empfehlung mitgeben. Ich habe irgendwo mal folgendes Bild gelesen: Wenn man zum ersten Mal bei einem Open-Source-Projekt unterstützen möchte, ist das wie auf eine Party zu gehen, auf der man niemanden kennt. Um nicht unangenehm aufzufallen, sollte man daher erst einmal den Raum, die Gäste und deren Gesprächsthemen beobachten, bevor man selber aktiv wird.

Diesen Tipp kann ich dir so uneingeschränkt weitergeben. Es ist ein guter Einstieg, zunächst die Health Files zu lesen<sup>40</sup> und danach zu schauen, wie in Issues und Pull-Requests kommuniziert wird. Hat das Projekt einen alternativen Kommunikationskanal (z.B. ein Chatprogramm), klinke dich dort mit ein und lies erst einmal nur mit. Schon bald hat man ein gutes Gefühl dafür, wie das Projekt tickt.

Solltest du etwas in einem Projekt entdeckt haben, bei dem du gern Änderungen einbringen würdest, ist es häufig eine gute Idee, im Vorfeld zu klären, ob du dich der Sache annehmen darfst. Je nach Thema und Komplexität ist es dabei durchaus sinnvoll, kurz zu skizzieren, wie deine Änderung aussehen könnte. So können potenzielle Missverständnisse oder Vorstellungen von Anfang an geklärt werden, bevor du eventuell Code schreibst, der am Schluss nicht übernommen wird.

## Editieren in einem fremden Projekt – Fork

Selbst wenn du ein passendes Projekt gefunden und dich eingelesen hast, gibt es noch ein Problem: Du bist weder Projekteigner noch Maintainerin und hast kaum Rechte auf dem Projekt. Du kannst beispielsweise Issues anlegen, aber sobald du eine Datei editieren möchtest, wirst du auf die fehlende Schreibberechtigung hingewiesen (siehe Abbildung 6-20).

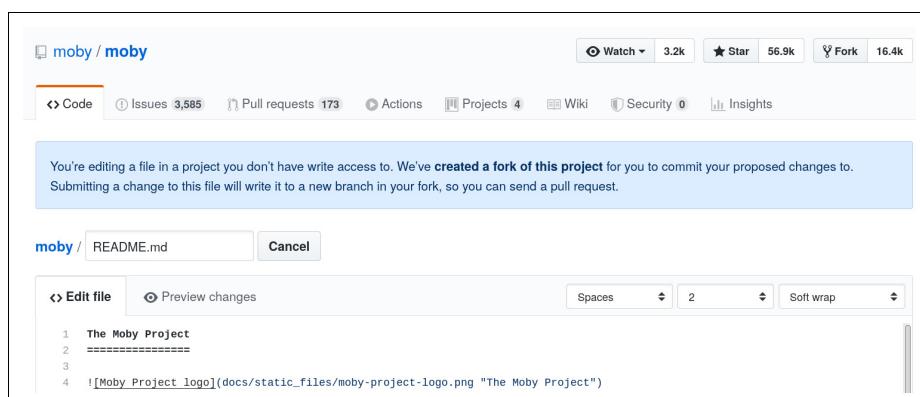


Abbildung 6-20: Editieren ohne Schreibrechte erzeugt automatisch einen Fork.

Sobald du ein fremdes Projekt editieren willst, erzeugt GitHub automatisch einen sogenannten Fork (deutsch »Gabelung«) für dich. Das ist eine Eins-zu-eins-Kopie des fremden Projekts in deinen Account hinein. Kopiert werden alle Dateien unter *Code*, aber nicht die Issues oder Pull-Requests. Der Fork ist wie ein neues, frisches Repository mit den Dateien des geforkten Repos. Wenn du jetzt auf die Übersicht deiner Repositories gehst (z.B. über *Your repositories* im GitHub-Menüband), sollte ein neues Repository aufgetaucht sein (in meinem Fall das Projekt *moby* (siehe Abbildung 6-21), auch wenn du das Projekt nicht editiert oder gespeichert hast.

40 Hast du während deiner Suche vermutlich eh schon gemacht.

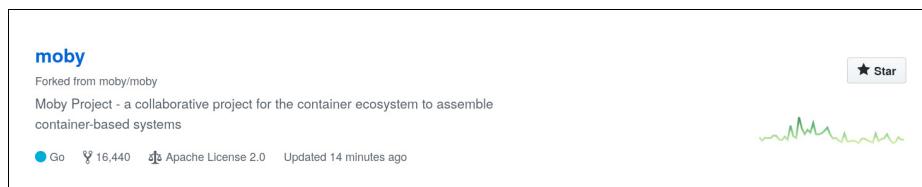


Abbildung 6-21: Nach einem Fork sagt uns GitHub, von welchem Repository wir geforkt haben.

In diesem Repository kannst du jetzt schalten und walten, wie du es für richtig hältst. Üblicherweise forkt man ein anderes Projekt nicht, indem man eine Datei editiert, sondern über den Button *Fork* im fremden Repository (siehe Abbildung 6-22). Die Nummer neben dem Button ist übrigens die Gesamtanzahl der Forks, die Menschen von diesem Projekt gemacht haben – ein Hinweis darauf, wie viele (oder wenige) Personen bei diesem Projekt mitwirken. Durch ein Forken erhöhest du den Zähler um eins. Die anderen beiden Buttons *Watch* (»Beobachten«) und *Star* (»Stern«) dienen als Lesezeichen für ein Repository. Durch Anklicken von *Star* kannst du zusätzlich noch deine Wertschätzung gegenüber dem Repository kundtun (ähnlich wie ein *Like*-Button in einigen Social-Media-Anwendungen).

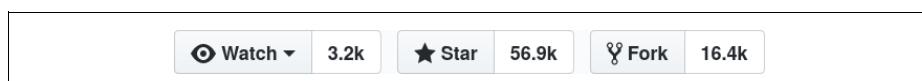


Abbildung 6-22: Forken geht mit einem Klick auf den Button Fork.

## Fork und Pull-Requests

Was ist ein Fork, und wofür brauche ich das? Stell dir ein fremdes Projekt auf GitHub wie ein fremdes Haus mit offenen Türen vor. Du kannst in jedes Zimmer gehen (jeden Dateiordner öffnen) und jede Schublade aufmachen (jede Datei anschauen). Du darfst aber nicht deine eigenen Socken in die Schublade packen oder Änderungen am Mobiliar vornehmen.

Beim Besichtigen des Hauses entdeckst du, dass ein paar der Vorhänge zerschlissen sind (Fehler sind vorhanden) und nach deinem Geschmack ein hübsches Bild über den Kamin gehört (eine Funktion/ein Feature fehlt). Zufällig hast du selber noch ein paar schöne Vorhänge und ein hübsches Bild übrig, die genau passen würden. Du würdest sie der Hausbesitzerin gerne schenken. Wie machst du das?

GitHub erlaubt es nicht, dass du bzw. ein x-beliebiger dahergelaufener Fremder Änderungen direkt am fremden Haus durchführen kann. Das fände die Hausbesitzerin vermutlich auch nicht so lustig. GitHub erlaubt es aber, dass du das komplette Haus (inklusive Rattenloch im Keller) kopieren und eins zu eins auf deinem Grund und Boden (innerhalb deines Accounts) wieder aufbaust. Was bei (echten) Häusern Wochen bis Monate braucht, dauert bei unserem metaphorischen Haus nur Sekunden (maximal Minuten). Das nennt man dann »einen Fork machen«. Man sagt auch gerne »ich forke das Projekt«, um dasselbe auszudrücken. Vielleicht hast du schon mal irgendwo gelesen, dass »Software X ein Fork von Software Z« ist.

Damit ist genau das gemeint, was ich eben beschrieben habe. Irgendjemand hat Z geforkt und ein eigenes, selbstständiges Projekt daraus gemacht. Das passiert meistens, wenn Projekt Z nicht instand gehalten wird (das Projekt also tot ist und keine neuen Funktionen oder Fehlerbehebungen mehr durchgeführt werden) oder aber wenn der Projekteigner von Z strategisch in eine andere Richtung möchte als die Person, die Projekt X geforkt hat.

Okay, jetzt hast du also das Haus auf eigenem Grund und Boden aufgebaut. Und nun? Auf deinem Grund (auf Projekten innerhalb deines Accounts) hast du die völlige Freiheit, zu gestalten, was du willst. Das bedeutet, in diesem »geforkten« Haus kannst du jetzt die Vorhänge austauschen und das Bild über den Kamin hängen.<sup>41</sup> Sobald du mit allem fertig bist, musst du der Projekteignerin des ursprünglichen Projekts irgendwie noch mitteilen, dass du da was cooles Neues geschaffen hast und sie das unbedingt bei sich im Haus auch machen sollte.<sup>42</sup> Das machst du mit einem Pull-Request. Wenn wir in unserem Bild mit dem Haus bleiben: Du schnürst ein Paket mit all deinen Änderungen zusammen (den neuen Vorhängen, dem Bild für den Kamin plus eine Anleitung, wo was anzubringen ist), stellst es der Projekteignerin vor die Haustür und klingelst. Jetzt können mehrere Dinge passieren: Hat sie kein Interesse mehr an ihrem Haus und wohnt vielleicht auch schon ganz woanders, kann es sein, dass dein Paket vor der Haustür vergammelt. Das ist häufig der Zeitpunkt, an dem Forks von Projekten entstehen. Im besten Fall öffnet sie die Tür, schaut sich das Paket genau an und nimmt es mit ins Haus, um die Vorhänge und das Kaminbild anzubringen. Hier wird, glaube ich, deutlich, warum Pull-Requests so heißen, wie sie heißen. Das Paket, das du vor die Tür stellst, ist eine Aufforderung (engl. *Request*), doch bitte die tollen Änderungen ins Haus zu »ziehen« (engl. *pull*). Du kannst deine Änderungen nicht ins Haus »drücken« (engl. *push*), da du ja keine Schreibberechtigung auf dem fremden Projekt hast.

Was ebenfalls passieren kann, ist, dass die Projekteignerin zwar noch in dem Haus wohnt, aber den Inhalt deines Pakets nicht so dolle findet (»Grün als Vorhangsfarbe? Ich bitte dich!«). Dann lehnt sie das Paket ab – in »GitHub-Sprech«: Sie schließt den Pull-Request.

## Loslegen 2 – anders forken

Es ist leichter, den folgenden Ausführungen zu folgen, wenn du ein bestimmtes Repository forkst, und zwar *first-contributions*<sup>43</sup> (deutsch »erste Unterstützung«). Dieses Repo ist – wie der Name schon sagt – dafür da, deine erste Unterstützung bei einem fremden Projekt auszuprobieren. Wenn du es vermeiden kannst, lies dir die *README.md* nicht durch. Sie beschreibt das Vorgehen mit Git. Wir wollen aber zunächst noch ohne Git arbeiten.

<sup>41</sup> Oder auch Wände einreißen und eine weitere Etage aufstocken.

<sup>42</sup> Du kannst natürlich auch einfach das geforkte Projekt als eigenes weiterentwickeln. Hier gehe ich davon aus, dass du erst einmal zu einem bestehenden Projekt etwas beisteuern möchtest.

<sup>43</sup> <https://github.com/firstcontributions/first-contributions>

Das Repository bietet einen sehr niedrigschwelligen Einstieg an. Die Aufgabe ist es, lediglich deinen Namen in eine bestimmte Datei zu schreiben und dann mithilfe eines Pull-Requests diese Änderung in das Originalprojekt zu übertragen.

Gehe auf die Seite des Repositorys und wähle dort den Button *Fork* aus (siehe Abbildung 6-22). GitHub braucht jetzt ein paar Sekunden, um den Fork durchzuführen (siehe Abbildung 6-23) und leitet dich danach zu dem gerade geforkten Repository innerhalb deines Accounts weiter. Unter dem Repository-Namen siehst du, ob und woher das aktuelle Repository geforkt wurde (siehe Abbildung 6-24). Dein geforktes Projekt ist also weiterhin mit dem Originalprojekt verknüpft.<sup>44</sup>

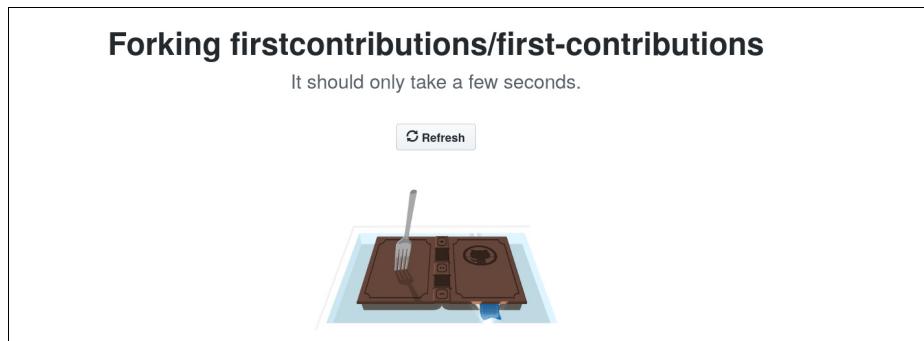


Abbildung 6-23: Der Vorgang für das Forken dauert in der Regel wenige Sekunden.



Abbildung 6-24: Unter dem Repository-Namen erkennt man direkt, ob etwas ein Fork ist und, wenn ja, von was.

Editiere in deinem Fork die Datei *Contributors.md*, indem du deinen eigenen Namen irgendwo in die lange Liste der Conributoren einträgst, z.B. wie folgt:

- [NAME] (<https://github.com/ACCOUNTNAME>)

Damit erzeugst du einen neuen Aufzählungspunkt mit dem Namen NAME und einem dahinterliegenden Link, der auf deinen Account verweist. Du kannst da natürlich auch eine andere Webadresse eintragen. Nach dem Abspeichern (Committen) hast du auf deinem geforkten Repository eine entsprechend angepasste *Contributors.md*, das Originalprojekt kennt diese Anpassungen aber noch nicht. Das ändern wir mit einem Pull-Request. Im Abschnitt »Pull-Request – Änderungen in Branches aufzeigen« auf Seite 53 in Kapitel 4 wurden Pull-Requests nur für Branches genutzt. Sie sind aber auch dafür da, um dem Originalprojekt Änderungen in

<sup>44</sup> Andersherum übrigens auch: Wenn du im Originalprojekt auf die Fork-Nummer klickst, werden alle seine Forks aufgelistet.

geforkten Projekten aufzuzeigen. Erstelle einen Pull-Request, indem du in deinem geforkten Repo im Register *Pull requests* auf den Button *New pull request* klickst (siehe Abbildung 6-25).



Abbildung 6-25: Das Anlegen eines Pull-Requests geht unter anderem über das Register *Pull requests*.

Du solltest auf eine Ansicht kommen ähnlich wie die in Abbildung 6-26. Pull-Requests innerhalb eines Projekts haben wir bereits in Kapitel 4 angelegt. Ich möchte hier noch auf ein paar Besonderheiten eingehen, die Pull-Requests projektübergreifend haben. Wie du vielleicht schon gesehen hast, erzeugen wir den Pull-Request nicht in deinem geforkten Repository, sondern im Originalprojekt ❶. Das ist auch sinnig, da wir ja schließlich das Originalprojekt über unsere Änderung informieren wollen.

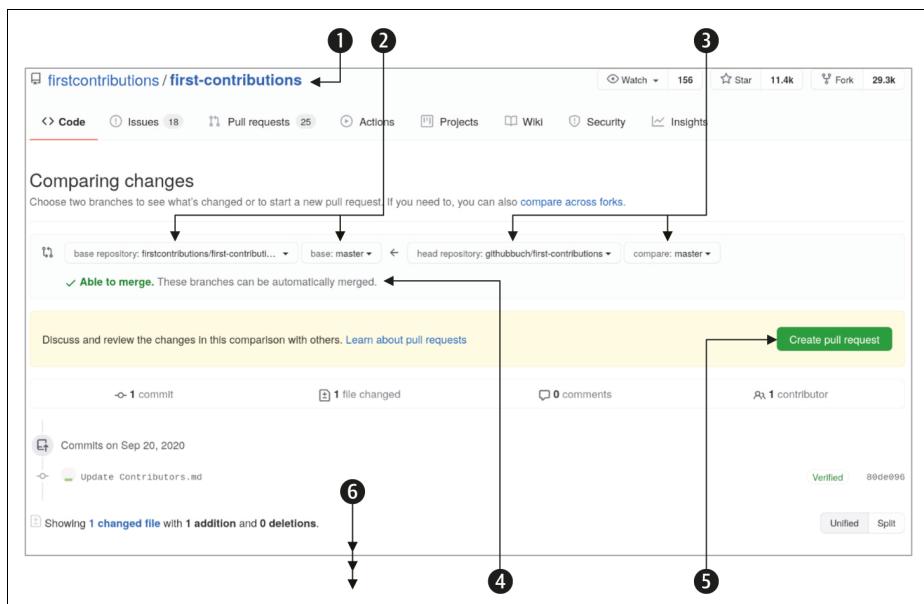


Abbildung 6-26: Ein Pull-Request wird in der Regel im Originalprojekt angelegt.

Die Festlegung, was in dem Pull-Request genau miteinander abgeglichen werden soll, beinhaltet mehr Optionen, als wir bisher kennen. Wir müssen ja nicht nur einzelne Branches innerhalb eines Projekts festlegen, sondern auch noch die jeweils betroffenen Projekte. Die Reihenfolge sollte dabei sein: Original ❷ links und geforktes Projekt ❸ rechts. Die Botschaft, dass der Pull-Request mergbar ist ❹, und den

Button, der den Pull-Request schlussendlich anlegt ❸, kennen wir schon aus früheren Pull-Requests. Wenn du noch mal sehen möchtest, welche Änderungen in dem Pull-Request sind, bekommst du diese Information durch Runterscrollen ❹.

Durch die Verknüpfung der beiden Repos sollte GitHub automatisch die richtige Einstellung vorgeschlagen haben, sodass du einfach den Button *Create pull request* anklicken kannst. Als Titel für den Pull-Request kannst du Folgendes verwenden:

Add NAME to Contributors list

Natürlich solltest du noch NAME durch deinen eigenen Namen ersetzen. Sobald du den Pull-Request erstellt hast, kannst du ihn im Originalprojekt unter dem Menüpunkt *Pull requests* anschauen. Mit großer Wahrscheinlichkeit wirst du eine Reihe an Pull-Requests sehen, aber nicht deinen eigenen. Das liegt daran, dass dein Pull-Request bereits gemerget und geschlossen wurde und wieder die Standardfilter greifen, die nur offene Pull-Requests anzeigen (siehe Abbildung 6-27 ❶). Lass dir die geschlossenen Pull-Requests anzeigen, indem du auf *Closed* (❷) klickst und damit die Filter änderst.



Abbildung 6-27: Einige Filter lassen sich durch einfaches Anklicken anpassen.

Du solltest jetzt deinen Pull-Request sehen und auch die Info, dass er gemerget wurde (wie z. B. in Abbildung 6-28 zu sehen). Normalerweise würde das Mergen in einem fremden Projekt nicht so schnell gehen, da in der Regel eine Maintainerin erst einmal auf die Änderungen schaut. Im Originalprojekt läuft aber ein Automatismus, der konfliktfreie Pull-Requests mithilfe eines Bots automatisch mergt. Zum Schluss kannst du deinen Fork löschen. Sofern du nicht vorhast, regelmäßig zu einem Projekt etwas beizusteuern, ist das Löschen generell eine gute Idee, um die Anzahl der Repositories in deinem Account überschaubar zu halten.



Abbildung 6-28: Ein erfolgreich gemergter Pull-Request

Herzlichen Glückwunsch, du hast soeben deinen ersten Beitrag zu einem fremden Projekt geleistet!

Vielleicht fragst du dich, warum es das mit dem Forken überhaupt gibt? Es würde doch genauso funktionieren, wenn die Projekteignerin dir die Schreibrechte für ihr Repository geben würde (du also Maintainerin werden würdest)? Ja, würde es.

Aber würdest du jedem »Dahergelaufenen«, den du nicht kennst, erlauben, dein Haus/deine Wohnung neu zu dekorieren, Möbel rauszuschmeißen und eigene Vorhänge aufzuhängen (siehe auch die Erklärbarbox »Fork und Pull-Requests«)? Vermutlich nicht, du würdest die Person erst einmal kennenlernen wollen und sie später vielleicht zur Maintainerin machen. Das bedeutete aber, dass es relativ lange dauern würde, bis jemand überhaupt im Open-Source-Bereich unterstützen könnte. Gerade das Konzept des Forkens ermöglicht jedoch einen schnellen Einstieg. Die Projekteignerin muss kein Vertrauen in dich haben. Wenn dein Beitrag (Pull-Request) gut genug ist, wird sie ihn mergen, und du kannst auch ohne Schreibrechte ein Projekt voranbringen.

### Ein geforktes Projekt aktuell halten

Wenn du regelmäßig zu einem anderen Projekt etwas in Form von Pull-Requests beitragen möchtest, ist es wichtig, zu verstehen, wie das mit dem Aktualisieren eines Forks funktioniert. Stell dir vor, du hättest gestern ein Projekt geforkt, um etwas daran zu ändern. Heute hat die Projekteignerin des Originalprojekts aber noch einige Änderungen vorgenommen, die gegebenenfalls auch die Dateien betreffen, die du gerade ändern möchtest. In einer solchen Situation ist Folgendes wichtig:



#### Wichtig: Aktualität von Forks

Forks bleiben nicht automatisch synchron mit dem Originalprojekt, das muss man manuell und selbst machen.

Um mit dem Originalprojekt wieder synchron zu werden, besteht der einfachste und sauberste Weg darin, das geforkte Repository komplett zu löschen (siehe Abschnitt »Ein bestehendes Repository löschen« auf Seite 39 in Kapitel 3) und neu zu forken. Damit hast du den aktuellen Stand des Originalprojekts und keine querschießenden Commits, falls du auf dem eigenen Repository bereits etwas committed hast.

Auf Dauer ist das aber nicht sonderlich praktisch, gerade wenn du an Änderungen arbeitest, die etwas mehr Zeit in Anspruch nehmen und dir der Projekteigner mit seiner (gefühlten) Hyperaktivität regelmäßig dazwischenfunkt. Auch wenn du mehr als einmal etwas zu dem Originalprojekt beisteuern möchtest, ist dieser Weg nicht sehr befriedigend. Rein über die Oberfläche von GitHub – ohne die Unterstützung von anderer Software – habe ich allerdings während der Recherchen keine gut funktionierende Variante gefunden. Es gibt zwar andere Möglichkeiten,<sup>45</sup> die sich aber (für mich) als wenig praktikabel herausgestellt haben. Die Projekteigner konnten meine Pull-Requests häufig nicht sauber oder gar nicht mergen. Ich musste dann alles noch mal neu machen, indem ich komplett neu geforkt und dann ein weiteres Mal neu editiert habe.

---

<sup>45</sup> Siehe z.B. <https://stackoverflow.com/questions/7244321/how-do-i-update-a-github-forked-repository>.

Zum Glück gibt es aber einen anderen, weniger nervenaufreibenden Weg. Dieser basiert auf dem Zusammenspiel von GitHub mit Git, und das lernst du im Abschnitt »Szenario 3: Geforktes Projekt auf GitHub lokal zu Git holen« auf Seite 179 in Kapitel 8 kennen. Im nächsten Kapitel schauen wir uns aber vorher Git (ohne GitHub) an, um uns erst einmal damit vertraut zu machen.

## KAPITEL 7

# Ein Projekt lokal mit Git verwalten

In diesem Kapitel schauen wir uns Git genauer an. Wir starten damit, dass ich dir zunächst erkläre, warum GitHub für manche Dinge nicht ausreicht, und im Anschluss, was Git eigentlich genau ist. Nachdem wir Git installiert haben, beschäftigen wir uns damit, wie du ein lokales Projekt mit Git verwaltetest. GitHub wird in diesem Kapitel noch keine große Rolle spielen. Das Zusammenspiel von Git und GitHub wird in Kapitel 8 beschrieben. Für diejenigen, die noch nie mit einer Konsole gearbeitet haben, gibt es in diesem Kapitel dazu einen kleinen Exkurs.



### Am Ende des Kapitels kannst du ...

- entscheiden, ob der Einsatz von Git für dich infrage kommt.
- mit der Konsole umgehen.
- Git installieren und einrichten.
- die Konsole auf Git-spezifische Merkmale anpassen.
- Projekte lokal mit Git verwalten.
- mit Branches in Git umgehen.
- mit großen Binärdateien in Git umgehen.

## Warum GitHub allein manchmal nicht ausreicht

Da du es bis hierhin geschafft hast, hast du schon einen guten Überblick darüber, was man mit GitHub über die Weboberfläche der Plattform alles anstellen kann. Im vorherigen Kapitel ging es beispielsweise darum, dass das Aktualisieren eines Forks über GitHub am einfachsten über das Löschen und Neuforken vonstattengeht. Solche und andere Dinge funktionieren mit Git aber viel eleganter, und deshalb arbeiten viele eben nicht nur mit GitHub, sondern nutzen ergänzend auch Git. Ein weiterer Grund: Manches geht mit GitHub alleine gar nicht, beispielsweise gibt es unter GitHub keine Möglichkeit, Verzeichnisse umzubenennen.

Wir haben zudem schon gesehen, dass alles, was auf der GitHub-Oberfläche passiert, »irgendwo im Internet« stattfindet. Wer Git nutzt, kann lokal auf seinem Rechner arbeiten, eine wacklige Internetverbindung ist dann für das Arbeiten kein

Problem. Und du kannst mit dem Editor deiner Wahl arbeiten – für viele ein ganz wichtiges Argument.

Weitere Gründe können sein, dass du ein Backup von deinem Projekt auf deiner lokalen Platte haben möchtest (nur für den Fall). Das geht zwar auch mit dem Download-Button über die Weboberfläche, ist mit Git aber komfortabler und vor allem leichter zu automatisieren.

Und zu guter Letzt ist es gerade im Programmierumfeld häufig einfacher, wenn du dir den Programmiercode auf den eigenen Rechner holst, um das Programm bzw. dessen Neuerung ausgiebig zu testen, bevor diese in den master-Branch gemerget werden. Im professionellen Bereich ist die Nutzung von Git sogar oft Voraussetzung.

Du siehst, es gibt viele gute Gründe dafür, dass die Weboberfläche von GitHub manchmal alleine nicht ausreicht und es sich lohnt, Git näher zu betrachten.

## Git, was ist das? – Eine kurze Einführung

Git ist ein Programm zur dezentralen Versionsverwaltung von Dateien. Das klingt ziemlich wichtig, aber was heißt das genau, und wofür brauche ich das? Dröseln wir das Ganze mal von hinten auf: Mit Git kann man also Versionen von Dateien verwalten, und das passiert irgendwie dezentral. Schauen wir uns diese beiden Punkte genauer an.

### Versionsverwaltung

Eine Versionsverwaltung verwaltet Versionen – so einfach und so offensichtlich. Eine Version ist ein zu einem bestimmten Zeitpunkt festgelegter »Fertigstellungsgrad« eines Dokuments oder einer Software. Beispielsweise könnte die erste Veröffentlichung einer Software die Version 1.0 sein oder der erste Zwischenstand einer Studienarbeit die Version 0.1 haben.

### Über semantische Versionierung

Es gibt verschiedene Ansätze, wie man eine Versionierung durchführen kann. Du hast vielleicht schon so etwas gesehen wie »1.7«, »1.7.3« oder »1.7.3-12« oder auch »1.7.-beta-2«. Das heißt, jede und jeder macht es ein wenig anders.

Ein Versuch, die Versionierung zu vereinheitlichen, damit auch allen klar ist, welche Bedeutung die eine oder andere Zahl in der Version hat, ist das sogenannte semantische Versionieren (*Semantic Versioning*)<sup>1</sup>. Jede Version besteht dabei aus drei Zahlen:

<sup>1</sup> <https://semver.org/> – wird auf GitHub weiterentwickelt: <https://github.com/semver/semver>.

1. **Major Version** (deutsch »Hauptversion« oder »große Version«) bedeutet große Veränderungen, die gegebenenfalls zu Inkompabilitäten an Schnittstellen führen können. Beispiel: Du nutzt eine Bibliothek der Version 1.3.0, und diese springt auf Version 2.0.0. Dann kann es sein, dass du deinen Code anpassen musst.
2. **Minor Version** (deutsch »kleine Version«) bedeutet kleinere Veränderungen aufgrund neuer Funktionalitäten, die aber rückwärtskompatibel sind. Du musstest deinen Code also nicht unbedingt anpassen.
3. **Patch Version** (deutsch »Flickenversion«) bedeutet kleinere Veränderungen durch das Beheben von Fehlern. Auch hier sind die Anpassungen rückwärtskompatibel bzw. sollten es sein.

Es gibt auch noch die Möglichkeit, Vorabveröffentlichungen kenntlich zu machen. Das geht inhaltlich hier aber zu weit.

Solltest du dich in deinem Projekt für eine semantische Versionierung entscheiden, habe ich ein Werkzeug auf GitHub gefunden,<sup>2</sup> das verspricht, diese Aufgabe für Git-Projekte zu vereinfachen. Ich habe es selber nicht ausprobiert, freue mich aber über einen Erfahrungsbericht.

Eine Versionsverwaltung bietet Werkzeuge an, mit deren Hilfe man mit verschiedenen Versionen von Dateien arbeiten kann, um beispielsweise eine alte und die aktuelle Version vergleichen zu können. Du kannst identifizieren, wer welche Änderung wann gemacht hat, und wenn die Menschen gut kommentieren, weißt du hoffentlich auch, warum. Insbesondere in der Softwareentwicklung sind dies wichtige Funktionen, um zu wissen, welche Features (und vielleicht auch Fehler) in welcher Softwareversion vorhanden sind oder waren und wer etwas eventuell verbockt (oder wieder geradegebogen) hat.



### Definition Versionsverwaltung

Eine Versionsverwaltung ist ein System, das Änderungen an Dateien über die Zeit aufzeichnet und Werkzeuge bereitstellt, um das Arbeiten mit Versionen zu erleichtern.

Oder um die Notwendigkeit der Versionsverwaltung noch etwas deutlicher zu machen: Stell dir vor, du bist zusammen mit einer Kollegin zuständig für eine Webseite. Ihr beide ändert relativ zeitgleich dieselbe Datei und ladet diese auf den Webserver hoch. Wenn deine Kollegin ihre Datei später hochlädt als du, überschreibt sie deine Datei und damit deine Änderungen. Mit einer Software zur Versionsverwaltung wäre das nicht passiert. Diese Software hätte mitbekommen, dass ihr beide Änderungen gemacht habt, und euch Werkzeuge bereitgestellt, um aus den

<sup>2</sup> <https://github.com/GitTools/GitVersion>

zwei Dateien eine einzige mit allen Änderungen zu machen (also einen »Merge« durchzuführen).

Aber auch im Nichtprogrammierumfeld trifft man auf notwendige Versionierungen von Dokumenten. Man denke an:

- *Abschlusspraesentation\_v1.ppt*
- *Abschlusspraesentation\_v2.ppt*
- *Abschlusspraesentation\_final.ppt*
- *Abschlusspraesentation\_final\_neu.ppt*

Und auch Dokumente wie Protokolle, Berichte oder Richtlinien werden häufig von mehreren Personen bearbeitet, und es existieren dadurch verschiedene Versionen davon. Hier ist zu denken an:

- *Protokoll\_Version\_Ronald.doc*<sup>3</sup>
- *Protokoll\_Version\_Christine.doc*
- *Protokoll\_Final\_verschickt.doc*
- *Protokoll\_Vom\_Chef\_abgesegnet.doc*

Versionsverwaltung ist also in vielen Bereichen ein Thema. GitHub ist ebenfalls ein Versionsverwaltungssystem, nur eben »in der Cloud« und mit Werkzeugen, die mehr –wenngleich nicht ausschließlich – mit Klicken zu bedienen sind als auf der Konsole.<sup>4</sup>

## Dezentral

Um den Punkt »dezentral« besser einordnen zu können, ist es erst einmal wichtig, zu wissen, was es sonst noch gibt. Grob unterscheidet man zwischen *lokaler*, *zentraler* und *dezentraler* Versionsverwaltung.

**Lokal** bedeutet, ich habe alle Informationen, Dateien und Funktionen vor Ort, so dass ich unabhängig von anderen Rechnern oder einer Internetverbindung arbeiten kann. Das bedeutet aber auch, dass es schwieriger ist, mit anderen zusammenzuarbeiten, und ich selbstständig dafür verantwortlich bin, Sicherungen durchzuführen. Wenn beispielsweise die lokale Platte kaputtgeht oder mein Laptop gestohlen wird und ich mich nicht ausreichend um Sicherungsmaßnahmen gekümmert habe, sind meine Daten verloren.

**Zentral** bedeutet, es gibt irgendwo einen Server, auf dem »die ganze Wahrheit« liegt. Alle Clients, die mit den Dateien arbeiten wollen, ziehen sich sogenannte Arbeitskopien auf ihren lokalen Rechner.<sup>5</sup> Dort werden die Dateien bearbeitet und

---

3 Mit einem Gruß an meine Kolleginnen und Kollegen :)

4 In Kapitel 11, Abschnitt »GitHub auf der Kommandozeile« auf Seite 245, werden wir noch sehen, dass es auch Möglichkeiten gibt, GitHub auf der Konsole zu nutzen.

5 Das nennt man auch »auschecken«.

wieder an den Server geschickt. Sollte aus irgendwelchen Gründen der Server nicht mehr funktionieren oder die Verbindung zu ihm unterbrochen sein, beschränkt das auch die Funktionen, die ich zu diesem Zeitpunkt nutzen kann. Beispielsweise ist es dann nicht möglich, die Versionshistorie einzusehen (die ist nur auf dem Server, und der will ja gerade nicht). Bekannte Anwendungen für die zentrale Versionsverwaltung sind CVS oder SVN (auch Subversion genannt).

**Dezentral** bedeutet, alle Teilnehmenden haben alle Informationen, Dateien und Funktionen auf ihren eigenen Rechnern liegen. Zusätzlich gibt es eventuell noch einen zentralen Server (das ist aber nicht zwingend notwendig). Fällt er aus oder ist die Verbindung zu ihm gestört, ist ein lokales Arbeiten mit allen Funktionen weiterhin möglich, da jeder Client das gesamte Repository gespiegelt hat. Das Arbeiten mit einer dezentralen Versionsverwaltung benötigt also keine dauerhafte Netzwerkverbindung, was beispielsweise bei einer Zugfahrt durchaus praktisch sein kann. Auch wenn eventuell ein zentraler Server mit »der ganzen Wahrheit« eingesetzt wird, sind rein technisch alle Klonen des Repositorys (einschließlich des Servers) erst einmal gleichwertig. Ein weiterer Vorteil ist: Wenn ein oder mehrere Clients z.B. aufgrund eines Verschlüsselungstrojaners Datenverlust haben, ist das Repository nicht verloren, solange noch mindestens ein Client »lebt«.



#### Definition Dezentral

Dezentral bedeutet hier also, dass alle Clients gleichwertig sind und die gleichen Daten haben.

## Exkurs: Umgang mit der Konsole

Alles, was wir mit Git machen werden, wird auf der Konsole (auch Kommandozeile, Terminal oder Eingabeaufforderung genannt) stattfinden. Dieses Kapitel ist für diejenigen, die unsicher darin sind, wie das funktioniert. Alle, die hin und wieder schon einmal mit der Konsole gearbeitet haben, können dieses Kapitel getrost überspringen.

Es gibt für Git auch grafische Benutzeroberflächen, eine Einführung würde hier aber zu weit führen. Falls du dich dafür interessierst, findest du in der Fußnote zwei Websites als Startpunkte für deine Recherche.<sup>6</sup>

Zurück zur Konsole und zur Konvention, wie die im Verlauf des Kapitels aufgeführten Befehle und ihre jeweiligen Rückmeldungen zu verstehen sind. Das Zeichen \$, der Prompt bzw. die Eingabeaufforderung, zeigt an, dass die Konsole bereit ist und du eine Eingabe machen, einen Befehl absetzen kannst. Bei dem nachfolgenden Beispiel muss also nicht das \$, sondern nur der Befehl ls -la in die Konsole

---

<sup>6</sup> <https://git-scm.com/downloads/guis/> und  
<https://git.wiki.kernel.org/index.php/InterfacesFrontendsAndTools>

eingegeben werden. Abhängig vom Betriebssystem kann auf der Konsole auch ein anderes Zeichen zu sehen sein, beispielsweise `C:\Users>` bei Windows.

```
$ ls -la
```

Die meisten – aber nicht alle<sup>7</sup> – Befehle geben in irgendeiner Form eine Rückmeldung. Bei dem obigen Befehl könnte das wie folgt aussehen:

```
insgesamt 8
drwxrwxr-x 2 user user 4096 Dez 18 19:33 .
drwxr-xr-x 71 user user 4096 Dez 18 19:09 ..
-rw----- 1 user user 46180 Jul 26 14:03 meinProjekt.java
```

Da der Befehl `ls -la` besagt: »Liste detailliert alle Dateien im aktuellen Verzeichnis auf«, sehen wir als Ergebnis alle Dateien im aktuellen Verzeichnis in einer detaillierten Auflistung. Der erste Buchstabe zeigt an, ob es sich um ein Verzeichnis (`d` wie Directory) oder eine Datei handelt, die nachfolgenden Buchstaben legen fest, welche Verzeichnis- bzw. Dateirechte gesetzt sind (`r` = read, `x` = execute, `w` = write), welchem User die Datei gehört etc.<sup>8</sup>



#### Tipp: Detailinfos zu Befehlen (Linux/Unix)

Zu fast allen Befehlen bekommt man detailliertere Informationen, wenn der entsprechende Befehl mit einem `man` davor aufgerufen wird, beispielsweise:

```
$ man ls
```

Das `man` steht für »Anleitung« (englisch *Manual*) und funktioniert bei sehr vielen Befehlen, unter anderem auch bei Git.

Wenn ich sowohl den Befehl als auch die entsprechende Rückmeldung zusammen in eine Darstellung bringen möchte, sieht das wie folgt aus:

```
$ ls -la
insgesamt 8
drwxrwxr-x 2 user user 4096 Dez 18 19:33 .
drwxr-xr-x 71 user user 4096 Dez 18 19:09 ..
-rw----- 1 user user 46180 Jul 26 14:03 meinProjekt.java
```

Und manchmal, wenn ich die allzu langen Rückmeldungen nicht alle zeigen möchte, um dich und mich nicht zu langweilen, verwende ich drei Punkte in eckigen Klammern:

```
$ ls -la
insgesamt 8
[...]
```

<sup>7</sup> Der Befehl »touch DATEINAME« legt beispielsweise eine Datei an, gibt bei Erfolg aber keine Meldung zurück.

<sup>8</sup> Nähere Infos zu dem Kommando und dessen Rückmeldung findest du unter anderem hier: <https://wiki.ubuntuusers.de/ls/>.

Und nur noch mal zur Erinnerung: Wenn ich etwas in GROSSBUCHSTABEN schreibe, bedeutet das, dass du hier deine eigene Datei, deinen Namen oder was auch immer eintragen musst, beispielsweise würdest du hier anstatt *DATEI.EXT* deine Datei namens *hallo\_welt.txt* eintragen:

```
$ vi DATEI.EXT
```

Alles klar so weit? Dann wollen wir mal starten!

## Git installieren und einrichten

Zur Installation von Git gibt es Dutzende von Büchern, Videos und anderweitige Tutorials im Internet, sodass ich hier nur in aller Kürze darauf eingehen werde. Im Abschnitt »Sich weiter schlaumachen über Git« auf Seite 160 habe ich weiteres Git-Informationsmaterial gesammelt.

Git gibt es für die Betriebssysteme Linux, Windows und macOS. Wer Windows oder macOS nutzt, kann sich die Software von der Git-Website<sup>9</sup> herunterladen. Ist Linux im Einsatz, kann für die Installation der distributionseigene Paketmanager genutzt werden. Die Website gibt auch eine Übersicht darüber, wie die Installation je nach Linux-Distribution vonstattengeht.<sup>10</sup>

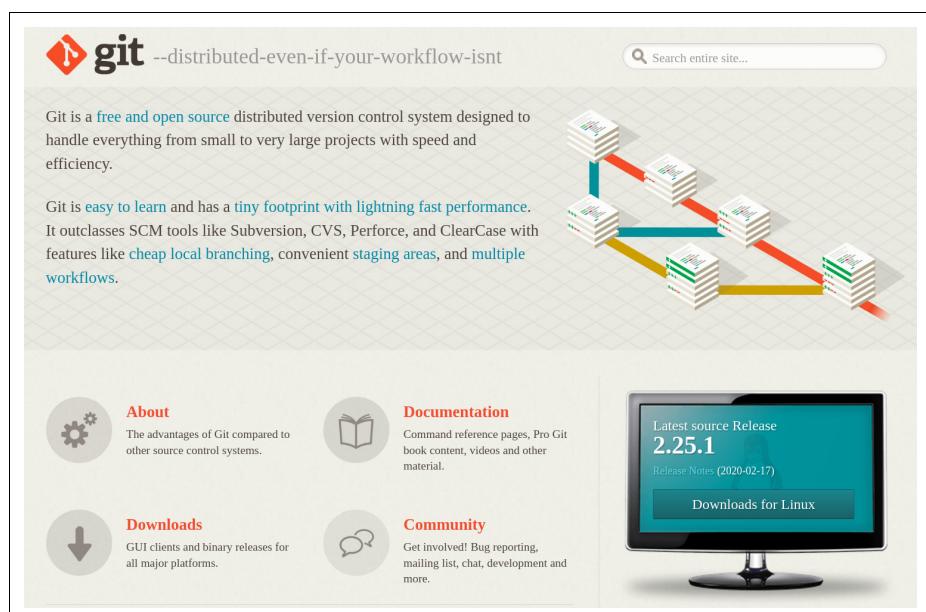


Abbildung 7-1: Die Website von Git hält alle Informationen bereit, um Git herunterzuladen und zu installieren.

9 <https://git-scm.com/downloads>

10 <https://git-scm.com/download/linux>

Wenn du Git frisch installierst, hast du in der Regel die aktuelle Version im Einsatz. Zum Zeitpunkt der Drucklegung ist das Version 2.28.0. Solltest du Git in einer älteren Version installiert haben, kann es gegebenenfalls sein, dass nicht alle Befehle so funktionieren wie erwartet oder dass bestimmte Befehle noch nicht vorhanden sind. Eventuell wäre jetzt ein guter Zeitpunkt gekommen, um die Version zu aktualisieren. Welche Version installiert ist, lässt sich wie folgt herausfinden:

```
$ git --version  
git version 2.28.0
```

Nachdem du Git installiert (oder aktualisiert) hast, solltest du es konfigurieren und unter anderem einen Namen und eine Mailadresse vergeben. Der Name ist dabei frei wählbar und wird dafür genutzt, deine zukünftigen Aktivitäten (wie Commits und Ähnliches) im System entsprechend zu dokumentieren. Das kann dein echter Name sein oder aber auch ein Spitzname. So geht es:

```
$ git config --global user.name "VORNAME NACHNAME"  
$ git config --global user.email "USER@MAIL.DE"
```

Die Texte in den Hochkommata ("") musst du natürlich auf deinen Usernamen und deine Mailadresse anpassen. Es ist auch möglich, für jedes Repository einen eigenen Namen und eine eigene Mailadresse zu vergeben. Das kann insbesondere dann sinnvoll sein, wenn man ein Repository beruflich nutzt und ein anderes privat.<sup>11</sup> Dafür musst du im entsprechenden Git-Verzeichnis sein und anstelle von --global den Schalter --local setzen.

```
$ cd meinGitProjekt  
$ git config --local user.name "Die maskierte Raecherin"  
$ git config --local user.email "maske@maildomain.de"
```



### **Mailadresse geheim halten**

Wer bei Git keine eigene Mailadresse angeben möchte, kann die von GitHub generierte nutzen. Diese wurde bereits in Kapitel 3 im Abschnitt »Account anlegen« auf Seite 20 erläutert und sieht wie folgt aus: ID+username@users.noreply.github.com (beispielsweise 1234567+geheimniskraemer@users.noreply.github.com).

Um die Parameter noch mal (oder zu einem späteren Zeitpunkt) zu überprüfen, funktioniert das mit den gleichen Befehlen wie beim Einrichten, nur ohne die Werte in den Hochkommata, beispielsweise:

```
$ git config --local user.email  
maske@maildomain.de
```

Das ist insbesondere dann empfehlenswert, wenn viele Repositories mit unterschiedlichen Mailadressen im Einsatz sind – wäre vielleicht peinlich, wenn die »Rächerin«-Leute deine »Rosa Kaninchen«-Mailadresse sähen, nur weil etwas falsch konfiguriert

---

<sup>11</sup> Oder wenn dich die Leute in einem Projekt als »maskierte Rächerin« kennen und in einem anderen als »rosa Kaninchen«.

ist. Falls übrigens ein Parameter nicht gesetzt ist (beispielsweise die Mailadresse für ein einzelnes Projekt), bekommst du gar keine Rückmeldung.

Eine Sache solltest du auf jeden Fall einstellen, und zwar die Farbausgabe auf der Benutzungsschnittstelle (*User Interface*, häufig UI abgekürzt). Diese macht das Arbeiten auf der Konsole etwas übersichtlicher. Das geht wie folgt:

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

Praktischerweise kann man Git so einrichten, dass der eigene Lieblingseditor immer dann aufgerufen wird, wenn Git irgendeine Form von Eingabe benötigt – etwa weil du vergessen hast, bei einem Commit eine Commit-Nachricht anzugeben. Das geht wie folgt:

```
$ git config --global core.editor vim  
$ git config --global core.editor nano  
$ git config --global core.editor "atom --wait"
```

Die drei Zeilen zeigen die Einrichtung von drei unterschiedlichen Editoren (Vim, Nano und Atom). Für Editoren, die nicht originär auf der Konsole zu Hause sind – Atom in diesem Beispiel –, musst du Git noch den Hinweis geben, dass es auf das Beenden des Editors warten soll (wait), da ansonsten das Zusammenspiel von Editor und Git nicht funktioniert.



### Konfigurationsdateien für Git

Falls du die Konfigurationsdateien von Git suchst, um sie manuell im Nachgang noch mal anzupassen: Die globale Konfigurationsdatei liegt im Home-Verzeichnis und heißt `.gitconfig` (also `/home/user/.gitconfig`), die lokale Konfigurationsdatei liegt im entsprechenden Repository-Verzeichnis im Unterverzeichnis `.git` und heißt `config` (also `/home/user/repo/.git/config`).

Jetzt ist Git so weit eingerichtet, dass wir damit arbeiten können. Wenn du die Konsolenausgabe noch weiter individualisieren möchtest, schau dir den nachfolgenden Abschnitt an. Überspring ihn ruhig, falls dir das erst einmal (noch) nicht so wichtig ist.

## Exkurs: Die Konsole für Git einrichten am Beispiel Bash (für Fortgeschrittene)

Falls du viel mit Git auf der Konsole arbeitest, lohnt es sich, die Umgebung so einzurichten, dass benötigte Informationen automatisch angezeigt und/oder farblich hervorgehoben werden. Das Thema dieses Abschnitts ist eher für Fortgeschrittene gedacht. Wir wollen zwei Anpassungen an der Konsole vornehmen:

1. Der aktuelle Branch soll angezeigt werden.
2. Der Branch soll in einer gewählten Farbe angezeigt werden.

Je nach Shell (siehe Erklärbärbox »Shell«) ist die Herangehensweise etwas anders. Ich zeige exemplarisch für die Bash (*Bourne again shell*), wie es funktioniert. Anleitungen für andere Shells, wie beispielsweise Zsh oder PowerShell, gibt es in der Dokumentation zu Git.<sup>12</sup>

## Shell

Die Konsole ist im Gegensatz zu einer grafischen Schnittstelle eine textbasierte Schnittstelle zum Betriebssystem. Eine Shell (deutsch »Schale« oder »Außenhülle«) ist dabei die Software, die in der Konsole läuft. Der Begriff Shell röhrt daher, dass das Betriebssystem (beispielsweise Windows, macOS, Linux) als »Kern« betrachtet wird und die Software, um auf diesen Kern zuzugreifen, die »Schale« bildet.

In der Regel kristallisiert sich nach dem Ausprobieren verschiedener Shells eine Lieblingsshell heraus.<sup>13</sup> Es ist aber durchaus auch nicht ungewöhnlich, die Shell regelmäßig zu wechseln. Ich kenne beispielsweise einen Administrator, der eine sehr einfache Shell als Standard-Shell eingetragen hat, aber beim Nutzen der Konsole sofort auf eine andere, mächtigere Shell wechselt. Er macht das »aus Sicherheitsgründen«, da er immer Angst hat, dass die mächtigere Shell mal nicht funktionieren könnte und er nicht mehr auf das System kommt. In seinen Augen ist die von ihm gewählte Standard-Shell zuverlässiger.

## Drei Schritte, um einen Branch farbig anzuzeigen

Wir wollen den aktuellen Branchnamen in der Konsole angezeigt bekommen und ihn in einer gewählten Farbe hervorheben. Am Ende wird das Ganze wie folgt aussehen (hier im Buch simuliert der fette Text die Farbe):

```
username@rechnername ~/gitdirectory (branchname)$
```

Um dieses Ziel zu erreichen, sind folgende Schritte notwendig:

1. Die Datei *git-prompt.sh* herunterladen<sup>14</sup> und an einem geeigneten Ort abspeichern, z. B. im Home-Verzeichnis (in der Regel */home/username/*).
2. Die Bash-Konfigurationsdatei *.bashrc*<sup>15</sup> lokalisieren (in der Regel im Home-Verzeichnis). Fehlt die Datei? Einfach mit einem Texteditor anlegen.
3. Die *.bashrc* anpassen, um den Git-Branch farbig darstellen zu können (siehe nächsten Abschnitt).

12 In »A1. Appendix A: Git in Other Environments« auf der Seite <https://www.git-scm.com/book/en/v2>.

13 Meine ist beispielsweise die Bash.

14 <https://github.com/git/git/tree/master/contrib/completion>

15 Bei manchen Betriebssystemen kann das eventuell die Datei *.bash\_profile* sein.

## Anpassen der .bashrc

Wir ergänzen die `.bashrc` um einige Zeilen Code, um den aktuellen Branch anzeigen zu lassen und um eine Farbe dafür festzulegen. Öffne sie in einem Editor deiner Wahl und füge Nachfolgendes hinzu:

```
# Definieren der Farbe
color='\[0;36m'          # Cyan
colorreset='\[0m'        # Text zurücksetzen
# Farbdefinition Ende

# Farbe und Anzeige des Branchs bei Git
. ~/git-prompt.sh # Pfad zur heruntergeladenen Datei
export PS1='\u@\h \w$(_git_ps1 "($color%$colorreset)")\$ '
```

Die obersten Zeilen legen zwei Variablen fest, zum einen die gewünschte Farbe (hier Cyan) und zum anderen eine Variable (`colorreset`), die später für das Entfernen der Farbe da ist, sobald diese nicht mehr benötigt wird. Für andere Farben oder Formatierungsmöglichkeiten schau dir die Erklärbarbox »Textformatierung des Prompts« an.

### Textformatierung des Prompts

Für die Textformatierung des Prompts wird eine sogenannte ANSI-Escape-Sequenz verwendet. Bei dem Beispiel `\e[0;36m` leitet das `\e` diese Sequenz ein und sagt der Konsole, dass alles, was zwischen `[` und `m` steht, als Textformatierung zu werten ist. Die Zahl vor dem Semikolon ist dabei eine Formatierung (fett, unterstrichen etc.), die Zahl nach dem Semikolon eine Farbe. Einige gebräuchliche Werte findest du in den Tabelle 7-1 und Tabelle 7-2. Für blinkende rote Schrift wählst du `5;31`, für fette blaue Schrift `1;34`.

*Tabelle 7-1: Formatierungsmöglichkeiten des Prompts*

Code	Formatierung
0	normale Formatierung
1	fett, teilweises Aufhellen einer Farbe
3	kursiv
4	Text unterstreichen an
5	Blinken an
7	Text hervorheben an
8	Text verstecken
23	kursiv aus
24	Text unterstreichen aus
25	Blinken aus
27	Text hervorheben aus

*Tabelle 7-2: Formatierungsmöglichkeiten des Prompts durch Farbe*

Code	Farbe
30	Schwarz
31	Rot
32	Grün
33	Braun
34	Blau
35	Lila
36	Cyan
37	Grau

Nach den Variablen für die Farbe bzw. die Farbentfernung müssen wir den Pfad zu der eben heruntergeladenen Datei angeben. Die letzte Zeile ist sicherlich die knackigste. Alles, was hinter dem Gleichheitszeichen kommt (`export PS1=`), legt die Darstellung der Konsole fest. Das heißt, wenn du Anpassungen vornehmen möchtest, dann dort. Abbildung 7-2 zeigt, wie eingesetzte Befehle und die Wunschausgabe miteinander verknüpft sind.

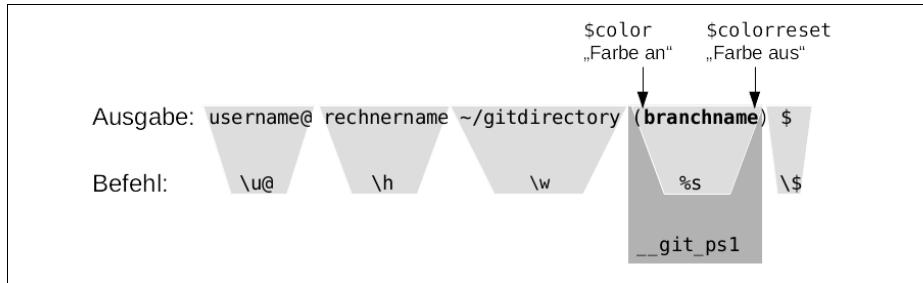


Abbildung 7-2: Verknüpfung der eingesetzten Befehle mit der gewünschten Ausgabe (mit zusätzlichen Leerzeichen für eine bessere Lesbarkeit)

- `\u001b@` bedeutet, dass der Username angezeigt werden soll, gefolgt von einem @-Zeichen.
- `\h` ist der Name des Rechners (das h steht für *Host*).
- `\w` steht für *working directory* (deutsch »Arbeitsverzeichnis«), mit einem Leerzeichen vorweg, weil es schöner aussieht.
- `git_ps1` ist ein Befehl aus *git-prompt.sh* und sorgt dafür, dass der Prompt den Git-Status (in unserem Fall den aktuellen Branch) versteht.
- `$color` ist die von uns gewählte Farbe.
- `%s` ist eine Variable aus *git-prompt.sh*, die den Git-Status (den aktuellen Branch) wiedergibt.
- `$colorreset` ist das Zurücksetzen der Farbe, damit nicht alles cyan wird. Lass die Variable ruhig mal weg und schau dir an, was es bewirkt.
- `\$` legt fest, wie der Prompt aussieht, in diesem Fall `$`. Du kannst diesen auch anpassen, beispielsweise auf `\>`, wenn dir das lieber ist.

Du kannst jetzt noch etwas damit herumspielen, die Farbe anpassen, die unterschiedlichen Zeichen wie @ oder \$ verändern oder auch bestimmte Infos wie den User- oder Hostnamen entfernen. Falls du beim Ausprobieren Fehler machst, ist das nicht dramatisch. Die Konsole gibt entsprechende Rückmeldungen, lässt sich aber weiter nutzen, sodass du die Fehler bereinigen kannst.

## Tiefer einsteigen

Tiefer möchte ich jetzt nicht in diese Konfigurationsmöglichkeiten einsteigen. Ich hoffe, du hast dir einen ersten Einblick in das verschaffen können, was alles mög-

lich ist, und es animiert dich zum weiteren Experimentieren. Die *git-prompt.sh* ermöglicht beispielsweise weitere Anpassungen, wie anzusehen, ob es Änderungen gibt, die noch nicht im Staging-Bereich<sup>16</sup> sind, oder ob es Dateien gibt, die noch nicht im Repository sind (*Untracked Files*). Falls das für dich interessant ist, lohnt sich ein Blick in den Quelltext. Dort sind diese Anwendungsfälle dokumentiert.

Darüber hinaus gibt es im Internet Werkzeuge, mit denen du dir per Drag-and-drop aus einzelnen Elementen deinen Wunschprompt zusammenstellen kannst. Die Zusammenstellung wird gleich als Vorschau angezeigt, und die Werkzeuge generieren den notwendigen Code für die Anpassung der *.bashrc*, beispielsweise <http://ezprompt.net> oder <http://bashrcgenerator.com>.

Wenn du tiefer einsteigen möchtest, findest du in der Fußnote noch mehr Web-sites zur Eigenrecherche.<sup>17</sup>

## Wie Git tickt – Staging

Um zu verstehen, wie die Versionierung bei Git funktioniert, ist es gut, erst einmal zu sehen, wie Git tickt. Git unterscheidet die Bereiche *Arbeitsverzeichnis*, *Staging-Bereich* und *Repository* (siehe Abbildung 7-3).

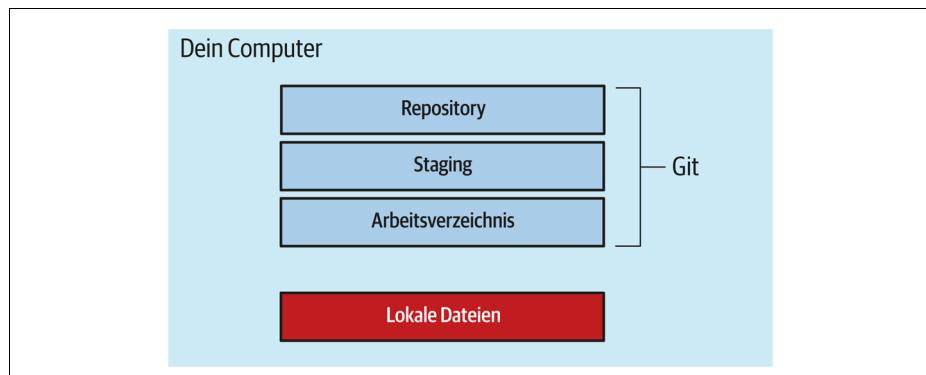


Abbildung 7-3: Git unterscheidet die Bereiche *Arbeitsverzeichnis*, *Staging* und *Repository*.

Das *Arbeitsverzeichnis* (auch *Working Directory* oder *Workspace* genannt) ist das Verzeichnis, in dem dein Projekt liegt. Git weiß dabei genau, ob in diesem Ver-

16 Den Staging-Bereich lernst du gleich noch im nächsten Abschnitt kennen.

17 <http://www.faqs.org/docs/Linux-HOWTO/Bash-Prompt-HOWTO.html>

<https://www.thegeekstuff.com/2008/09/bash-shell-ps1-10-examples-to-make-your-linux-prompt-like-angelina-jolie>

<https://web.archive.org/web/20160704140739/http://ithaca.arpinum.org/2013/01/02/git-prompt.html> – da es die Original-Website nicht mehr gibt, ist dies ein Link auf eine Archiv-Website. Daher sieht der Link evtl. etwas merkwürdig aus.

<https://blog.grahampoulter.com/2011/09/show-current-git-bazaar-or-mercurial.html>

zeichnis neue Dateien hinzugekommen sind, ob die Dateien bereits bekannt und unverändert sind oder ob und wann sie geändert wurden.

Im Bereich *Staging* werden die veränderten oder neuen Dateien »zwischengelagert«, die mit dem nächsten Commit in das Repository geschoben werden sollen. Staging könnte man übersetzen mit »auf die Bühne bringen«, wir bringen also einige – aber nicht immer unbedingt alle – veränderten Dateien auf die Commit-Bühne (siehe hierzu auch die Erklärbärbox »Staging-Bereich«).

Alles, was im Bereich *Repository* angelangt ist, ist (jetzt endlich) unter der Versionskontrolle von Git.<sup>18</sup> Der Standardfluss der Daten verläuft in der Regel von unten nach oben: Lokale Dateien werden in den Arbeitsbereich kopiert, gelangen dann in den Staging-Bereich und danach in das Repository. Es gibt aber auch die Möglichkeit, beispielsweise Daten aus dem Staging-Bereich zurück in den Arbeitsbereich zu verfrachten.

## Staging-Bereich

Der *Staging-Bereich* erscheint vom Konzept her erst einmal unnötig. Das geht nicht nur vielen Neulingen so, auch viele Expertinnen, die bisher andere Versionsverwaltungssysteme genutzt haben, sind davon zunächst verwirrt. Warum gibt es diesen Zwischenbereich?

Ich versuche, es mit einem Bild zu verdeutlichen. Stell dir eine Fabrik (das Arbeitsverzeichnis) vor, die unterschiedliche Waren (Bugfixes, neue Features) erzeugt. In einer Lagerhalle (Staging-Bereich) werden fertige Waren zwischengelagert, bis ein Kleinlaster (Commit) kommt, um diese abzuholen. Alle Lasterfahrer haben den Auftrag, alles in ihren Laster zu packen, was in der Lagerhalle steht, und jeder Laster fährt nur von der Lagerhalle zu *einem anderen Ort*.<sup>19</sup> Die Ware »Hundefutter« soll zum Tierheim, die Ware »Rollator« in die Seniorenresidenz. Wenn man jetzt nicht möchte, dass das Tierheim ohne Futter dasteht und die Residenz sich über einen Schwung Rollatoren und Hundefutter freuen darf, müssen die beiden Warengruppen getrennt in die Lagerhalle gestellt werden. Also wird beispielsweise erst das Hundefutter gelagert, der Laster wird abgewartet, und dann erst werden die Rollatoren gelagert und abgeholt.

Genauso läuft das auch bei Git und dem Staging-Bereich. Wenn meine »Ware« ein Bugfix ist, sollte ich ihn nicht zusammen mit einem »neuen Feature« in den Staging-Bereich bringen. Der Staging-Bereich gibt dir die Kontrolle und Flexibilität, um Commits exakt zu »portionieren«.

<sup>18</sup> Man nennt das auch: Die Datei wurde eingeccheckt.

<sup>19</sup> Wir blenden die ökonomischen und ökologischen Auswirkungen dieser Vorgehensweise mal aus.

Um Dateien zwischen den verschiedenen Bereichen zu bewegen, müssen bestimmte Befehle und Aktivitäten ausgeführt werden. Abbildung 7-4 und der Kasten »Standardarbeitsablauf mit Git« geben einen Überblick über den Standardablauf. Damit Git weiß, was unser Arbeitsverzeichnis ist, muss dieses einmalig mit `init` initialisiert werden.

Dateien, die ich lokal (irgendwo) liegen habe, kann ich in mein Arbeitsverzeichnis kopieren, neu erzeugen oder verschieben. In der Grafik habe ich den Linux-Befehl `cp` (für `copy` – »kopieren«) stellvertretend für diese Aktivität geschrieben.

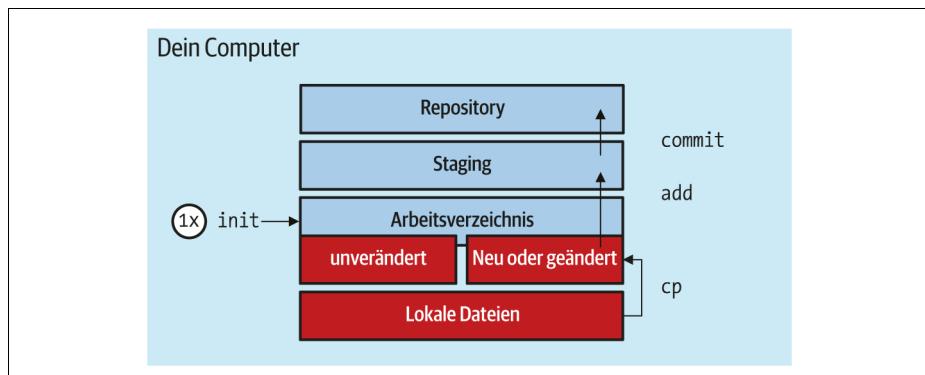


Abbildung 7-4: Um zwischen den unterschiedlichen Bereichen Dateien hin- und herzubewegen, sind unterschiedliche Befehle notwendig

Dateien in meinem Arbeitsverzeichnis füge ich mit `add` dem Staging-Bereich hinzu. Und vom Staging-Bereich komme ich via `commit` in das lokale Repository.

## Standardarbeitsablauf mit Git

Ein Git-Repository richtest du so ein:

```
$ mkdir VERZEICHNIS  
$ cd VERZEICHNIS  
$ git init
```

Geänderte oder neue Dateien bekommt man so in ein Git-Repository:

```
[...Dateien ändern oder dem Verzeichnis hinzufügen...]  
$ git add DATEI  
$ git commit -m "NACHRICHT"
```

Diese einzelnen Befehle schauen wir uns jetzt konkret an einem Beispiel an.

# Das eigene Projekt mit Git verwalten

Die Befehle und Aktivitäten aus dem letzten Abschnitt wollen wir jetzt gemeinsam ausprobieren. Wie schon angekündigt, ignorieren wir erst einmal, dass es GitHub (oder andere Plattformen) gibt. Wir werden rein lokal arbeiten, um Git kennenzulernen und den Umgang etwas einzuüben.

## Das Arbeitsverzeichnis initialisieren

Als Erstes müssen wir Git sagen, wo unser Arbeitsbereich ist. Das nennt sich initialisieren. Erstelle bzw. wähle hierfür ein leeres Verzeichnis, in dem du dein Projekt verwalten möchtest. In meinem Fall erstelle ich ein neues, leeres Verzeichnis *gittest*:

```
$ mkdir gittest
```

Wir werden mit bzw. in diesem Verzeichnis etwas rumspielen. Auf der Konsole wechseln wir in das entsprechende Verzeichnis:

```
$ cd gittest/
```

In diesem Verzeichnis bleiben wir bei allen folgenden Befehlen. Wenn also etwas nicht klappt: in das Verzeichnis wechseln und noch mal probieren. Wir machen das Verzeichnis Git wie folgt bekannt:

```
$ git init  
Leeres Git-Repository in /home/user/gittest/.git/ initialisiert
```

Das war's auch schon! Bei dir erscheint jetzt als Rückmeldung das von dir gewählte Verzeichnis. Was ist passiert? Git hat mit diesem Befehl ein Unterverzeichnis namens *.git* erzeugt und dort alle wichtigen Informationen abgespeichert. Der Punkt (.) vor dem Namen bedeutet in der Linux-/Unix-Welt, dass es sich hierbei um ein verstecktes Verzeichnis handelt und es daher gegebenenfalls nicht sichtbar ist. Du kannst dorthin wechseln wie in jedes andere Verzeichnis auch, entweder auf der Konsole:

```
$ cd .git/
```

oder über einen grafischen Dateimanager, indem du in den Dateipfad ein *.git/* (für beispielsweise Linux) einfügst. Wenn du möchtest, kannst du in das erzeugte *.git*-Verzeichnis wechseln und dir anschauen, welche Dateien und Ordner dort angelegt wurden. Für das weitere Verständnis ist das aber nicht notwendig, deswegen werde ich hier nicht näher darauf eingehen.



### Tipp: Git-Kontrolle rückgängig machen

Wenn du aus irgendwelchen Gründen möchtest – beispielsweise weil du etwas nur zu Testzwecken unter Git-Kontrolle gebracht hast oder vielleicht die Versionsverwaltung wechseln willst –, dass ein Verzeichnis nicht mehr unter der Verwaltung von Git stehen soll, lösche einfach das Unterverzeichnis *.git*.

Jetzt haben wir Git gesagt, wo dein Projekt liegen wird und dass es sich um die Versionierung dieses Projekts kümmern soll.

## Eine neue Datei ins lokale Repository einfügen

In unserem neuen Verzeichnis, das Git ab sofort überwacht, erzeugen wir eine neue Textdatei. In Kapitel 2, Abschnitt »Informationen finden (interessierte Anwenderin)« auf Seite 13, haben wir gesehen, wie wichtig eine gute *README.md* ist, also legen wir am besten eine solche mal an. Unter Linux könnte das über den Befehl `touch` in der Konsole erfolgen:

```
$ touch README.md
```

Aber auch jede andere Form der Erzeugung ist möglich, beispielsweise über den grafischen Dateimanager in Windows per Rechtsklick und *Neu/Textdatei*. Du musst noch nichts in die Datei reinschreiben, es reicht, wenn sie existiert. Sobald du die Datei in deinem Verzeichnis erzeugt hast, schauen wir mal, was Git zu dieser neuen Situation sagt. Dafür gibt es nachfolgenden Befehl auf der Konsole:

```
$ git status
Auf Branch master
Noch keine Commits
Unversionierte Dateien:
  (benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)
  README.md
```

nichts zum Commit vorgemerkt, aber es gibt unversionierte Dateien  
(benutzen Sie "git add" zum Versionieren)

Jetzt haben wir eine Menge Rückmeldungen von Git bekommen.

- **Auf Branch master** – Wir befinden uns aktuell auf dem master-Branch – Branches hast du bereits in Kapitel 4 im Abschnitt »Branch – unterschiedliche Handlungsstränge aufmachen« auf Seite 44 kennengelernt. Git nennt den Hauptzweig – wie auch GitHub – immer master, und da wir noch keine eigenen Branches angelegt haben, gibt es folglich nur diesen einen Branch, auf dem wir uns gerade befinden.
- **Noch keine Commits** – Da wir ein frisches neues Repository angelegt haben, gibt es derzeit exakt null Commits. Diese Mitteilung wird nach dem ersten Commit nicht mehr zu sehen sein.
- **Unversionierte Dateien [...]** – Git erkennt offensichtlich, dass wir eine neue Datei *README.md* im Arbeitsverzeichnis angelegt haben und dass diese noch nicht unter der Versionsverwaltung von Git ist. Git gibt uns zudem einen Hinweis, wie wir damit umgehen sollten (Befehl `git add` verwenden).
- **nichts zum Commit vorgemerkt [...]** – Hier gibt uns Git erneut den Hinweis, wie wir das Problem mit den unversionierten Dateien beheben können. Das schauen wir uns am besten mal genauer an.

Wie du ja bereits weißt, reicht es nicht, einfach nur eine Datei in unserem von Git verwalteten Verzeichnis abzulegen, um sie zu versionieren. Wir müssen Git noch sagen, welche der vorhandenen Dateien verwaltet werden sollen.

Um diese – für Git – neue Datei aus dem Arbeitsverzeichnis in den Staging-Bereich zu bringen, nutzen wir den Befehl add (siehe auch Abbildung 7-4):

```
$ git add README.md
```

Damit ist die neue Datei im Staging-Bereich gelandet. Wenn wir sie jetzt in das lokale Repository bringen wollen, müssen wir einen Commit durchführen und am besten auch gleich mit einer aussagekräftigen Commit-Nachricht versehen:

```
$ git commit -m "Füge README.md hinzu"  
[master (Root-Commit) db2bf6c] Füge README.md hinzu  
 1 file changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 README.md
```

Git gibt uns als Rückmeldung, auf welchem Branch es committet hat (master), dass es unser erster Commit ist (Root-Commit), welche Commit-ID dieser Commit hat (db2bf6c, die Commit-ID kennen wir schon aus Kapitel 3) und noch ein paar Informationen darüber, wie viele Dateien sich verändert haben und in welcher Form.

Vielleicht erinnerst du dich: In Kapitel 3, Abschnitt »Eine inhaltliche Änderung am Projekt vornehmen« auf Seite 26, habe ich gesagt, dass Commit so etwas wie Speichern sei. Hier hast du aber schon lange vorher deine Datei gespeichert, und erst jetzt kommt ein Commit. Das ist ein Unterschied zu GitHub. Git ist viel feingranularer mit seinen unterschiedlichen Bereichen und ermöglicht dir mehr Kontrolle darüber, was in einen Commit kommt und was vielleicht erst einmal noch nicht. Auch einer der Gründe dafür, dass GitHub allein manchmal nicht ausreicht.



#### Definition Commit für Git und GitHub

Commit bedeutet also sowohl für Git als auch für GitHub »Füge etwas zum Repository hinzu« – nur Git erlaubt im Vorfeld, dieses Hinzufügen feingranularer zu steuern.

## Eine Datei im lokalen Repository ändern

Unsere *README.md* ist jetzt im lokalen Repository. Angenommen, wir möchten die Datei ändern und fügen etwa eine neue Zeile hinzu. Was sagt Git nach dem Abspeichern dazu?

```
$ 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 restore <Datei>...", um die Änderungen im Arbeitsverzeichnis zu verwerfen)  
geändert:      README.md  
  
keine Änderungen zum Commit vorgemerkt (benutzen Sie "git add" und/  
oder "git commit -a")
```

Die Rückmeldung sieht etwas anders aus, das liegt daran, dass Git ein neues leeres Repository etwas anders betrachtet als ein bereits »benutztes«. Die Rückmeldung gibt uns aber wieder gute Hinweise darauf, was als Nächstes zu tun ist. Wir machen also wieder vom Ablauf her exakt das Gleiche wie bei einer neuen Datei, nur dieses Mal mit einer anderen Commit-Nachricht:

```
$ git add README.md  
$ git commit -m "neues Feature"
```

Damit hast du schon die beiden wichtigsten Git-Befehle kennengelernt.

## Dateien von der lokalen Versionsverwaltung ausschließen

Eine Sache sollten wir noch einrichten, die insbesondere dann relevant wird, wenn in deinem Projekt Dateien enthalten sind oder sein werden, die nicht unbedingt der Versionskontrolle unterliegen sollen.

Im Programmierumfeld sind das häufig Logdateien oder andere »Abfallprodukte«, die beim Programmieren entstehen. Beim Schreiben dieses Buchs habe ich beispielsweise eine Datei *Recherche.txt* angelegt, in der ich alle noch offenen Fragen notiert habe. Die Datei sollte aber nicht Teil des Buchs werden. Es nervt jetzt kolossal, wenn man solche Dateien beim Arbeiten mit Git regelmäßig aufs Brot geschmiert bekommt: »Hei, du hast die Datei *Recherche.txt* immer noch nicht unter die Versionsverwaltung gebracht.«

Andere Dateien, die besser nicht der Versionsverwaltung unterzogen werden sollten, könnten geheime Informationen wie bspw. Kryptoschlüssel oder Passworddateien sein. Das ist insbesondere dann relevant, wenn du nicht mehr lokal und allein auf einem Projekt arbeitest.

Zu diesem Zweck legen wir in unserem Arbeitsverzeichnis jetzt eine Datei mit dem Namen *.gitignore* an (den Punkt am Anfang des Namens beachten!):

```
$ touch .gitignore
```

Es handelt sich dabei erst einmal um eine reine (und in unserem Fall noch leere) Textdatei, und diese kannst du ohne Probleme mit jedem Textprogramm öffnen, z.B. mit Notepad bei Windows oder vi bei Linux. An sich beschreibt der Name *.gitignore* es schon recht treffend: Alles was in dieser Datei steht, wird von Git ignoriert – allerdings nicht die Datei *.gitignore* selber.<sup>20</sup>

Eine *.gitignore* könnte beispielsweise wie folgt aussehen:

```
# Zwei Einzeldateien  
Recherche.txt  
.gitignore
```

---

<sup>20</sup> Wenn du das möchtest, musst du den Dateinamen explizit angeben.

```
# Ein ganzes Verzeichnis  
geheimeDateien/  
  
# Bestimmte Dateiendungen in einem Ordner  
/images/*.png
```

Ähnlich wie bei der Datei CODEOWNERS (siehe Abschnitt »Reviews automatisch zuweisen – CODEOWNERS« auf Seite 65 in Kapitel 4) sind die Satzzeile beginnend mit der Raute (#) Kommentare, die bei der Auswertung ignoriert werden. Falls du dich intensiver mit der Syntax für diese Datei beschäftigen möchtest, empfehle ich einen Blick in die Dokumentation.<sup>21</sup>

Befülle die Datei jetzt, so wie es dein Projekt braucht. Falls du noch nicht sicher bist, kannst du sie aber auch erst einmal leer lassen und später befüllen.

Vielleicht ist dir die Datei auch schon bei fremden Projekten auf GitHub aufgefallen? GitHub selbst empfiehlt sogar, sie mit ins Repository hochzuladen, damit auch die Mitentwickler am Projekt nicht ihre eigene Datei bauen müssen.



#### Tipp: Vorlagen für .gitignore

Da es Fälle gibt, in denen viele Menschen die gleichen Dateien/Verzeichnisse ignorieren wollen, gibt es auch ein GitHub-Repository mit Vorlagen (<https://github.com/github/gitignore>). Beispielsweise sind in der `Java.gitignore` alle für Java-Entwicklungen gängigen Dateiendungen aufgeführt, die man üblicherweise nicht im Repository haben möchte. Stark gekürzt, sieht das wie folgt aus:

```
# Compiled class file  
*.class  
  
# Log file  
*.log  
[...]
```

Ein weiteres nützliches Werkzeug ist in diesem Zusammenhang <https://gitignore.io/>. Dort gibt man beispielsweise das eingesetzte Betriebssystem und benutzte Anwendungen an und kann sich daraus eine `.gitignore`-Datei generieren lassen.

## Branching in Git

Eine Sache schauen wir uns noch an, nämlich wie man Branches in Git anlegt und auch zwischen ihnen wechselt.

## Branches erzeugen

Mit dem Befehl `git branch` kann man Branches erzeugen und sich den aktuellen Branch anzeigen lassen. Wir fangen mit Letzterem an und tippen den Befehl ohne weitere Parameter in die Konsole:

21 <https://git-scm.com/docs/gitignore>

```
$ git branch  
* master
```



### Tipp: Initialer Commit

Du hast ein neues Git-Repository angelegt, und manche Befehle erzeugen nicht die gewünschte Wirkung? Dann liegt das eventuell daran, dass noch nichts in das Repository committet wurde. Mach einen initialen Commit und probiere den Befehl dann erneut.

Bei einem neuen Repository oder einem ohne Branches sollte deine Ausgabe so aussehen wie meine. Sprich, es gibt nur den master-Branch. Das Sternchen vor dem Namen zeigt an, dass wir uns aktuell auf diesem Branch befinden. Einen neuen Branch erzeugen wir wie folgt:

```
$ git branch mein-feature
```

Damit erzeugen wir einen Branch namens *mein-feature*. Wenn wir uns jetzt die Branchübersicht anschauen, ist der neue Branch hinzugekommen, wir befinden uns aber weiterhin auf dem master-Branch:

```
$ git branch  
  mein-feature  
* master
```

Ein Branch von einem anderen Branch können wir so erzeugen:

```
$ git branch NEUERBRANCH ALTERBRANCH
```

Also zum Beispiel:

```
$ git branch erweitertes-feature mein-feature
```

Wird *ALTERBRANCH* nicht angegeben, wird der master-Branch verwendet (haben wir bereits weiter oben angewendet). Du kannst auch einen Branch ab einem festgelegten Commit erzeugen. Das geht mit der (abgekürzten) Commit-ID (in meinem Fall 9edb5) genauso einfach:

```
$ git branch mein-weiteres-feature 9edb5
```

## Zwischen Branches wechseln

Für das Wechseln zwischen Branches gibt es den Befehl `git checkout` zusammen mit dem Branchnamen, auf den du wechseln möchtest:

```
$ git checkout mein-feature  
Zu Branch 'mein-feature' gewechselt
```

Jetzt sollte sich auch das Sternchen verschieben:

```
$ git branch  
  mein-weiteres-feature  
  erweitertes-feature  
* mein-feature  
  master
```

Der Befehl `git branch` erzeugt zwar einen neuen Branch, wenn du danach aber gleich auf dem Branch direkt arbeiten willst, müsstest du noch den zweiten Befehl `git checkout` eintippen. Wenn dir das zu umständlich ist, geht das alles auch in einem Rutsch mit:

```
$ git checkout -b neuer-branch  
Zu Branch 'neuer-branch' gewechselt
```

Der Parameter `-b` sorgt dafür, dass ein neuer Branch angelegt wird, und es wird auch sofort auf diesen gewechselt. Der angegebene Branch darf aber noch nicht vorhanden sein, sonst gibt es eine Fehlermeldung. Damit bist du jetzt für die wichtigsten Branching-Aufgaben mit Git gerüstet.

Eine kleine, aber nicht ganz unwichtige Sache zu Git möchte ich dir noch zeigen, und zwar wie du mit Binärdateien in Git umgehen kannst und warum das überhaupt ein Thema ist.

## Binärdateien mit Git verwalten

Wenn du mit Git ein Projekt verwaltetest, kommt es häufig vor, dass neben reinen Textdateien auch Binärdateien wie Bilder oder Ähnliches verwaltet werden sollen (z.B. bei einer Website oder einer App). Je nachdem, wie groß diese Binärdateien sind und wie häufig sie angepasst werden müssen, kann man sich das eigene Repository damit ganz schön aufblähen. Warum? Das hat ein Stück weit damit zu tun, wie Git die Dateien verwaltet.

Git ist hocheffizient darin, Textdateien zu verwalten und dabei ein Repository klein und schlank zu halten.<sup>22</sup> Mit Binärdateien kann Git allerdings nicht so effizient umgehen. Beispiel: Du hast ein 20-MByte-Bild in deinem Repository unter Kontrolle von Git. Jedes Mal, wenn du dieses Bild änderst und committest, legt Git eine neue 20-MByte-Datei für die neue Version des Bilds an. Die alte Version mit ihren 20 MByte ist aber auch noch da. Mach das zehnmal, und du hast 10 x 20 MByte extra auf der Festplatte.

Dass das bei Projekten mit vielen Binärdateien, die sich häufig ändern, zu einem großen Platzproblem führen könnte, ist offensichtlich. Das Schöne ist – wie immer –, dass es auch hierfür eine Lösung gibt, die sehr einfach zu installieren und zu nutzen ist. Sie nennt sich *Git LFS*, (das steht für *Git Large File Storage*) und ist eine Erweiterung für Git. Sie sorgt dafür, dass sich das Git-Repository dadurch nicht unnötig aufbläht, da sie entsprechende Dateien anders verwaltet.

---

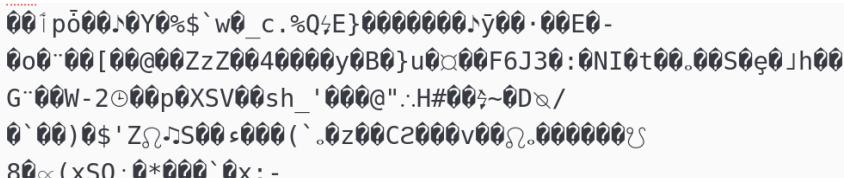
<sup>22</sup> Das hat unter anderem mit den Hashwerten zu tun, die du bei der Commit-ID in der Erklärbärbox »Commit-ID« in Kapitel 3 kennengelernt hast.

## Unterschied Text- und Binärdatei

Im Grunde gibt es zwei Arten von Dateien: Textdateien und Binärdateien. Textdateien sind Dateien, die ausschließlich Text enthalten – so weit so offensichtlich. Eine mit einer beliebigen Tabellenkalkulationssoftware erstellte Datei mit Text drin ist aber in der Regel eine Binärdatei. Sie beinhaltet neben dem reinen Text weitere Informationen zur Steuerung der Tabellenkalkulationssoftware. Reiner Softwarecode besteht dagegen meist aus Textdateien.

Binärdateien dienen dagegen nicht zum alleinigen Darstellen von Text, sondern beinhalten unter anderem die bereits oben erwähnten Steuerinformationen. Typische Beispiele für Binärdateien sind Bilder, Musik, komprimierte Archive und ausführbare Programme.<sup>23</sup>

**Tipp:** Wenn du dir nicht ganz sicher bist, ob eine Datei eine Binärdatei ist oder nicht, gibt es einen einfachen Test: Öffne das entsprechende Dokument in einem schlichten Texteditor. Ist das, was du da siehst, überwiegend nicht leserlich, handelt es sich um eine Binärdatei (siehe als Beispiel Abbildung 7-5).



The image shows a screenshot of a terminal window or text editor displaying binary data. The data consists of various non-printable characters, such as underscores, colons, and symbols, which do not form readable text. This visual representation serves as a practical example of what a binary file looks like when opened in a plain text editor.

Abbildung 7-5: Eine Tabellenkalkulationsdatei – in einem Texteditor geöffnet, offenbart sie ihre wahre Natur als Binärdatei.

Sollte ich das nutzen, auch wenn ich nur wenige kleine Binärdateien in meinem Repository habe? Die Antwort lautet: Nein, nicht unbedingt. Wenn du eher wenige kleine und sich selten ändernde Binärdateien hast, kannst du auch gut ohne diese Erweiterung auskommen. Das Einrichten ist allerdings so leicht und die Handhabung genauso wie bei Standard-Git, sodass es sich lohnt, diese Möglichkeit anzusehen.

## Installation von Git LFS

Für die Installation von Git LFS musst du einfach auf die Homepage<sup>24</sup> gehen, das Paket herunterladen und es installieren. Wer unter Linux – beispielsweise Debian oder Ubuntu – unterwegs ist, kann es gegebenenfalls direkt installieren:

```
$ sudo apt-get install git-lfs
```

<sup>23</sup> Gemeinerweise gibt es aber auch Ausnahmen, beispielsweise Scalable Vector Graphics. Diese Bilddateien mit der Endung \*.svg sind in der Regel Textdateien.

<sup>24</sup> <https://git-lfs.github.com/>

Nach der Installation musst du einmalig noch Folgendes ausführen:

```
$ git lfs install
Updated git hooks.
Git LFS initialized.
```

Damit wird Git LFS für deinen aktuellen Account eingerichtet und sogenannte Hooks ebenfalls (siehe hierzu die Erklärbärbox »Git Hooks«). Was hat sich verändert? Im Hook-Verzeichnis deines Git-Repositorys (bei mir heißt es *gittest*), sind vier neue Hooks hinzugekommen:

```
$ ls gittest/.git/hooks/
post-checkout
post-commit
post-merge
pre-push
[...]
```

Diese werden zukünftig vor oder nach den angegebenen Git-Aktivitäten ausgeführt. Hooks, die mit post (Lateinisch für »nach« oder »nachher«) beginnen, werden nach der entsprechenden Aktivität ausgeführt, Hooks mit dem Kürzel pre (Lateinisch »prä« für »vorher«) entsprechend vorher. Beispielsweise wird der Hook *post-commit* nach dem Ausführen eines Commits ausgeführt. Eine Sache müssen wir aber noch tun, damit das Ganze auch funktioniert. Wir müssen Git LFS sagen, worauf es genau achten soll.

## Git Hooks

Git Hooks (deutsch »Haken«) sind programmierte Skripte, die vor oder nach einer Git-Operation (beispielsweise einem Commit) ausgeführt werden. Sie sind im Projektverzeichnis unter *.git/hooks/* zu finden. Standardmäßig gibt es dort bereits einige Dateien mit der Endung *.sample*. Sobald du die Endung von der Datei entfernst, schaltest du den entsprechenden Hook »scharf«.

Hooks können dazu genutzt werden, lästige Arbeiten zu übernehmen oder eine bestimmte Qualität sicherzustellen. Beispielsweise könntest du selbst einen Hook programmieren, der beim Committen überprüft, ob hinter jedem Satz ein Punkt steht, oder einen Hook, der automatisch nach dem Committen irgendwelche Dateien aufräumt.

## Git LFS einrichten

Git LFS richtet man über den Befehl `git lfs track` ein. Dadurch bekommt Git LFS den Hinweis, bei welchen Dateien oder in welchen Verzeichnissen es aktiv werden soll. Zum Beispiel könntest du für dein Projekt einrichten, dass alle Dateien mit der Endung *\*.png* und alle Dateien im Unterordner *images* von Git LFS anders behandelt werden sollen:

```
$ git lfs track *.png
Tracking "*.png"
```

```
$ git lfs track images/
Tracking "images/"
```

Wenn du wissen willst, was genau noch mal von Git LFS getrackt wird, musst du einfach nur denselben Befehl ohne zusätzliche Parameter aufrufen:

```
$ git lfs track
Listing tracked patterns
*.png (.gitattributes)
images (.gitattributes)
```

Git LFS weist uns hierbei auch noch darauf hin, wo diese Informationen gespeichert werden, nämlich in einer Datei namens `.gitattributes`.<sup>25</sup> Diese Datei ist in deinem Arbeitsbereich neu und sollte daher auch in dein Repository eingefügt werden:

```
$ git add .gitattributes
[...]
$ git commit -m "fuege .gitattributes hinzu"
[...]
```



#### Tipp: Repository nachträglich aufräumen

Wenn du ein bestehendes Repository nachträglich von großen Binärdateien »befreien« möchtest, lohnt sich ein Blick auf den *BFG Repo-Cleaner*<sup>26</sup>. Damit lassen sich große Dateien oder auch Passwörter und Ähnliches relativ einfach aus der Git-Historie löschen. Das Kürzel steht übrigens unter anderem für die Superschusswaffe »BFG 9000« aus dem 3-D-Shooter Doom.

Jedes Mal, wenn du jetzt Bilder mit der Endung `.png` oder Dateien im Ordner `images` in dein Git-Repository committest, kümmert sich Git LFS um alles Weitere. Du musst grundsätzlich nichts an deinem Workflow verändern. Und das Beste: GitHub unterstützt Git LFS. Das heißt, selbst wenn du später dein lokales Git-Repository auf GitHub hochlädst, musst du dich um nichts weiter kümmern, das passiert automatisch beim Hochladen durch das Committen der `.gitattributes`. Auf GitHub wird das beim Anwählen der entsprechenden Datei sichtbar, siehe auch Abbildung 7-6.



Abbildung 7-6: Dateien unter Git LFS werden auf GitHub entsprechend markiert.

25 Mit der kann man auch noch andere Sachen anstellen, das geht hier aber zu weit.

26 <https://rtyley.github.io/bfg-repo-cleaner/>



### Achtung: GitHub und Git LFS

Ein paar Einschränkungen gibt es im Zusammenspiel zwischen GitHub und Git LFS:

- Dateien dürfen nicht größer als 2 GByte sein.
- Git LFS funktioniert nicht zusammen mit GitHub Pages (die lernen wir im Abschnitt »Websites aus GitHub generieren (GitHub Pages)« auf Seite 225 in Kapitel 10 noch kennen).

## Sich weiter schlaumachen über Git

So viel zum Thema Git. Um die ersten zarten Schritte mit Git zu machen, hat dieser Einstieg hoffentlich ausgereicht. Wenn du mehr willst, möchte ich dir noch einige Quellen empfehlen: zum einen zwei Bücher und zum anderen einige Ressourcen im Internet – vom kostenlosen Git-Buch bis hin zu kurzweiligen Git-Spielen.

### Oldschool: Bücher

Wenn du intensiver und auch mal ohne Computer in Git eintauchen möchtest, kann ich dir die beiden nachfolgenden Bücher empfehlen:

- **Versionsverwaltung mit Git – Praxiseinstieg** von Sujeevan Vijayakumaran (2019, mitp-Verlag, ISBN 978-3-74750042-2) – Gut für den Einstieg, erklärt sehr schön Schritt für Schritt, was zu tun ist.
- **Git – Dezentrale Versionsverwaltung im Team, Grundlagen und Workflows** von René Preißel und Björn Stachmann (2019, dpunkt.verlag, ISBN 978-3-86490-649-7) – Eher für Fortgeschrittene geeignet. Größter Pluspunkt ist die Darstellung verschiedener Git-Workflows.

### Neumodischer Kram: Internet

Natürlich gibt es im Internet eine ganze Menge Ressourcen, um tiefer in Git einzusteigen. Da ich unmöglich alle Blogs, Video- und Techseiten aufzählen kann und eine solche Liste sehr lang und vermutlich auch relativ schnell veraltet sein dürfte, fokussiere ich mich auf ein paar wenige.

- **Pro Git** ist ein kostenloses Buch, das direkt auf der Git-Homepage zur Verfügung steht.<sup>27</sup> Du kannst die englische oder auch eine deutsche Version als PDF, ePub oder MOBI herunterladen oder auch direkt im Browser lesen. Ist dir totes Holz in deinen Händen lieber? Das Buch ist auch bestellbar.
- **Git Reference** ist eine vom GitHub-Team zusammengestellte Referenz<sup>28</sup> zu den wichtigsten Befehlen von Git.

27 <https://git-scm.com/book/de/v2>

28 <https://git.github.io/git-reference/>

- Falls du Interesse an grafischen Oberflächen für Git hast, lohnt sich ein Blick auf die Git-Website.<sup>29</sup>
- **Learn git-branching** ist eine interaktive Website<sup>30</sup> (siehe auch Abbildung 7-7), auf der du dich spielerisch mit dem Thema Git beschäftigen kannst. Aufgebaut ist das Ganze über mehrere Level, wobei zu jedem Level am Anfang eine kleine Einweisung bereitsteht, die erklärt, was du jetzt genau tust und warum. Sehr empfehlenswert!
- Ein weiteres Spiel ist das **Git-Game**<sup>31</sup>, das über die Konsole gespielt wird. Es ist eine Schnitzeljagd auf der Suche nach Hinweisen. Mit Git-Befehlen musst du das nächste Puzzlestück finden, z.B. indem du auf den richtigen Branch wechselst. Toll gemacht! Du solltest aber schon etwas vertraut sein mit der Konsole und den Git-Befehlen.
- Eine etwas ungewöhnliche Darstellung bietet **git concepts simplified**<sup>32</sup>. Es handelt sich dabei um eine Präsentation (*Slide Show*) im Browser, in der einige Git-Konzepte mit vielen Bildern dargestellt werden. Tipp: den rechten oder linken Rand anklicken, um zwischen den einzelnen »Folien« hin- und herzuspringen.

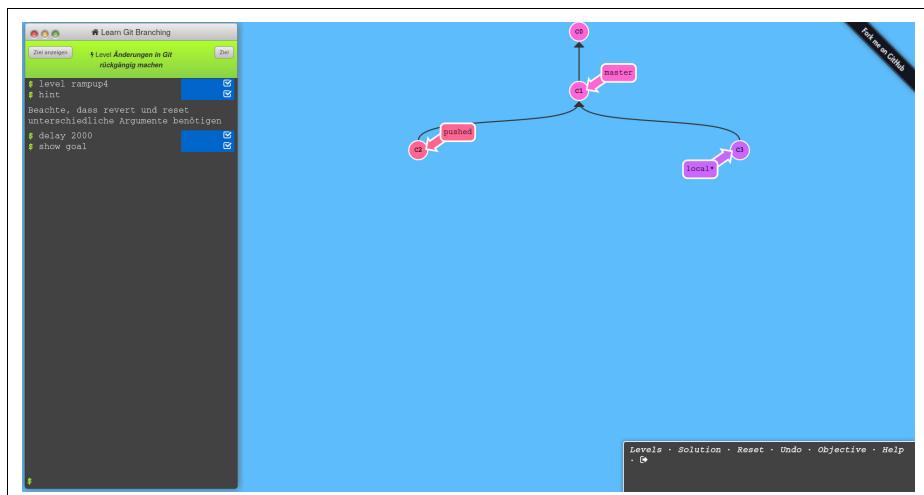


Abbildung 7-7: Auf der Website »Learn git-branching« lernt man spielerisch Git. Die Eingabe ist übrigens unten links, und ein Level zurücksetzen kann man unten rechts.

Als Nächstes schauen wir uns im folgenden Kapitel an, wie Git und GitHub zusammen wirken und was man damit alles anstellen kann.

29 <https://git-scm.com/downloads/guis/>

30 <https://learngitbranching.js.org/>

31 <https://github.com/git-game/git-game>

32 [https://gitolite.com/gcs.html#\(1\)](https://gitolite.com/gcs.html#(1))



## KAPITEL 8

# Git und GitHub im Zusammenspiel

In Kapitel 4 im Abschnitt »Ein bestehendes Projekt hochladen« auf Seite 40 haben wir bereits gesehen, wie du lokale Dateien direkt über den Webbrower auf GitHub hochladen kannst. Und das vorangegangene Kapitel 7 hat dir gezeigt, wie du mit Git Dateien lokal verwaltetest. In diesem Kapitel werden wir sehen, wie GitHub und Git zusammenwirken, um noch mehr aus GitHub herauszuholen. Dafür werden wir unter anderem lokal verwaltete Git-Repositories auf GitHub über die Konsole hochladen.



### Am Ende des Kapitels kannst du ...

- ein lokales Git-Projekt auf GitHub überführen und Änderungen im lokalen Projekt auf GitHub synchron halten.
- ein bestehendes eigenes GitHub-Repository auf den heimischen Rechner holen und Anpassungen auf GitHub im lokalen Projekt synchron halten.
- ein fremdes Repository forken und zu diesem synchron bleiben.
- ein fremdes Repository forken und Vorschläge zu Anpassungen an das Original-Repository zurückspielen.
- Merge-Konflikte auf GitHub und lokal lösen.
- die Zugangsdaten für dein GitHub-Repository während der Nutzung von Git zwischen speichern.

Wir erweitern unser Verständnis über den Aufbau von Git um das *Repository (remote)* (zu Deutsch etwa »fernes Repository«, siehe Abbildung 8-1). Damit sind Repositories gemeint wie unseres auf GitHub. Hier könnten jetzt aber auch beliebige andere ferne Repositories stehen, beispielsweise GitLab oder Bitbucket. Und es ist sogar möglich, ein anderes Verzeichnis auf dem heimischen Rechner als Remote-Repository festzulegen, beispielsweise zu Übungszwecken oder wenn der eigene Rechner als Git-Server im Einsatz ist. Darüber hinaus kann ein lokales Git-Repository sogar mit mehreren Remote-Repositories verknüpft sein. Einen solchen Anwendungsfall wirst du in diesem Kapitel auch noch kennenlernen.

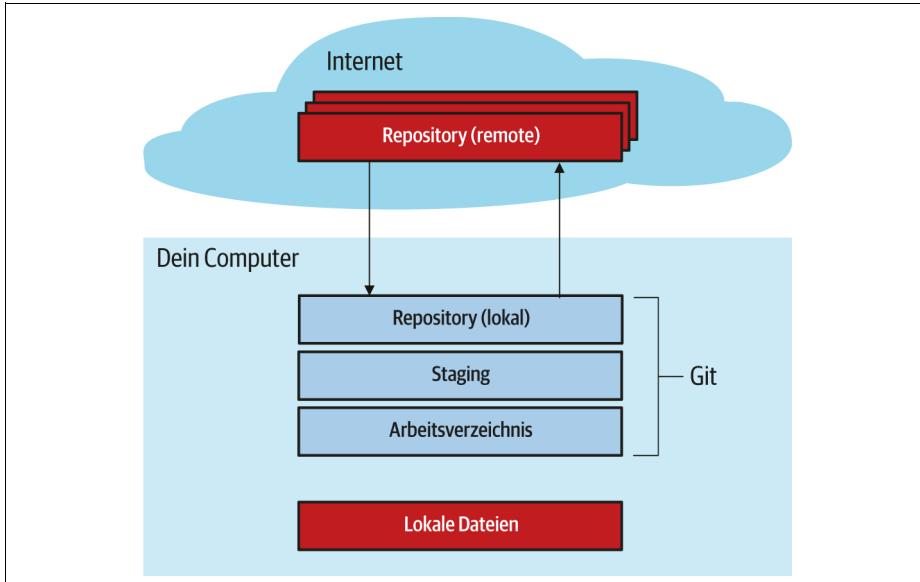


Abbildung 8-1: Git kennt auch ferne Repositories.

Wir werden für das Zusammenspiel von Git und GitHub vier verschiedene Szenarien beleuchten (siehe Abbildung 8-2), die das ganze Spektrum der Zusammenarbeit abdecken. Wir bauen dabei jedes Szenario von Grund auf neu auf, tun also so, als wäre es jedes Mal ein ganz neues Projekt (mit Ausnahme von Szenario 4, das bauen wir auf Szenario 3 auf). Wir beleuchten, wie das entsprechende Szenario konfiguriert wird und wie darin konkret gearbeitet wird. RepoA ist dabei immer ein eigenes Repository – also eines, auf das wir vollen Zugriff haben –, RepoX ist dagegen stets ein fremdes Repository, auf das wir keinen vollen Zugriff haben.

1. **Szenario 1:** Wir überführen ein lokales Git-Projekt auf GitHub (konfigurieren) und schauen uns an, wie wir Änderungen von lokal zu remote bekommen (arbeiten).
2. **Szenario 2:** Wir nehmen ein bestehendes eigenes GitHub-Repository und holen es uns auf den heimischen Rechner (konfigurieren). Dann schauen wir, wie wir Anpassungen auf GitHub auch lokal vorliegen haben (arbeiten).
3. **Szenario 3:** Wir forken ein fremdes Repository und schauen, wie wir es schaffen, mit diesem fremden Repository synchron zu bleiben (konfigurieren + arbeiten).
4. **Szenario 4:** Wir erweitern Szenario 3, indem wir Anpassungen an dem geforkten Repository vornehmen und versuchen, diese an das Original-Repository zurückzuspielen.

Mit dem Kennenlernen der vier Szenarien sind wir dann auch handlungsfähig, um uns mit dem Thema »Merge-Konflikte« beschäftigen zu können. Ich werde dir später zwei Wege zeigen, wie du einen Merge-Konflikt lösen kannst. Legen wir los!

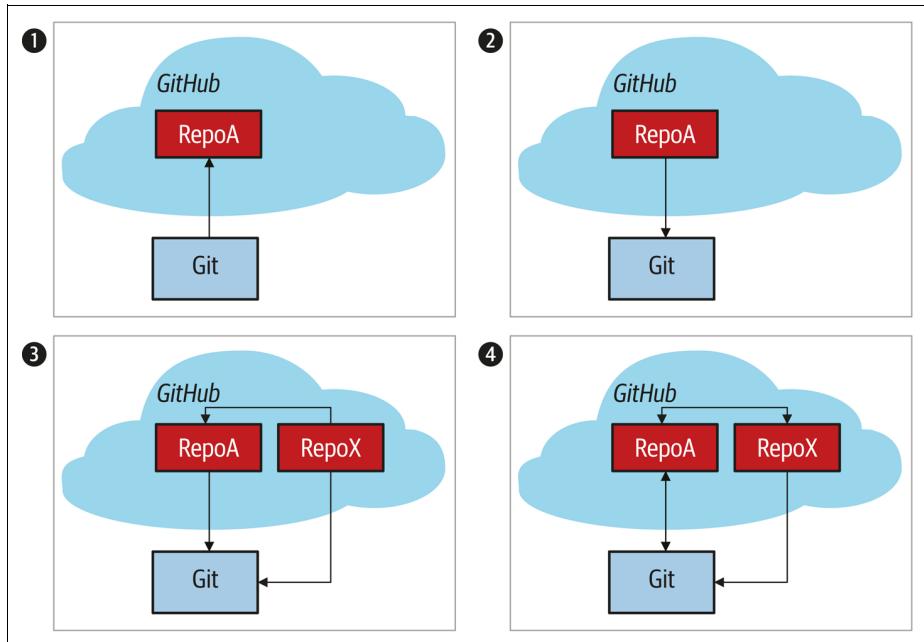


Abbildung 8-2: Diese vier Szenarien werden in diesem Kapitel betrachtet – RepoA ist ein eigenes, RepoX ein fremdes Repository.

## Szenario 1: Lokales Git-Projekt auf GitHub hochladen

Wir überführen ein lokales Git-Projekt auf GitHub und schauen uns an, wie wir Änderungen von lokal zu remote bekommen (siehe auch Abbildung 8-3).

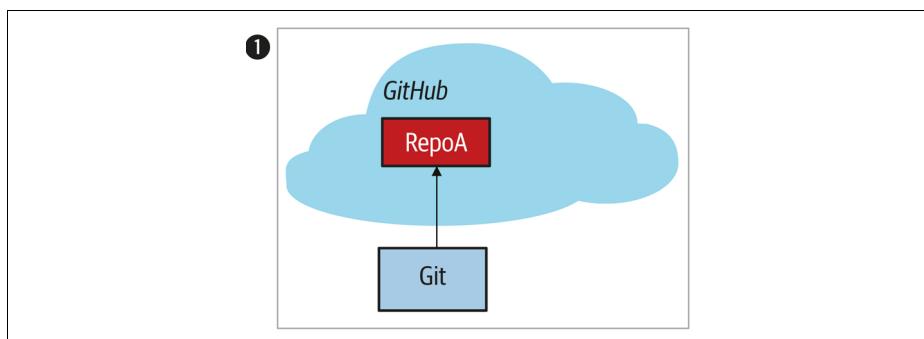


Abbildung 8-3: Szenario 1: Lokales Git-Projekt auf GitHub hochladen

Um das zu erreichen, werden wir folgende Schritte nacheinander durchführen:

1. Lokal ein Git-Repository mit einer Datei anlegen.
2. Leeres Repository auf GitHub anlegen.

3. Das Git-Repository mit dem GitHub-Repository verknüpfen.
4. Git-Repository auf GitHub hochladen (pushen).
5. Lokal Änderungen vornehmen und diese auf GitHub hochladen.

## Lokal ein Git-Repository mit einer Datei anlegen

Falls noch nicht geschehen, erstelle lokal ein Verzeichnis und erzeuge dort die Datei *README.md*. Wenn du magst, kannst du die Datei auch schon befüllen, beispielsweise mit ein paar Überschriften:

```
# Mein tolles Projekt  
## Unterpunkt 1  
## Unterpunkt 2
```

Initialisiere das Verzeichnis als Git-Repository und committe die *README.md*. Du hast also folgende Befehle so oder so ähnlich durchgeführt:

```
$ mkdir VERZEICHNIS  
$ cd VERZEICHNIS  
$ git init  
$ touch README.md  
$ git add README.md  
$ git commit -m "initialisiere Repo"
```



### Tipp: Datei erzeugen und gleichzeitig befüllen

Es gibt einen einfachen Weg, eine Datei auf der Konsole zu erzeugen und gleichzeitig diese Datei initial zu befüllen, und zwar diesen:

```
$ echo "# Mein tolles Projekt" >> README.md
```

Der Befehl `echo` gibt normalerweise das, was in Hochkommata folgt, einfach auf der Konsole aus. Durch die beiden `>>` wird aber diese Ausgabe in die Datei *README.md* umgeleitet. Sofern die Datei noch nicht existiert, wird sie erzeugt.

## Leeres Repository auf GitHub anlegen

Wir wechseln jetzt auf die GitHub-Seite und erstellen ein GitHub-Repository mit einem für dich passenden Namen (wie das geht, wurde im Abschnitt »Das erste eigene Repository anlegen« auf Seite 24 bereits erklärt). Beim Anlegen des neuen Repositorys solltest du aber darauf achten, dass GitHub keine Dateien wie *README.md*, *.gitignore* oder *LICENSE* für dich anlegt.<sup>1</sup> Warum? Wenn wir schon erste Dateien auf GitHub angelegt haben, ist es komplizierter, das lokale Git-Projekt hochzuladen. Dann müssten wir beide erst synchronisieren, was ein paar Handgriffe extra erfordert. Das ist unnötige Arbeit, und die wollen wir uns daher sparen.

---

<sup>1</sup> Standardmäßig werden die Dateien nicht angelegt, du musst schon explizit sagen, dass du sie willst.

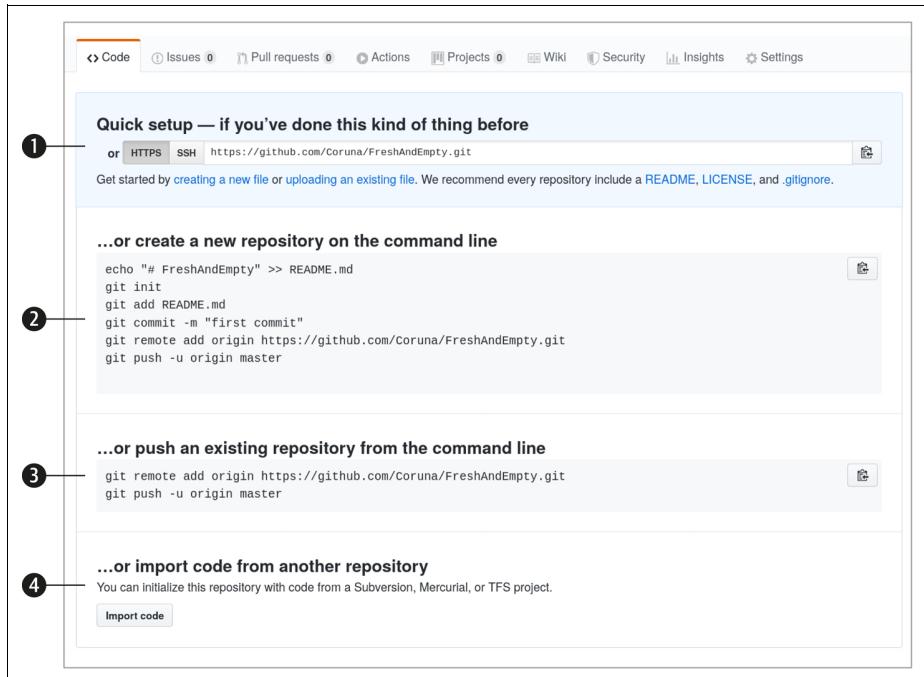


Abbildung 8-4: Nach dem Anlegen eines leeren Repositorys gibt GitHub hilfreiche Informationen darüber, was als Nächstes getan werden kann.

Nach dem Anlegen des Repositorys gibt uns GitHub wertvolle Hinweise dazu, welche Schritte als Nächstes sinnvoll wären (siehe Abbildung 8-4, 1–4). Wenn du dir die zweite Option *...or create a new repository on the command line* (deutsch etwa »... oder erzeuge ein neues Repository von der Konsole«) genauer anschaugst, wirst du sehen, dass die ersten vier Befehle genau das sind, was wir im Abschnitt vorher gemacht haben: Datei anlegen, Git-Repo initialisieren, Datei ins Repo über den Staging-Bereich committen. Lediglich die unteren beiden Befehle kennen wir noch nicht. Diese sind auch bei der dritten Option *...or push an existing repository from the command line* (deutsch etwa »... oder pushe ein bereits existierendes Repository von der Konsole«) aufgeführt. Wir schauen sie uns in den nächsten beiden Abschnitten genauer an.

## Das Git- mit dem GitHub-Repository verknüpfen

Das Erste, was wir machen müssen, um Git und GitHub zu verknüpfen, ist, Git zu sagen, wo genau das GitHub-Repo liegt. Dafür gibt es den Befehl `remote`, der alle fernen Repositories, die zu einem Git-Projekt gehören, verwaltet. Mit dem Zusatz `add` fügen wir ein neues fernes Repository hinzu, was in unserem Fall unser gerade erstelltes GitHub-Repository sein wird. Das könnte dann wie folgt aussehen:

```
$ git remote add origin https://github.com/USERNAME/REPONAME.git
```

Das Wort `origin` ist dabei einfach ein Kürzel – auch Alias<sup>2</sup> genannt – für die Verknüpfung mit unserem fernen Repository. Damit können wir später einfach nur den Alias schreiben und müssen nicht immer die lange Adresse eintippen. Wenn du möchtest, kannst du auch einen anderen Alias verwenden, beispielsweise:

```
$ git remote add github https://github.com/USERNAME/REPONAME.git
```

Der Alias `origin` hat sich als Standard etabliert, sodass viele User ihn exakt gleich einrichten und du ihn häufig in Tutorials finden wirst. Ich werde ihn hier daher auch weiter verwenden. Das `https://github.com/USERNAME/REPONAME.git` ist die Adresse unseres GitHub-Repositorys, die wir uns später mit dem Alias ersparen wollen. Im vorherigen Abschnitt hat GitHub uns diese bereits präsentiert (siehe ❶ in Abbildung 8-4). Ansonsten bekommst du sie, wenn du in deinem GitHub-Repository `Code` auswählst (siehe hierzu auch Abbildung 8-5).

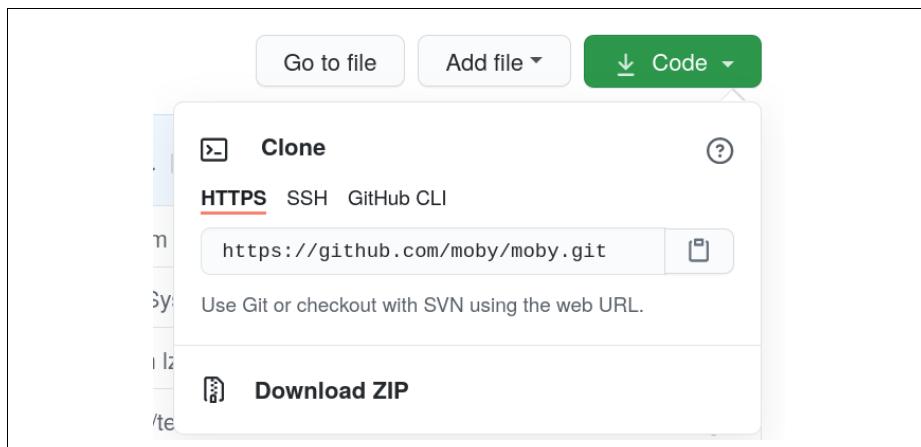


Abbildung 8-5: Die URL eines GitHub-Repos wird über den Code-Button angezeigt.



#### Tipp: GitHub-URL vom Repository direkt kopieren

Um Tippfehler zu vermeiden, ist es meist besser, die URL von GitHub zu kopieren und einzufügen, als sie manuell einzutippen.

Sobald du das neue Remote-Repository bei Git bekannt gemacht hast, überprüfen wir, ob alles seine Ordnung hat:

```
$ git remote -v
origin https://github.com/username/reponame.git (fetch)
origin https://github.com/username/reponame.git (push)
```

<sup>2</sup> Ein Alias (Lateinisch für »sonst«) hat die Bedeutung »mit anderem Namen« und wird oft als Vereinfachung komplizierter Befehle genutzt.

Dieser Befehl listet dir auf, welche fernen Quellen dein lokales Repository derzeit kennt. Der Schalter `-v` (für *verbose*, zu Deutsch etwa »wortreich«) sorgt dafür, dass wir mehr hilfreiche Informationen bekommen. Bei dir sollten das wie bei mir zwei Einträge zu ein und demselben Remote-Ort sein, den du gerade angegeben hast. Der erste Eintrag bedeutet, dass wir uns von dort Informationen holen können (*fetch*, deutsch »holen«), und der zweite Eintrag sagt uns, dass wir dort auch Informationen »hinschieben« können (*push*, deutsch »schieben«).

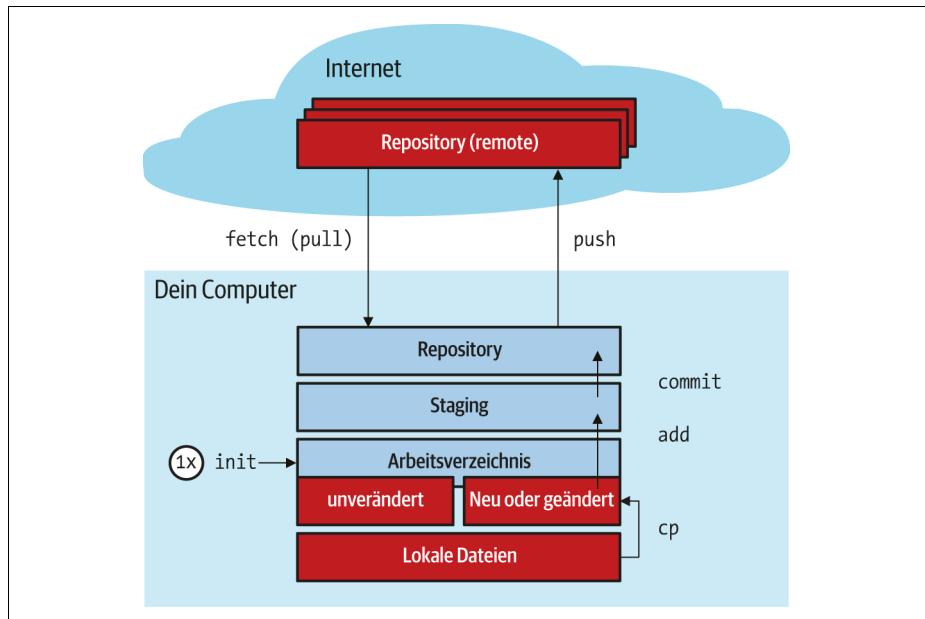


Abbildung 8-6: Um mit einem fernen Repository zu kommunizieren, gibt es die Befehle `fetch` und `push`.

In Abbildung 8-6 siehst du die Befehle, die du für die Kommunikation mit dem Remote-Repository benötigst. Bei `fetch` steht noch ein zweiter Eintrag, nämlich `pull`. Der Befehl `pull` holt sich die Änderungen vom Remote-Repository und mergt diese auch sofort mit deinem lokalen Repository. Bei `fetch` werden zwar ebenfalls die Änderungen geholt, aber du musst das Mergen im Nachgang noch manuell anstreßen. Letzteres hat den Vorteil, dass du mehr Kontrolle darüber hast, was lokal passiert. Es wird häufig empfohlen, lieber mit `fetch` und `merge` zu arbeiten als mit `pull`.<sup>3</sup> Ich folge daher diesen Empfehlungen.



**Unterscheidung zwischen pull und fetch bei Git**  
`pull` = `fetch + merge`

<sup>3</sup> Hier z. B.: <https://longair.net/blog/2009/04/16/git-fetch-and-merge/>.

Falls du noch mehr Details zu deinem Remote-Repository benötigst, ist folgender Befehl hilfreich:

```
$ git remote show origin
* Remote-Repository origin
  URL zum Abholen: https://github.com/username/reponame.git
  URL zum Versenden: https://github.com/username/reponame.git
  Hauptbranch: (unbekannt)
```

Falls du einen anderen Alias als origin gewählt hast, musst du ihn hier stattdessen verwenden. Spannend ist diese Rückmeldung allerdings im Moment noch nicht, da wir ja bereits durch `git remote -v` wissen, welches unsere pull- und push-Adressen sind. Das ändert sich gleich im nächsten Abschnitt.

## Git-Repository auf GitHub hochladen (pushen)

Dein lokales Git-Repo weiß jetzt also, wo dein fernes GitHub-Repo liegt. Jetzt müssen wir das lokale Git-Repo nur noch in das GitHub-Repo kopieren. Das wird auch *pushen* genannt und funktioniert grundsätzlich wie folgt:

```
$ git push REMOTE_REPO LOKALER_BRANCH
```

Mit `REMOTE_REPO` ist offensichtlich unser Remote-Repository und mit `LOKALER_BRANCH` der lokale Branch gemeint. In unserem Fall wollen wir zu dem Remote-Repo pushen, dem wir weiter oben bereits den Alias `origin` verpasst haben. Es wäre aber auch möglich, hier die URL des Remote-Repositorys einzugeben. Als lokalen Branch wählen wir den `master`-Branch aus, das sieht dann wie folgt aus:

```
$ git push -u origin master
[...]
To https://github.com/username/reponame.git
 * [new branch]      master -> master
Branch 'master' folgt nun Remote-Branch 'master' von 'origin'.
```



### Tipp: Mailadresse (noch mehr) schützen

Du kannst GitHub so einrichten, dass ein Push vonseiten Git geblockt wird, wenn du bei Git eine private Mailadresse konfigurierst. Die Option ist in deinem Profil unter dem Punkt *Settings/EMails* zu finden und heißt *Block command line pushes that expose my email* (deutsch etwa »Blockieren von Pushs von der Konsole, die meine E-Mail offenlegen«, siehe auch Abbildung 3-5).



### Achtung: Konsole und 2FA

Falls die 2-Faktor-Authentifizierung aktiviert ist (siehe Abschnitt »Account schützen« auf Seite 22 in Kapitel 3), benötigst du für den Zugriff über die Konsole auf ein GitHub-Repository einen sogenannten Personal Access Token. Diesen kannst du über *Settings* und dann *Developer Settings* einrichten.<sup>4</sup> Wenn du dich über die Konsole bei GitHub anmeldest, verwendest du dann den Token anstelle des GitHub-Passworts.

<sup>4</sup> Siehe auch <https://docs.github.com/en/github/authenticating-to-github/creating-a-personal-access-token>.

Der Parameter `-u` sorgt dafür, dass eine Verknüpfung vom lokalen mit dem remote Branch erzeugt wird. Das muss nur einmal gemacht werden, es sei denn, du erzeugst neue Branches. Hier wirst du deine Zugangsdaten für GitHub eingeben müssen.

Im Abschnitt »Log-in-Erleichterungen bei HTTPS« auf Seite 194 weiter unten zeige ich dir, wie du das vereinfachen kannst, damit du nicht ständig dein Passwort eingeben musst. Sollte es den angegebenen lokalen Branch (hier: `master`) im Remote-Repository nicht geben, wird er übrigens angelegt. Schauen wir uns jetzt noch einmal die Details zu unserem Remote-Repository an, bekommen wir eine etwas andere Rückmeldung:

```
$ git remote show origin
* Remote-Repository origin
  URL zum Abholen: https://github.com/username/reponame.git
  URL zum Versenden: https://github.com/username/reponame.git
  Hauptbranch: master
  Remote-Branch:
    master gefolgt
    Lokaler Branch konfiguriert für 'git pull':
      master führt mit Remote-Branch master zusammen
    Lokale Referenz konfiguriert für 'git push':
      master versendet nach master (aktuell)
```

Insbesondere die letzten vier Zeilen sind interessant. Durch den Parameter `-u` haben wir dafür gesorgt, dass sowohl für `git pull` als auch für `git push` eine Verknüpfung des lokalen mit dem remote Branch erfolgt ist. Hätten wir den Parameter weggelassen:

```
$ git push origin master
```

sähe die Konfiguration so aus:

```
$ git remote show origin
[...]
  Lokale Referenz konfiguriert für 'git push':
    master versendet nach master (aktuell)
```

Es wäre eine Verknüpfung für `git push` erstellt worden, aber nicht für `git pull`. Falls du von dem lokalen Git-Repo nur pushen möchtest, macht das keinen Unterschied. Möchtest du aber auch pullen, solltest du sie einrichten, da sie das spätere Arbeiten vereinfacht.

Wenn du zurück zu GitHub wechselst, kannst du dir dort das Ergebnis anschauen. Deine zuvor in Git vorhandenen Dateien sollten jetzt auf GitHub zu finden sein. Das war der Konfigurationsteil, um eine Verknüpfung von Git mit GitHub einzurichten. Im nächsten Abschnitt schauen wir uns an, wie du damit an deinem Projekt arbeiten kannst.

## Lokal Änderungen vornehmen und diese auf GitHub hochladen

Mit Git auf GitHub-Repositories zuzugreifen und dort Änderungen vorzunehmen, ist denkbar einfach. Du editierst die Dateien wie gewünscht und überführst sie über den Staging-Bereich in das lokale Repository, hier einmal exemplarisch:

```
[...Dateien ändern...]  
$ git add DATEIEN  
$ git commit -m "NACHRICHT"
```



### Commit-Nachrichten sind Pflicht

Wer einen Commit macht, muss bei Git eine entsprechende Commit-Nachricht angeben. Wird diese vergessen, beispielsweise beim Aufruf von `git commit` ohne entsprechende weitere Parameter, öffnet Git den eingerichteten Standardeditor und verlangt Nachbesserung. Dieser Standardeditor lässt sich auf die eigenen Vorlieben anpassen (siehe Abschnitt »Git installieren und einrichten« auf Seite 141).

Per `git push` überträgst du diese Commits aus deinem lokalen Repository in dein remote-Repository:

```
$ git push
```

Da du vorhin mit `-u` eine Verknüpfung der lokalen und remote Branches durchgeführt hast, brauchst du nicht mehr `origin master` einzugeben. Deine Änderungen sollten danach auf GitHub sichtbar sein.

Lass uns zu Übungszwecken einmal Folgendes durchgehen: Führe einen Commit auf deinem lokalen Repository aus – wie oben beschrieben, aber ohne `push`. Über den Befehl `git status` kannst du dir die Informationen dazu holen, wie es nach dem Commit um das lokale Repository bestellt ist:

```
$ git status  
Auf Branch master  
Ihr Branch ist 1 Commit vor 'origin/master'.  
(benutzen Sie "git push", um lokale Commits zu publizieren)
```

nichts zu committen, Arbeitsverzeichnis unverändert

Bei dir sollte das so ähnlich aussehen – je nachdem, was und wie viel du geändert hast. Der Befehl `git status` stellt dir also den Status deines Arbeitsverzeichnisses und des Staging-Bereichs dar. Jetzt pushe mit `git push` das Ganze auf GitHub und führe dann noch mal einen `git status` aus:

```
$ git status  
Auf Branch master  
Ihr Branch ist auf demselben Stand wie 'origin/master'.  
  
nichts zu committen, Arbeitsverzeichnis unverändert
```

Und das war es auch schon. Jetzt hast du dein Projekt von der lokalen Festplatte in die Verwaltung von Git überführt und von da aus auf GitHub veröffentlicht. Als

weiteren Schritt hast du lokal eine Änderung durchgeführt, die du danach wieder auf GitHub veröffentlicht hast.

## Standardarbeitsabläufe für Szenario 1

Ein Git-Repository richtest du so ein:

```
$ cd VERZEICHNIS  
$ git init  
$ git add DATEI  
$ git commit -m "NACHRICHT"
```

Das Git-Repository verknüpfst du mit einem GitHub-Repository so:

```
[...leeres GitHub-Repository anlegen...]  
$ git remote add origin https://github.com/USERNAME/REPONAME.git  
$ git push -u origin master
```

Änderungen am Git-Repository bringst du so ins GitHub-Repository:

```
[...Dateien lokal ändern...]  
$ git add DATEI  
$ git commit -m "NACHRICHT"  
$ git push
```

## Szenario 2: Projekt auf GitHub lokal zu Git holen

Wir nehmen ein bestehendes eigenes GitHub-Repository und holen es uns auf den heimischen Rechner. Dann schauen wir, wie wir Anpassungen auf GitHub auch lokal vorliegen haben (siehe Abbildung 8-7).

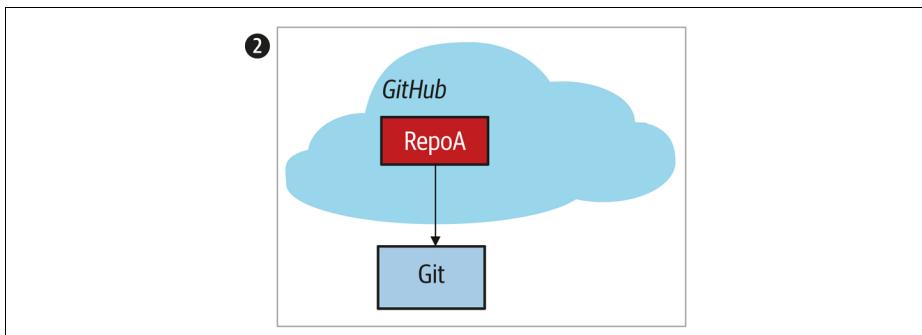


Abbildung 8-7: Szenario 2: Projekt auf GitHub lokal zu Git holen

Wir wollen ein Projekt aus deinem GitHub-Account mithilfe von Git lokal auf deine Platte rüberkopieren. Dieses Projekt kann entweder etwas von dir neu Erschaffenes sein oder aber auch ein Fork eines anderen Projekts (Forks hast du im Abschnitt »Fremdes Projekt unterstützen – Fork« auf Seite 126 in Kapitel 6 ken-

nengelernt). Für die weiteren Schritte ist eine Unterscheidung unerheblich. Wir werden in diesem Abschnitt Folgendes machen:

1. Ein neues GitHub-Repository mit einer Datei erstellen.
2. Das GitHub-Repo mittels Git lokal klonen.
3. Das GitHub-Repo anpassen und die Änderung in die lokale Git-Arbeitsumgebung holen.

## Ein neues GitHub-Repository mit einer Datei erstellen

Kurz und schmerzlos: Wir starten auf GitHub. Erstelle dort ein neues Repository und erzeuge und speichere die Datei *README.md*, die gern auch leer sein darf. Du kannst natürlich auch ein bestehendes Repository für die weiteren Schritte nutzen.

## Das GitHub-Repo mittels Git lokal klonen

Jetzt wechsle auf deine Konsole und klonen (kopiere) dein GitHub-Repository wie folgt:

```
$ git clone https://github.com/USERNAME/REPONAME.git  
Klonen nach 'repename' ...  
remote: Enumerating objects: 3, done.  
remote: Counting objects: 100% (3/3), done.  
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0  
Empfange Objekte: 100% (3/3), Fertig.
```

An der Stelle von *USERNAME/REPONAME* stehen natürlich wieder dein Name und dein Reponame.<sup>5</sup> Bei dir sollte die Rückmeldung in der Konsole so oder so ähnlich aussehen wie in diesem Beispiel. GitHub hat jetzt ein Verzeichnis mit dem Namen deines Repositorys angelegt. Gehe in das Verzeichnis und führe `git status` aus:

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

Dadurch, dass `git status` uns eine solche Rückmeldung gibt, wissen wir, dass in diesem Verzeichnis bereits ein Git-Repo initialisiert worden ist.<sup>6</sup> Es ist also nicht nötig, dass wir das Verzeichnis vorher mit `git init` initialisieren. Falls du `git status` in einem Verzeichnis ausführst, das nicht unter der Verwaltung von Git ist, würdest du stattdessen nämlich folgende Meldung bekommen:

```
$ git status  
fatal: Kein Git-  
Repository (oder irgendein Elternverzeichnis bis zum Einhängepunkt '/')  
Stoppe bei Dateisystemgrenze (GIT_DISCOVERY_ACROSS_FILESYSTEM nicht gesetzt).
```

5 In Abbildung 8-5 kannst du sehen, woher du die Adresse bekommst.

6 Alternativ könntest du auch schauen, ob es dort das Verzeichnis `.git` gibt.



### Geklontes Repository

Ein geklontes Repository muss nicht mehr mit git init initialisiert werden.

Worum wir uns ebenfalls nicht mehr kümmern müssen, ist das Anlegen des Remote-Repositorys, wie uns eine kurze Überprüfung verrät:

```
$ git remote -v
origin https://github.com/username/reponame.git (fetch)
origin https://github.com/username/reponame.git (push)
```

Und auch unser einziger (master-)Branch ist bereits lokal mit dem Remote-Repository verknüpft:

```
git remote show origin
[...]
Lokaler Branch konfiguriert für 'git pull':
  master führt mit Remote-Branch master zusammen
Lokale Referenz konfiguriert für 'git push':
  master versendet nach master (aktuell)
```

Die Einrichtung ist damit abgeschlossen – und alles nur mit einem einzigen Befehl: git clone. Wenn du es dir also in Zukunft einfach machen willst, würde ich dir diesen Weg empfehlen.

## Das GitHub-Repo anpassen und die Änderung in die lokale Git-Arbeitsumgebung holen

Wir haben jetzt dein GitHub-Repo lokal auf deinen Rechner geklont. Wechselst du nun zurück zu GitHub, kannst du dem Repository einen neuen Commit hinzufügen (z.B. durch das Anlegen einer neuen Datei oder das Editieren der *README.md*). Wenn du dann wieder zurück auf die Konsole wechselst und git status eingibst, hat sich – offensichtlich nichts verändert?

```
$ git status
Auf Branch master
Ihr Branch ist auf demselben Stand wie 'origin/master'.
nichts zu committen, Arbeitsverzeichnis unverändert
```



### Überwachung der Remote-Repositories

Git überwacht nicht automatisch Veränderungen aufseiten der Remote-Repositories. Das musst du immer manuell machen.

Mit origin/master ist hier der master-Branch auf unserem Remote-Repository origin gemeint, den wir ja gerade in GitHub geändert haben. Wo ist der Fehler? Hinweise hierzu finden sich in Abbildung 8-8.

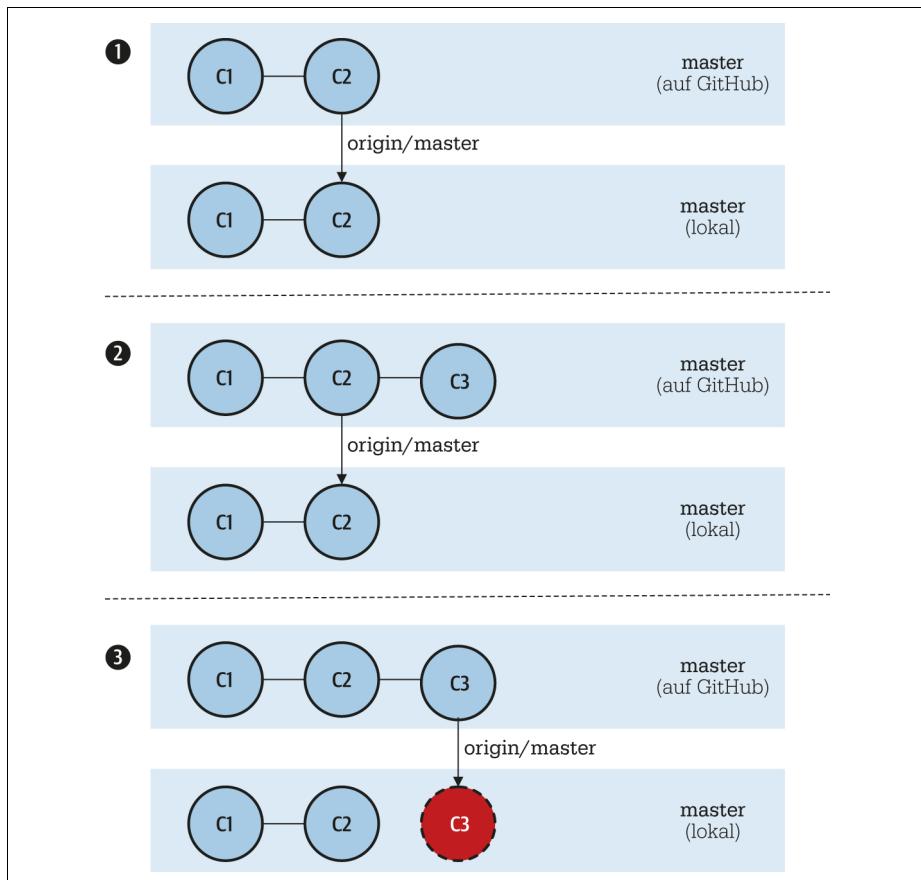


Abbildung 8-8: ❶ Ausgangssituation nach dem Klonen, ❷ Situation nach einem Commit auf GitHub, ❸ Situation nach dem manuellen Aktualisieren (beispielsweise über git fetch)

Ausgangssituation ❶: Wir haben zwei Branches master, einmal auf GitHub und einmal lokal, die wir bereits miteinander verknüpft haben (ausgedrückt durch origin/master, den man auch den Remote Tracking Branch nennt). Das ist der Zustand, den wir direkt nach dem Klonen des GitHub-Repositorys hatten. Nach deinem Commit auf den master-Branch in GitHub ist Situation ❷ entstanden, in der wir uns gerade befinden. Unser »Zeiger« origin/master hat sich nicht bewegt – trotz des Commits auf GitHub. Folgerichtig sagt uns git status auch, dass es keine Veränderungen von dem lokalen Repo zu diesem Zeiger gegeben hat.

Ähnlich wie beim Forken (siehe Abschnitt »Fremdes Projekt unterstützen – Fork« auf Seite 126 in Kapitel 6) schaut unser Klon nicht permanent und von selbst, ob es

remote irgendwelche Veränderungen gegeben hat. Das müssen wir schon selbst erledigen, um Situation ❸ herbeizuführen. Hier ist unser Zeiger aktualisiert und weiß jetzt, dass es remote einen weiteren Commit gegeben hat. Würden wir in dieser Situation git status anwenden, würden wir über die Veränderungen informiert werden (angedeutet durch Commit3 im lokalen Repository mit dem gestrichelten Rand). Eine Möglichkeit, den Zeiger zu aktualisieren, ist git fetch, das wir bereits kennengelernt haben. Das probieren wir gleich mal aus:

```
$ git fetch
[...]
Von /home/USERNAME/REPONAME
a70d14c..3480a5f master    -> origin/master
```



#### Achtung: Remote Tracking Branch

Auch wenn der Remote Tracking Branch ein Branch ist und du grundsätzlich via git checkout auf ihn wechseln kannst, ist er nicht dafür gedacht. Tust du es trotzdem, kommst du in den Modus *detached HEAD* (siehe Erklärbarbox »*detached HEAD – ein losgelöster Kopf*«).

### detached HEAD – ein losgelöster Kopf

Wenn du dich intensiver mit Git beschäftigst, wirst du früher oder später über den Begriff *HEAD* stolpern. *HEAD* ist ein Zeiger (*Pointer*), der immer auf den letzten Commit im aktuellen Branch zeigt, auf dem du gerade arbeitest bzw. den du gerade ausgecheckt hast (in Abbildung 8-9 ist es beispielsweise der master-Branch). Dieser Zeiger erlaubt es Git, immer zu wissen, auf welchem Branch du dich aktuell befindest. Wechselst du den Branch, wechselt *HEAD* ebenfalls hin zu diesem Branch. Im Bild würde bei einem Wechsel auf den Feature-Branch der Pfeil dann auf den Feature-Branch zeigen.

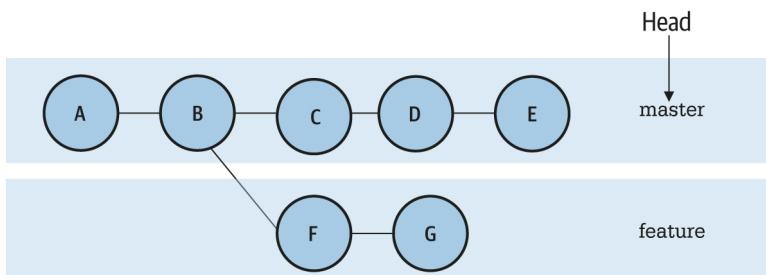


Abbildung 8-9: HEAD zeigt immer auf den aktuellen Branch.

Es ist aber auch möglich, HEAD auf etwas zeigen zu lassen, das kein regulärer Branch ist. Das kann beim Wechseln auf einen Remote Tracking Branch oder auf einen einzelnen Commit passieren, beispielsweise über:

```
$ git checkout 4d13b24
```

Das erzeugt den sogenannten *detached Head* (deutsch »losgelöster Kopf«) und sieht dann wie in Abbildung 8-10 aus. Sinnvolle Anwendungsfälle für ein solches Vorgehen sind eher selten, ein Beispiel sind Submodule<sup>7</sup>. In der Regel entsteht der Modus *detached HEAD* ungewollt.

Wenn du jetzt im Modus *detached HEAD* committest, gehören diese Änderungen zu keinem Branch. Solltest du nicht wissen, was du tust, könntest du diese Commits (bzw. die darin enthaltenen Änderungen) verlieren, wenn du danach auf einen anderen Branch wechselst.<sup>8</sup>

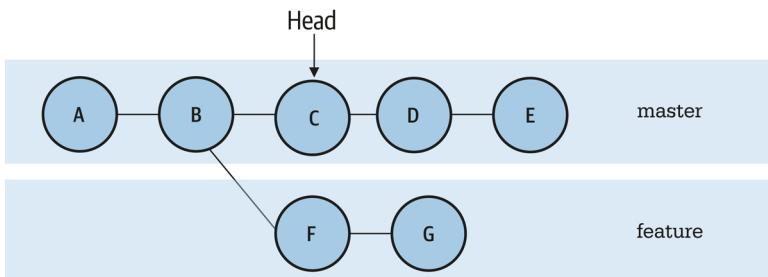


Abbildung 8-10: Bei einem *detached HEAD* sollte man nicht committen, es sei denn, man weiß genau, was man tut.

Solltest du doch aus Versehen einen Commit im Modus *detached HEAD* gemacht haben, lässt sich das Ganze heilen, indem du einfach einen Branch erzeugst:

```
$ git checkout -b new_branch
```

Dadurch zeigt HEAD wieder auf einen regulären Branch, und der Modus *detached HEAD* wird verlassen. Solltest du keine Commits gemacht haben, kommst du zum »Normalzustand« zurück, indem du einfach auf einen bestehenden Branch wechselst:

```
$ git checkout master
```



### Tipp: Aufräumen bei Git

Standardmäßig wird alles an Information bei Git aufbewahrt, es sei denn, Git wird explizit aufgefordert, etwas zu verwerfen. In Zusammenarbeit mit anderen kann es passieren, dass Maintainer A einen Branch auf GitHub löscht und Maintainerin B weiterhin eine lokale Referenz auf diesen Branch besitzt. Mit dem Parameter `prune` (deutsch »stutzen«) beim Befehl `git fetch`<sup>9</sup> kann man Git veranlassen, entsprechende Aufräumarbeiten während des Fetchens durchzuführen:

```
$ git fetch --prune
```

7 <https://www.git-tower.com/learn/git/ebook/en/command-line/advanced-topics/submodules>

8 »Verlieren« ist nicht ganz richtig, mit `git reflog` gibt es Möglichkeiten, den verlorenen Commit wiederzuholen. Einen Anfänger könnte das aber erst einmal verwirren, daher meine Empfehlung: Nicht im Modus *detached HEAD* committen!

9 Siehe <https://git-scm.com/docs/git-fetch>.

Wenn du jetzt noch einmal git status aufrufst, bekommst du plötzlich eine ganz andere Rückmeldung:

```
$ git status
Auf Branch master
Ihr Branch ist 1 Commit hinter 'origin/master', und kann vorgespult werden.
  (benutzen Sie "git pull", um Ihren lokalen Branch zu aktualisieren)

nichts zu committen, Arbeitsverzeichnis unverändert
```

Netterweise sagt uns Git wieder, was wir tun können, um unseren Klon zu aktualisieren:

```
$ git pull
Aktualisiere a70d14c..3480a5f
Fast-forward
 README.md | 3 ++
 1 file changed, 2 insertions(+), 1 deletion(-)
```

Wir hatten weiter oben ja bereits gesehen, dass pull = fetch + merge ist. git fetch haben wir schon gemacht, alternativ würde auch Folgendes gehen:

```
$ git merge origin/master
```

Du hast nun ein Projekt auf GitHub mithilfe von Git lokal auf deine Platte bekommen. Zudem weißt du jetzt, wie du Änderungen auf GitHub in dein lokales Git-Repository überführen kannst. Szenario 1 und Szenario 2 waren bisher immer nur in eine Richtung unterwegs, also entweder von lokal zu remote oder von remote zu lokal. Das ändern wir jetzt in Szenario 3.

## Standardarbeitsabläufe für Szenario 2

Ein (beliebiges) GitHub-Repository kannst du dir mit Git so herunterladen:

```
[...GitHub-Repository mit mindestens 1 Datei anlegen...]
$ git clone https://github.com/USERNAME/REPONAME.git
$ cd REPONAME
```

Ändern sich Dateien auf GitHub, hältst du dein lokales Git-Projekt so aktuell:

```
[...Dateien auf GitHub ändern...]
$ git fetch
$ git merge
Alternativ: $ git pull
```

## Szenario 3: Geforktes Projekt auf GitHub lokal zu Git holen

Wir forken ein fremdes Repository und schauen, wie wir es schaffen, mit diesem fremden Repository synchron zu bleiben (siehe Abbildung 8-11).

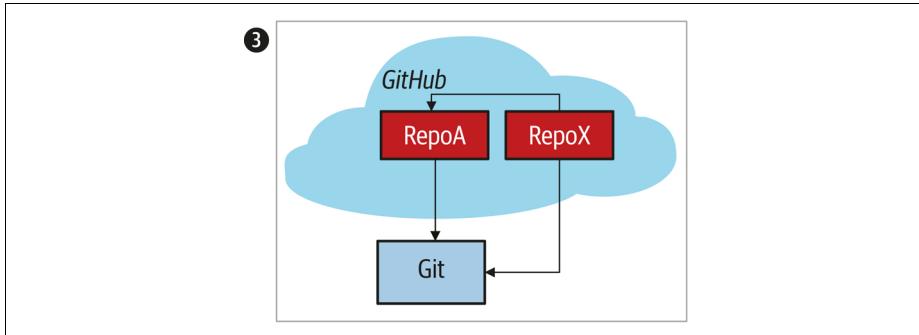


Abbildung 8-11: Szenario 3: Geforktes Projekt auf GitHub lokal zu Git holen

Wir führen dafür folgende Schritte durch:

1. Wir forken auf GitHub ein Projekt.
2. Wir klonen den Fork lokal mittels Git.
3. Wir legen in Git ein zweites Remote-Repository fest.
4. Wir aktualisieren den lokalen Klon aus dem Originalprojekt.

## Wir forken auf GitHub ein Projekt

Wir stehen jetzt vor der Herausforderung, dass wir zu Übungszwecken ein fremdes Repository benötigen, dass sich relativ flott verändert.<sup>10</sup> Wir wollen ja unter anderem das Aktualisieren üben und nicht tagelang darauf warten, bis sich die Projektleiterin bequemt, etwas anzupassen. Mein Vorschlag für ein solches Repository ist wieder *first-contributions*<sup>11</sup>, das wir bereits im Abschnitt »Loslegen 2 – anders forken« auf Seite 129 in Kapitel 6 kennengelernt haben. Das Repo ändert sich relativ schnell (häufig stündlich), daher ist es für unsere Testzwecke gut geeignet, und ich werde es als Beispiel nehmen. Jedes andere fremde Repository geht auch, und alternativ ist ein Projekt eines zweiten Accounts möglich. Wähle ein Repository und forke es.

## Wir klonen den Fork lokal mittels Git

Wie das geht, hast du in Szenario 2 bereits kennengelernt. Wir wechseln auf die Konsole und führen den Befehl `git clone` aus:

```
$ git clone https://github.com/USERNAME/first-contributions.git
```

Bis hierhin sind wir auf dem Stand, den wir bereits in Szenario 2 erreicht hatten. Wenn der Projektleiter des Originalprojekts jetzt Änderungen macht, ist der Fork

<sup>10</sup> Es lässt sich leider kein eigenes Repository forken.

<sup>11</sup> <https://github.com/firstcontributions/first-contributions>

veraltet und muss auf den aktuellen Stand gebracht werden. Das haben wir im Abschnitt »Ein geforktes Projekt aktuell halten« auf Seite 133 in Kapitel 6 bereits kennengelernt. Wie dort versprochen, nutzen wir jetzt einen eleganteren Weg, um den Fork aktuell zu halten, als ihn jedes Mal zu löschen und neu zu forken.

## Wir legen in Git ein zweites Remote-Repository fest

Ein kurzer Blick auf unsere konfigurierten Remote-Repositories zeigt uns, dass wir – erwartungsgemäß – nur unseren Fork eingetragen haben:

```
$ git remote -v
origin https://github.com/username/first-contributions.git (fetch)
origin https://github.com/username/first-contributions.git (push)
```

Was wir jetzt benötigen, ist eine Verknüpfung unseres lokalen Git-Repos mit dem GitHub-Repository des Originalprojekts. Das geht mit dem Befehl `git remote add`, den wir bereits aus Szenario 1 kennen:

```
$ git remote add upstream https://github.com/firstcontributions/first-
contributions.git
```

Auch hier ist es wieder möglich, ein Kürzel (auch Alias genannt) anzugeben, um die lange URL nicht bei nachfolgenden Befehlen jedes Mal eintippen zu müssen. Der Alias ist wie gehabt frei wählbar, viele Menschen nutzen dafür den Begriff `upstream`. Ich werde ihn daher hier ebenfalls verwenden. Die URL ist diesmal die URL des Originalprojekts (solltest du ein anderes fremdes Repository gewählt haben, musst du das entsprechend anpassen). Danach sollte ein Blick auf unsere konfigurierten Remote-Repositories uns Folgendes zeigen:

```
$ git remote -v
origin https://github.com/username/first-contributions.git (fetch)
origin https://github.com/username/first-contributions.git (push)
upstream https://github.com/firstcontributions/first-contributions.git (fetch)
upstream https://github.com/firstcontributions/first-contributions.git (push)
```

Wir haben jetzt also zwei Remote-Repositories mit unserem lokalen Git-Repository verknüpft. Eine Sache müssen wir noch einrichten. Wenn du dir die Konfiguration zu `upstream` anschauust, siehst du, dass nur für das Pushen eine Referenz eingerichtet ist:

```
$ git remote show upstream
[...]
      Lokale Referenz konfiguriert für 'git push':
          master versendet nach master (aktuell)
```

Da wir von diesem Repository aber auf jeden Fall pullen werden, holen wir das wie folgt noch nach:

```
$ git branch -u upstream/master
Branch 'master' folgt nun Remote-Branch 'master' von 'upstream'.
```

Der Parameter `-u` (das steht übrigens für upstream) sorgt hier wieder<sup>12</sup> für die Magie. Ein weiterer Blick in die Konfiguration bestätigt unsere Einstellungen:

```
$ git remote show upstream
[...]
  Lokaler Branch konfiguriert für 'git pull':
    master führt mit Remote-Branch master zusammen
  Lokale Referenz konfiguriert für 'git push':
    master versendet nach master (aktuell)
```

Jetzt haben wir alles so weit eingerichtet, dass wir damit arbeiten können.

## Wir aktualisieren den lokalen Klon aus dem Originalprojekt

Als Erstes müssen wir warten, bis sich auf dem Originalprojekt etwas verändert. Die Überprüfung, ob sich etwas verändert hat, findet entweder über GitHub auf der Projektstartseite statt (siehe Abbildung 8-12) oder über den Befehl `git remote` (siehe letzte Zeile lokal nicht aktuell):

```
$ git remote show upstream
[...]
  Lokale Referenz konfiguriert für 'git push':
    master versendet nach master (lokal nicht aktuell)
```



Abbildung 8-12: Auf der Startseite eines Projekts sieht man, wann die letzte Änderung erfolgt ist.

Sobald das der Fall ist, nutzen wir wieder den Befehl `git fetch`, um uns die Änderungen zu holen, und aktualisieren damit auch unseren Remote Tracking Branch, wie wir es in Szenario 2 kennengelernt haben. Da wir jetzt zwei Remote-Repositorien konfiguriert haben, müssen wir Git noch sagen, welches wir meinen. Wir wollen die Änderungen vom Originalprojekt, also nehmen wir den Alias `upstream`:

```
$ git fetch upstream
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 3 (delta 2), pack-reused 0
Entpacke Objekte: 100% (4/4), 980 Bytes | 29.00 KiB/s, Fertig.
Von https://github.com/firstcontributions/first-contributions
  d88476f5a..97230e8f8  master      -> upstream/master
```

Du solltest jetzt eine ähnliche Rückmeldung bekommen haben. Falls der Befehl nichts rückgemeldet hat, gibt es entweder keine Änderungen am Originalprojekt, oder du hast nicht den richtigen Alias verwendet. Jetzt schauen wir, was `git status`

12 Wie beim `git push` aus Szenario 1.

zu der aktuellen Situation sagt. Auch hier müssen wir wieder den richtigen Remote-Alias angeben:

```
$ git status upstream  
Auf Branch master  
Ihr Branch ist 2 Commits hinter 'upstream/master', und kann vorgespult werden.  
(benutzen Sie "git pull", um Ihren lokalen Branch zu aktualisieren)  
  
nichts zu committen, Arbeitsverzeichnis unverändert
```

Git sagt uns wieder, dass wir mit `git pull` arbeiten können, wir nehmen dieses Mal aber zu Übungszwecken den Befehl `git merge`, da wir bereits `git fetch` ausgeführt haben:

```
$ git merge  
Aktualisiere d88476f5a..97230e8f8  
Fast-forward  
 Contributors.md | 3 ++-  
 1 file changed, 2 insertions(+), 1 deletion(-)
```

Damit hätten wir auch Szenario 3 abgeschlossen. Wir haben ein fremdes Projekt geforkt, es dann via Git lokal geklonnt, und dann haben wir es aktuell gehalten, indem wir Änderungen im Originalprojekt in unser lokales Git-Repository geholt haben. Jetzt fehlt nur noch der letzte Schritt: bei einem fremden Projekt auch eigene Änderungen einreichen. Das machen wir in Szenario 4.

### Standardarbeitsabläufe für Szenario 3

Ein fremdes GitHub-Repository kannst du dir mit Git so herunterladen:

```
[...Projekt auf GitHub forken...]  
$ git clone https://github.com/USERNAME/REPONAME.git  
$ cd REPONAME
```

Dein Git-Repository verknüpfst du mit dem fremden GitHub-Repository so:

```
$ git remote add upstream https://github.com/ANDERER_USER/ORIGINALREPO.git  
$ git branch -u upstream/master
```

Ändern sich Dateien auf GitHub, hältst du dein lokales Git-Projekt so aktuell:

```
[...Änderungen auf dem Originalprojekt auf GitHub...]  
$ git fetch upstream  
$ git merge
```

## Szenario 4: Lokale Änderung an Originalprojekt übergeben

Wir erweitern Szenario 3, indem wir Anpassungen an dem geforkten Repository vornehmen und versuchen, diese an das Original-Repository zurückzuspielen (siehe Abbildung 8-13).

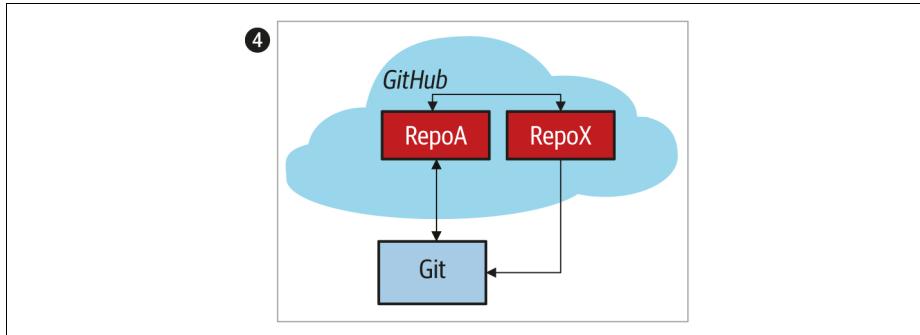


Abbildung 8-13: Szenario 4: Lokale Änderung an Originalprojekt übergeben

Im Großen und Ganzen nutzen wir jetzt alles, was wir bisher kennengelernt haben, in einem einzigen Arbeitsablauf. Wir führen dafür folgende Schritte durch:

1. Wir richten alles so ein, wie in Szenario 3 beschrieben (falls noch nicht geschehen).
2. Wir editieren lokal eine Datei und pushen sie zum Fork auf GitHub.
3. Wir erstellen einen Pull-Request aus dem Fork an das Originalprojekt.
4. Wir üben uns in Geduld und warten auf das Mergen des Pull-Requests.

## Wir richten alles so ein, wie in Szenario 3 beschrieben

Wenn du das fremde Repository frisch geforkt hast, brauchst du es natürlich nicht sofort wieder zu aktualisieren (`git fetch`). Sollte es doch notwendig sein, zur Erinnerung:

```
$ git fetch upstream  
$ git merge
```

Ich empfehle auch hier wieder *first-contributions*<sup>13</sup>, da dort das Mergen in der Regel automatisiert erfolgt.

## Wir editieren lokal eine Datei und pushen sie zum Fork auf GitHub

Sobald dein Repository wie in Szenario 3 eingerichtet ist, können wir loslegen. Das, was wir hier machen, haben wir in Szenario 1 bereits kennengelernt. Editeiere eine der lokalen Dateien auf deinem frisch geforkten Repository, z.B. *Contributors.md*, in dem von mir vorgeschlagenen Repository. Speichern, stagern und committen nicht vergessen! Danach pushe sie nach GitHub:

```
[...Datei editieren...]  
$ git add DATEI  
$ git commit -m "NACHRICHT"  
$ git push origin master
```

<sup>13</sup> <https://github.com/firstcontributions/first-contributions>

Wir pushen natürlich zu origin auf unseren Fork, da wir auf upstream, dem Originalprojekt, in der Regel keine Schreibrechte haben.

## Wir erstellen einen Pull-Request aus dem Fork an das Originalprojekt

Jetzt wechseln wir wieder auf die Weboberfläche von GitHub, und zwar in dein geforktes Repository. Erstelle dort einen Pull-Request an das Originalprojekt (wie das geht, wird im Abschnitt »Loslegen 2 – anders forken« auf Seite 129 in Kapitel 6 beschrieben).

## Wir üben uns in Geduld und warten auf das Mergen des Pull-Requests

Jetzt heißt es abwarten und Tee trinken. Im besten Fall wird dein Pull-Request schnell und anstandslos gemerget. Es kann aber auch passieren, dass die Projekteigenerin Anpassungswünsche hat oder dass es Merge-Konflikte gibt. Wie diese entstehen und was du dagegen tun kannst, sehen wir uns im nächsten Abschnitt an.

### Standardarbeitsabläufe für Szenario 4

Ein fremdes GitHub-Repository kannst du dir mit Git so herunterladen (wie Szenario 3):

```
[...Projekt auf GitHub forken...]  
$ git clone https://github.com/USERNAME/REPONAME.git  
$ cd REPONAME
```

Dein Git-Repository verknüpfst du mit dem fremden GitHub-Repository so (wie Szenario 3):

```
$ git remote add upstream https://github.com/ANDERER_USER/ORIGINALREPO.git  
$ git branch -u upstream/master
```

Ändern sich Dateien auf GitHub, hältst du dein lokales Git-Projekt so aktuell (wie Szenario 3):

```
[...Bei Änderungen auf dem Originalprojekt auf GitHub...]  
$ git fetch upstream  
$ git pull  
Alternativ: $ git merge
```

Einen Vorschlag für eine Änderung am Originalprojekt erstellst du so:

```
[...Dateien lokal ändern...]  
$ git add DATEI  
$ git commit -m "NACHRICHT"  
$ git push origin master  
[...Erstelle Pull-Request auf GitHub...]  
[...warten...]
```

Du hast jetzt erfolgreich alles einmal durchlaufen, was es an grundsätzlichem Zusammenspiel zwischen Git und GitHub gibt. Wenn du auf eigenen Repositories arbeitest, wirst du vermutlich eine Mischung aus Szenario 1 und 2 durchführen (je nachdem, wo du lieber deine Dateien editierst). Bei der Unterstützung von fremden Projekten kommen eher Szenario 3 und 4 zum Tragen. Wir schauen uns im nächsten Abschnitt den Umgang mit Merge-Konflikten an.

## Merge-Konflikte lösen

Bisher hat immer alles, was wir gemacht haben, reibungslos funktioniert. Pull-Requests ließen sich stets problemlos mergen. Das ist aber nicht immer der Fall, daher schauen wir uns an, wie solche Konflikte entstehen und wie du diese mit GitHub bzw. Git lösen kannst.

### Wie entstehen Merge-Konflikte?

Ein Merge-Konflikt<sup>14</sup> ist an sich erst einmal nichts Schlimmes und bedeutet auch nicht unbedingt, dass du etwas falsch gemacht hast. Merge-Konflikte können auf unterschiedliche Weise entstehen, beispielsweise wenn zwei Personen an ein und derselben Stelle Änderungen vornehmen oder wenn eine Person eine Datei editiert und eine andere Person diese Datei löscht. Verhindern kann man solche Konflikte in der Zusammenarbeit mit anderen nicht, daher ist es wichtig, zu wissen, wie man mit ihnen umgehen kann.

Wir schauen uns das an einem Beispiel an und nehmen dafür den Fall: »Zwei Personen editieren dieselbe Datei.« Um das praktisch durchzuspielen, ist es sinnvoll, diesen Konflikt zunächst zu erzeugen und dann aufzulösen:

1. Erstelle ein neues Repository mit einer *README.md*.
2. Erstelle einen neuen Branch, z.B. *conflict1*.
3. Verändere die Überschrift in der *README.md* und committe auf den Branch *conflict1*.
4. Erstelle einen Pull-Request von *conflict1* für diese Änderung – aber nicht merken!
5. Erstelle einen zweiten Branch, z.B. *conflict2*, und achte unbedingt darauf, dass der zweite Branch von master abgeht (siehe Abbildung 8-14).
6. Verändere die Überschrift an derselben Stelle in der *README.md* und committe auf den Branch *conflict2*.
7. Erstelle einen Pull-Request von *conflict2* für diese Änderung.
8. Merge einen der beiden Pull-Requests, z.B. den ersten.

---

<sup>14</sup> Eine Anekdote aus dem Buch »Projekt Unicorn« von Gene Kim: »Du kennst den Witz, oder? Wie lautet der Plural von ›Entwickler?‹, fragt Maxine. ›Merge-Konflikt!‹«

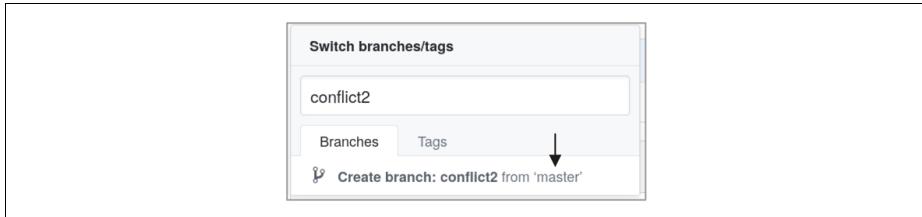


Abbildung 8-14: Um den Konflikt zu erzeugen, ist ein Abzweigen von master wichtig.

Wenn du jetzt in den anderen, noch nicht gemergten Pull-Request gehst, solltest du so etwas sehen wie das in Abbildung 8-15 Gezeigte.

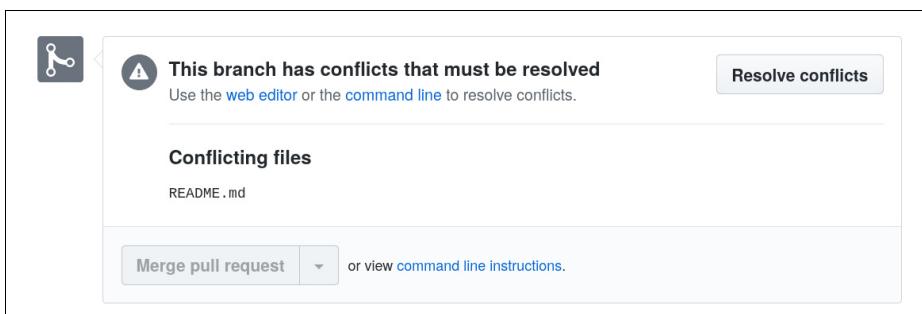


Abbildung 8-15: Bei Merge-Konflikten gibt GitHub Hilfestellung dazu, was getan werden kann.

Du siehst, dass der Button *Merge pull request* unten links ausgegraut ist, und nicht einmal du als Projekteignerin kannst einen Merge durchführen. Um hier weiterzukommen, müssen wir den Konflikt also zunächst lösen. GitHub gibt uns hier schon den Hinweis, dass du das entweder über den *web editor* (»Webeditor«) oder die *command line* (»Kommandozeile«, »Konsole«) machen kannst. Wir schauen uns beide Wege an.

## Konflikte auflösen mit GitHub (Webeditor)

Der Webeditor ist dafür gedacht, kleinere und einfach zu lösende Konflikte zu beheben. Wenn GitHub erkennt, dass ein Konflikt zu komplex ist, um ihn im Webeditor zu bearbeiten, wird der Button *Resolve conflicts* (deutsch »Löse Konflikte auf«, zu sehen in Abbildung 8-15 oben rechts) einfach ausgegraut. GitHub interpretiert alles als komplex, was über konkurrierende Zeilenänderungen hinausgeht. Dann musst du den Weg über die Konsole gehen.

Unser Konflikt ist relativ einfach, und deswegen können wir den Webeditor nutzen. Klicke hierfür auf den Link *web editor* oder wähle den Button *Resolve conflicts*, um zum Webeditor zu gelangen (siehe Abbildung 8-16).

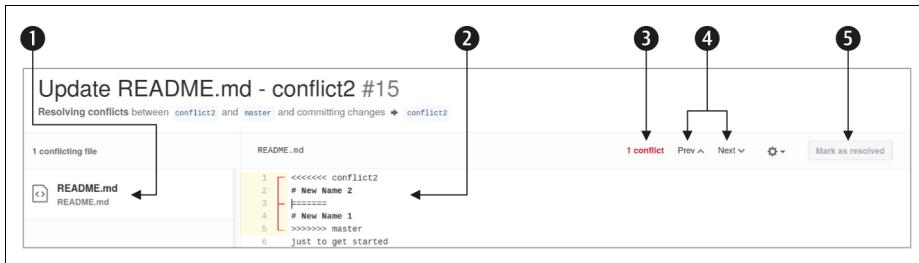


Abbildung 8-16: Der Webeditor bietet eine schlichte Oberfläche für die Beseitigung von Konflikten.

Wir sehen zum einen, welche Datei(en) vom Konflikt betroffen sind ①. Bei uns ist das aktuell nur eine, hier könnten aber auch mehrere untereinander auftauchen, die du dann durch Anklicken auswählen und bearbeiten kannst. Daneben sehen wir in ② den eigentlichen Editor, in dem unsere angeklickte Datei geöffnet ist. Bei ③ siehst du, wie viele Konflikte wir aktuell haben, und über die Buttons bei ④ kannst du zwischen all diesen Konflikten hin- und herspringen. Da wir nur einen einzigen Konflikt haben, benötigen wir diese Funktion aktuell nicht. Der Button *Mark as resolved* ⑤ wird nach dem Lösen des Konflikts anklickbar sein und uns zum nächsten Prozessschritt führen.

In meinem konkreten Fall siehst du Folgendes im Editor:

```
<<<< conflict2
# New Name 2
=====
# New Name 1
>>>> master
```

Mein Branch *conflict2* erzeugt einen Konflikt, da er die Änderung # New Name 2 durchführen will. Auf dem master-Branch ist an derselben Stelle aber bereits # New Name 1 aus dem ersten Pull-Request entstanden. Die Zeichen <<<<, ===== und >>>> sind sogenannte Konfliktmarker (*Conflict Markers*) und werden gleich noch wichtig. Die Zeichen ===== trennen dabei die beiden unterschiedlichen Versionen voneinander. Du musst jetzt entscheiden, wie du die Situation auflösen willst. Dazu hast du mehrere Optionen:

- Du kannst die Änderung aus dem Branch *conflict2* übernehmen und die auf dem master-Branch löschen.
- Oder genau umgekehrt: Du behältst die Änderungen von *master* und löschst die von *conflict2*.
- Du kannst eine Mischung aus beidem übernehmen.
- Du kannst etwas komplett Neues machen.

Wir entscheiden uns für die letzte Option, einfach um zu sehen, dass auch etwas völlig Neues passieren kann und du nicht auf das Bestehende festgelegt bist. Mache dafür Folgendes:

1. Lösche die Konfliktmarker und die beiden vorgeschlagenen Überschriften (in der Abbildung sind das die ersten fünf Zeilen).

2. Schreibe eine neue Überschrift deiner Wahl an diese Stelle.

Sobald die Konfliktmarker entfernt sind, sollte der Button *Mark as resolved* anklickbar sein. Klicke ihn also an. Danach sollten einige grüne und graue Haken zu sehen sein, ähnlich wie bei mir in Abbildung 8-17.

Hast du alle Konflikte als gelöst markiert, erscheint ein neuer Button *Commit merge* oben rechts. Wenn du ihn anklickst, erzeugst du einen neuen Commit auf dem Pull-Request, der es uns erlaubt, den Pull-Request endlich zu mergen.



Abbildung 8-17: Nach dem Auflösen aller Konflikte lassen sich die Änderungen committen.

Wenn du dir den entsprechenden Pull-Request anschaugst, siehst du dort den neuen Commit (siehe Abbildung 8-18). Und jetzt können wir auch den Pull-Request mergen.

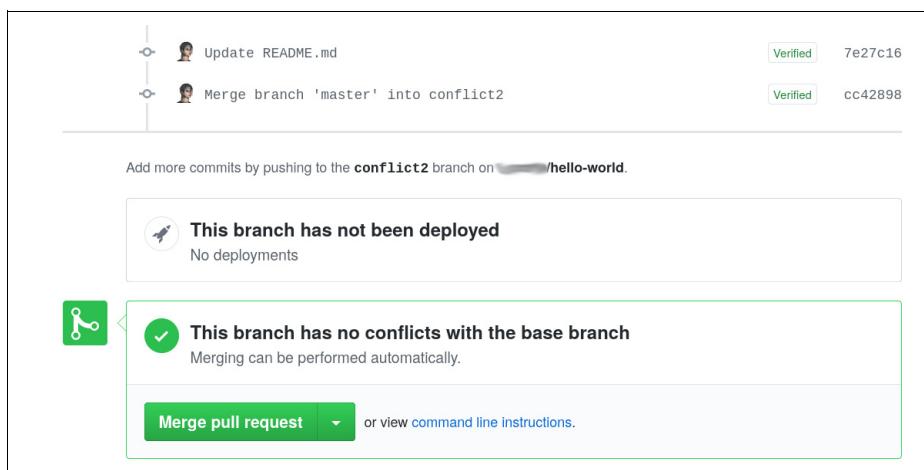


Abbildung 8-18: Der neue Commit erlaubt endlich, dass der Pull-Request gemergt werden kann.

## Konflikte auflösen mit Git (Konsole)

Manche Konflikte lassen sich nur auf der Konsole lösen, deswegen schauen wir uns in diesem Abschnitt an, wie das geht. Wir werden zwei Fälle betrachten:

1. Ein Konflikt durch gleichzeitiges Editieren – das ist derselbe Konflikttyp, den wir schon im Abschnitt zuvor bearbeitet haben.
2. Ein Konflikt durch Löschen einer Datei – das ist ein Konflikttyp, den wir nicht auf der GitHub-Oberfläche lösen können.

## Das Git-Mergetool

Git bietet über den Parameter `mergetool`<sup>15</sup> die Möglichkeit, einen von mehreren grafischen Editoren für Merge-Konflikte zu verwenden. Welche Editoren möglich und auch bereits auf dem System installiert sind, erfährst du über:

```
$ git mergetool --tool-help
```

Fehlende Editoren, wie im nachfolgenden Beispiel `meld`, lassen sich auf Linux-Systemen (Debian/Ubuntu) leicht nachinstallieren mit:

```
$ sudo apt-get install meld
```

Einen Lieblingseditor der Wahl lässt sich wie folgt festlegen:

```
$ git config merge.tool meld
```

Schlägt ein Merge fehl, kann das Mergetool wie folgt aufgerufen werden:

```
$ git mergetool
```

Danach startet eine grafische Oberfläche, die das Lösen des Merge-Konflikts erleichtert.<sup>16</sup>

## Konflikt durch gleichzeitiges Editieren

Erzeuge wieder einen Konflikt wie den im vorherigen Abschnitt beschriebenen, bei dem zwei Pull-Requests ein und dieselbe Datei anpassen wollen. Anstatt den Webeditor zu nutzen, wechseln wir auf die Konsole. Wir gehen in unser Projektverzeichnis und aktualisieren erst einmal unseren lokalen Klon:

```
$ cd REPONAME  
$ git fetch origin  
$ git merge
```

Da Branches beim Klonen nicht automatisch mitgeklont werden, müssen wir unseren remote Branch (bei mir `conflict2`) erst einmal lokal bekannt machen (bzw. einen Remote Tracking Branch festlegen):

```
$ git checkout -b conflict2 origin/conflict2  
Branch 'conflict2' folgt nun Remote-Branch 'conflict2' von 'origin'.  
Zu neuem Branch 'conflict2' gewechselt
```

<sup>15</sup> Die Dokumentation dazu siehe <https://git-scm.com/docs/git-mergetool>.

<sup>16</sup> Ein Tutorial zum Git-Mergetool findest du unter <https://gist.github.com/karenyyng/f19ff75c60f18b4b8149>.

Danach versuchen wir zu mergen und bekommen eine entsprechende Fehlermeldung:

```
$ git merge master
automatischer Merge von README.md
KONFLIKT (Inhalt): Merge-Konflikt in README.md
Automatischer Merge fehlgeschlagen; beheben Sie die Konflikte und committen Sie dann das Ergebnis.
```

Git sagt uns, welche Datei den Konflikt hat, die wir gleich noch editieren müssen. Das ist auf jeden Fall praktisch, da du ja nicht immer wissen kannst, an welchen Dateien andere parallel zu deinen Aktivitäten gerade arbeiten. Öffne die *README.md* in einem dir genehmen Editor. Du solltest etwas Ähnliches wie diese Ausgabe sehen:

```
<<<<< HEAD
# neue Überschrift - conflict2
=====
# neue Überschrift - conflict1
>>>>> master
```

Das Bild kommt uns aus dem Webeditor bekannt vor. Und genau so wie dort lösen wir den Konflikt auch hier auf. Wir entfernen die Konfliktmarker, schreiben den Text/die Überschrift deiner Wahl in die Datei, speichern, stagern, committen und, nicht vergessen, pushen:

```
$ git add README.md
$ git commit -m "NACHRICHT"
$ git push
```

Wenn du jetzt zurück zu GitHub wechselst und dir den noch offenen Pull-Request anschaugst, wirst du sehen, dass der Pull-Request einen weiteren Commit bekommen hat und er jetzt gemerget werden kann, ganz so wie über den Webeditor (siehe Abbildung 8-19).

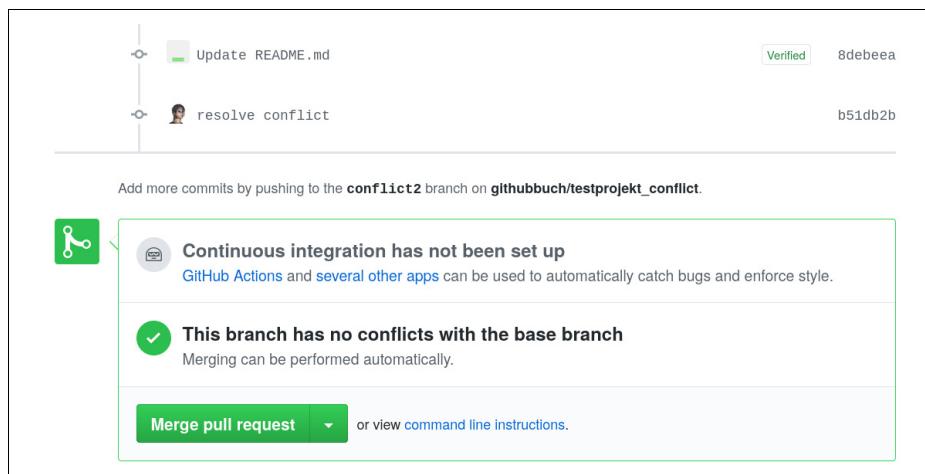


Abbildung 8-19: Nach Konfliktlösung auf der Konsole ist der Pull-Request in GitHub jetzt mergbar.

## Konflikt durch Löschen einer Datei

Bis hierhin konntest du sowohl die Konsole als auch den Webeditor nutzen. Für den nachfolgenden Konflikt ist zwingend eine Lösung über die Konsole notwendig. Der Konflikt stellt die Situation dar, in der eine Person eine Datei editiert, während eine andere eben diese Datei löschen möchte. Das Vorgehen ähnelt dem in den vorangegangenen Beispielen. Der Ablauf sieht jetzt so aus:

1. Erstelle ein neues Repository mit einer *README.md* (oder verwende das zuletzt genutzte Repository).
2. Erstelle einen neuen Branch, z.B. *conflict3*.
3. Verändere die Überschrift in der *README.md* und committe auf den Branch *conflict3*.
4. Erstelle einen Pull-Request von *conflict3* für diese Änderung – aber nicht mergen!
5. Erstelle einen zweiten Branch, z.B. *conflict4*, und achte auch hier wieder darauf, dass der zweite Branch von master abgeht (siehe Abbildung 8-14).
6. Lösche die *README.md* und committe auf den Branch *conflict4*.
7. Erstelle einen Pull-Request von *conflict4* für diese Änderung.
8. Merge einen der beiden Pull-Requests, z.B. den ersten.

Wenn du dir jetzt den zweiten, noch nicht gemergten Pull-Request anschaugst, ist eine Lösung über den Webeditor nicht mehr möglich (siehe Abbildung 8-20).

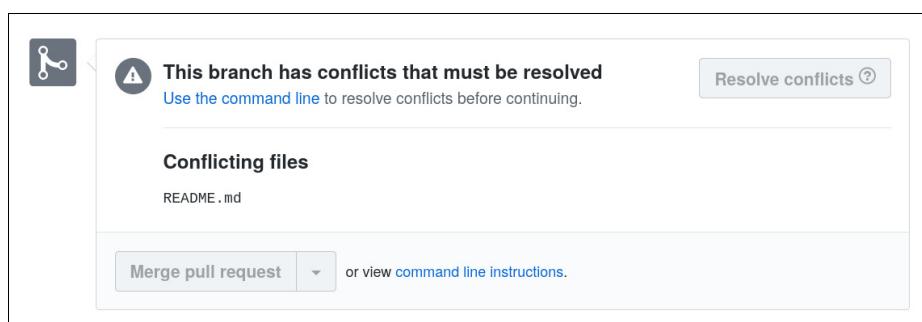


Abbildung 8-20: Manche Konflikte lassen sich nicht über den Webeditor lösen.

Wir wechseln auf die Konsole und gehen zunächst die gleichen Schritte wie zuvor mit dem Webeditor: ins Projektverzeichnis wechseln, Klon aktualisieren, Remote Tracking Branch festlegen:

```
$ cd REPONAME
$ git fetch origin
$ git merge
$ git checkout -b conflict4 origin/conflict4
Branch 'conflict4' folgt nun Remote-Branch 'conflict4' von 'origin'.
Zu neuem Branch 'conflict4' gewechselt
```

Danach versuchen wir noch einmal, unseren Branch in master zu mergen, und schauen, was Git dazu sagt:

```
$ git merge master  
KONFLIKT (ändern/löschen): README.md gelöscht in HEAD und geändert in master.  
Stand master von README.md wurde im Arbeitsbereich gelassen.  
Automatischer Merge fehlgeschlagen; beheben Sie die Konflikte und committen Sie  
dann das Ergebnis.
```

Wir erinnern uns: HEAD ist der Bereich, auf dem wir aktuell unterwegs sind (hier Branch *conflict4*), und Git erläutert uns genau die Situation, die wir gerade künstlich geschaffen haben. Ein Blick in git status versorgt uns mit weiteren Informationen:

```
$ git status  
Auf Branch conflict4  
Ihr Branch ist auf denselben Stand wie 'origin/conflict4'.  
  
Sie haben nicht zusammengeführte Pfade.  
(beheben Sie die Konflikte und führen Sie "git commit" aus)  
(benutzen Sie "git merge --abort", um den Merge abzubrechen)  
  
Nicht zusammengeführte Pfade:  
(benutzen Sie "git add/rm <Datei>...", um die Auflösung entsprechend zu markieren)  
von uns gelöscht: README.md  
  
keine Änderungen zum Commit vorgemerkt (benutzen Sie "git add" und/  
oder "git commit -a")
```

Git gibt uns netterweise wieder Hinweise dazu, was getan werden muss, um den Konflikt zu lösen. Die entscheidende Zeile ist hierbei:

```
Nicht zusammengeführte Pfade:  
(benutzen Sie "git add/rm <Datei>...",  
um die Auflösung entsprechend zu markieren)  
von uns gelöscht: README.md
```

Du kannst dich jetzt entscheiden, die angegebene Datei beizubehalten oder zu löschen. Wir wollen den Konflikt lösen, indem wir die Datei behalten. Dafür überführen wir sie in den Staging-Bereich mit git add und pushen sie danach auf GitHub:

```
$ git add README.md  
$ git commit -m "NACHRICHT"  
$ git push
```

Falls die Datei wirklich gelöscht werden soll, würden wir dasselbe mit git rm durchführen:

```
$ git rm README.md  
$ git commit -m "NACHRICHT"  
$ git push
```

Wenn du jetzt zurück zu GitHub wechselst, siehst du auch wieder, dass ein weiterer Commit hinzugekommen ist, der den Pull-Request mergbar macht.

Mit diesen Konfliktlösungsmöglichkeiten hast du jetzt ein erstes Rüstzeug, um Merge-Konflikte aus der Welt zu schaffen. Wenn du das Risiko für Merge-Konflikte generell reduzieren möchtest, gilt es, früh und häufig zu committen und zu mergen.<sup>17</sup>

Wir sind jetzt fast am Ende des Git-GitHub-Zusammenspiel-Teils. Eine kleine, aber sehr hilfreiche Sache möchte ich dir noch mit auf den Weg geben, nämlich wie du dir das Einloggen von Git zu GitHub erleichtern kannst.

## Log-in-Erleichterungen bei HTTPS

Nervt es dich, ständig dein Passwort eintippen zu müssen, sobald du mit Git etwas pushen möchtest? In diesem Abschnitt zeige ich dir zwei Wege, mit denen du den Vorgang vereinfachen kannst. Du hast für die Kommunikation von Git zu GitHub das Protokoll HTTPS<sup>18</sup> benutzt, z.B. bei git clone:

```
$ git clone HTTPS://github.com/username/reponame.git
```

Jedes Log-in via HTTPS benötigt einen Benutzernamen und ein entsprechendes Passwort, und beides hast du bisher jedes Mal per Hand eingegeben oder aus einem Passwort-Safe kopiert. Git bietet zur Unterstützung eine Funktion namens credential.helper an. Diese erlaubt dir, deine Zugangsdaten entweder für eine festgelegte Zeit oder dauerhaft zu speichern. Bei jedem Verbindungsversuch übernimmt der credential.helper dann die Eingabe der benötigten Daten. Um diese Funktion nutzen zu können, muss dein Git in der Version 1.7.10 oder höher vorliegen.

Es gibt auch noch andere Möglichkeiten, das Log-in zu vereinfachen, beispielsweise über SSH<sup>19</sup>. Das einzurichten, ist aber deutlich komplexer und soll daher hier nicht weiter Thema sein.

## Zugangsdaten auf Zeit zwischenspeichern (meine Empfehlung)

Um Zugangsdaten temporär zu speichern, gibst du sie nach dem Einrichten einmal ein, für den Rest des definierten Zeitraums werden sie aus dem Cache geholt. Ist der Zeitraum abgelaufen, musst du sie wieder eintippen. Das Einrichten ist ganz einfach:

```
$ git config --global credential.helper cache
```

---

<sup>17</sup> Das ist auch einer der Kerngedanken von Continuous Integration (siehe Erklärbärbox »Continuous Integration« in Kapitel 4).

<sup>18</sup> HTTPS steht für *Hypertext Transfer Protocol Secure* (deutsch »sicheres Hypertext-Übertragungsprotokoll«) und ist ein verschlüsseltes Kommunikationsprotokoll im Internet.

<sup>19</sup> SSH steht für *Secure Shell* und beschreibt sowohl ein Netzwerkprotokoll für verschlüsselte Verbindungen als auch eine Software, um solche Verbindungen herzustellen.

Das Wort `cache` ist hier der entscheidende Parameter. Wenn du es nicht anders festlegst, sind 15 Minuten (900 Sekunden) als Standard eingestellt. Erscheint dir das zu lang (oder zu kurz), kannst du durch Angabe in Sekunden den Wert anpassen, beispielsweise auf eine Stunde (3600 Sekunden):

```
$ git config --global credential.helper 'cache --timeout=3600'
```

Aus meiner Sicht ist die temporäre Speicherung ein guter Kompromiss zwischen Komfort und Sicherheit – auch wenn du dann hin und wieder dein Passwort erneut eingeben musst.

## Zugangsdaten dauerhaft speichern

Falls du deine Zugangsdaten dauerhaft speichern möchtest, bietet der `credential.helper` hier ebenfalls eine Lösung an:

```
$ git config --global credential.helper store
```

Hier ist das Wort `store` das entscheidende. Damit werden deine Zugangsdaten in der Datei `.git-credentials` gespeichert, und zwar im *Klartext*. Klartext bedeutet, dass jeder und jede mit Zugang zu dieser Datei deine Zugangsdaten lesen (und anwenden) kann. Es handelt sich also um eine unverschlüsselte Textdatei. Falls du den Dateinamen selbst festlegen möchtest, geht das mit:

```
$ git config --global credential.helper store --file ~/.MEINEDATEI
```

Ich empfehle diesen Weg aus Sicherheitsgründen ausdrücklich nicht! Falls du dich dafür entscheidest, ist es natürlich enorm wichtig, die Zugangsdaten abzusichern.



### Tipp: `credential.helper` wieder loswerden

Es kann Situationen geben, in denen du den `credential.helper` wieder loswerden möchtest oder musst. Das geht über:

```
$ git config --global --unset credential.helper
```

Du hast jetzt alle Basiswerkzeuge kennengelernt, um mit Git und GitHub gut arbeiten zu können. Das Buch könnte hier zu Ende sein. Du möchtest aber noch mehr coolen Kram? Z.B. Pull Requests nach bestimmten Kriterien automatisch schließen lassen oder neue Issues automatisch labeln? Dann geht es im nächsten Kapitel weiter.



## KAPITEL 9

# Der GitHub Marketplace – Actions und Apps

Wenn du dein Projekt um weitere Features erweitern möchtest, ist der GitHub Marketplace<sup>1</sup> die richtige Adresse (siehe auch Abbildung 9-1). Hier gibt es die Möglichkeit, sogenannte *Actions* und *Apps* zu unterschiedlichen Themen (wie beispielsweise Projektmanagement, Lokalisation oder Sicherheit) »hinzuzuinstallieren«, beispielsweise um GitHub mit der Chatsoftware Slack zu nutzen oder um die Übersetzung des Projekts in unterschiedliche Sprachen einfacher managen zu können.



### Am Ende des Kapitels kannst du ...

- eine App aus dem Marketplace in ein Repository installieren und anpassen.
- eine Action aus dem Marketplace in ein Repository installieren und anpassen.
- eine eigene Action programmieren.
- Passwörter oder Ähnliches in deinem Quelltext geheim halten.

Sollten dir einige oder viele der Actions oder Apps nichts sagen (»Was zur Hölle ist Docker?«): Keine Sorge, die Chance ist groß, dass du sie (noch) nicht brauchst.

## Was können Actions und Apps?

Actions und App sind in der Regel dazu da, dir das Leben durch das Automatisieren lästiger Handarbeiten zu erleichtern, z.B. im Bereich Continuous Integration, Delivery und Deployment (siehe Erklärbärbox »Continuous Integration« in Kapitel 4). Aber auch das Managen von Branches oder der Umgang mit Issues können automatisiert werden. Bei den Apps gibt es neben kostenlosen und auch kostenpflichtige. Die Actions sind in der Regel kostenfrei.

Was ist der Unterschied zwischen Action und App? Die Begriffe sind nicht ganz trennscharf. Actions sind »Community-driven« (deutsch etwa »aus der Community heraus angestoßen«), und wenn du Lust, Zeit und die entsprechenden Fähig-

<sup>1</sup> <https://github.com/marketplace>

keiten hast, kannst du dich mit eigenen Actions im Marketplace verewigen.<sup>2</sup> Die Apps werden an einer anderen Stelle im Repository konfiguriert als Actions, aber so manche App findet man auch als Action wieder. Verwirrend? Ja, aber so wichtig ist die Unterscheidung erst einmal nicht. Wir schauen uns beides an.

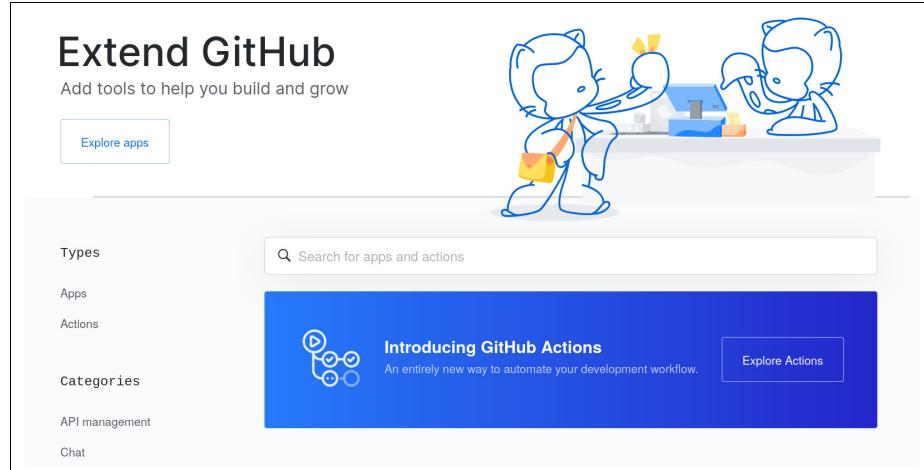


Abbildung 9-1: Im Marketplace findet man Apps und Actions, die man nutzen kann, um das Arbeiten mit GitHub zu vereinfachen.

In diesem Abschnitt werden wir eine App und eine Action in jeweils einem deiner Repositories installieren. Ich habe dafür relativ einfache gewählt, die sich leicht installieren lassen. Wie du eine eigene Action erstellen kannst, sehen wir uns im Anschluss an. Das Thema ist allerdings so umfassend, dass ich dir hier nur einen kleinen Einstieg geben möchte. Am besten ist es, wenn du für jeden der nachfolgenden Abschnitte ein neues Repository erstellst. So kannst du Änderungen im Repository leichter nachverfolgen.

## Eine App aus dem Marketplace installieren

In Projekten können sich im Laufe der Zeit Issues und Pull-Requests ansammeln, die keine Relevanz mehr haben und eigentlich geschlossen werden sollten – beispielsweise weil die Ursache von 20 Issues durch eine Codeänderung gefixt wurde und die Maintainer aus Unachtsamkeit oder Versehen nicht alle dazugehörigen Issues schließen. Um das Repository aufgeräumt zu halten und auch potenzielle Conributoren nicht abzuschrecken (das Alter und die Aktivität von Issues und Pull-Requests waren im Abschnitt »Projekt über Issues und Pull-Requests bewerten« auf Seite 126 in Kapitel 6 ein Bewertungsindikator für ein Projekt), wollen wir für diese Aufgabe eine App installieren, die uns hierbei automatisiert unterstützt.

<sup>2</sup> GitHub stellt ebenfalls eigene Actions bereit, diese sind im GitHub-Actions-Repository zu finden: <https://github.com/actions>.

Wir werden dafür die App *Stale* (deutsch etwa »abgestanden«) installieren, die in unserem Repository automatisch alle Issues und Pull-Requests bearbeiten wird, auf denen eine bestimmte Zeit keine Aktivität mehr stattgefunden hat (die also etwas »abgestanden« sind). Mit den Standardeinstellungen passiert Folgendes:

1. Issues und Pull-Requests, bei denen 60 Tage lang nichts passiert ist, bekommen automatisch das Label *wontfix*. Es wird zudem ein (konfigurierbarer) Kommentar hinzugefügt.
2. Issues und Pull-Requests mit dem Label *wontfix*, bei denen nach sieben Tagen immer noch nichts passiert ist, werden automatisch geschlossen.

Du kannst die Zeitangaben, Labels, Kommentare und vieles andere mehr nach deinen Wünschen anpassen.

## App installieren

Wir installieren zunächst die App. Lege dafür ein neues Repository an und gehe dann auf die Website der App<sup>3</sup>. Diese sollte ähnlich aussehen wie in Abbildung 9-2.

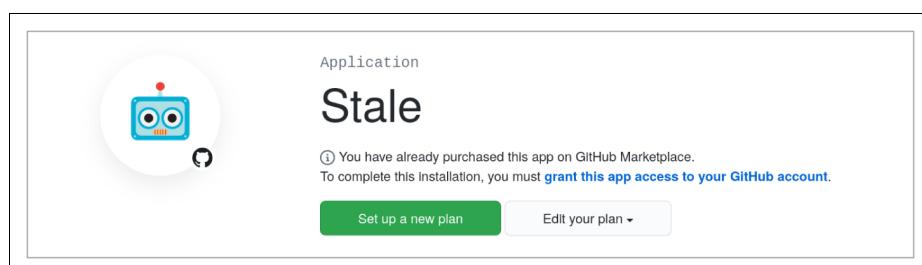


Abbildung 9-2: Die Website der App »Stale«

Wenn du den Button *Set up a new plan* (deutsch »Einen neuen Plan aufstellen«) anklickst, solltest du zu einer Übersicht kommen, die ähnlich wie in Abbildung 9-3 aussieht.

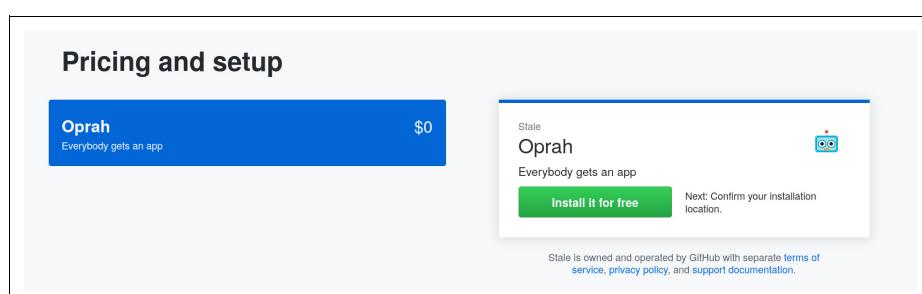


Abbildung 9-3: Diese App ist ausschließlich kostenlos erhältlich, für andere Apps gibt es neben der Basisversion noch zusätzliche kostenpflichtige Angebote.

3 <https://github.com/marketplace/stale>

Auf der linken Seite werden dann alle verfügbaren Angebote (»Pläne«) der App angezeigt. Häufig gibt es Apps in einer kostenlosen Version (»Free«) oder in einer Premium-Version mit weiteren Features. Auskunft über die Preisgestaltung geben dann in der Regel Buttons, die so oder ähnlich heißen. Bei dieser App gibt es nur einen Preisgestaltungsplan namens *Oprah*<sup>4</sup> für \$0.

Klicke *Install it for free* auf der rechten Seite an, und du kommst zum Abschluss noch eine Übersicht deiner Bestellung angezeigt (siehe Abbildung 9-4).

The screenshot shows the 'Review your order' page for the 'Oprah' app. On the left, there's a summary of the purchase: 'Stale' and 'Oprah' are selected, with the note 'Everybody gets an app'. Below this is the price '\$0 / month'. On the right, the 'Order total' is listed as '\$0.00 / month' for 'Oprah'. Under 'Due today', it says '\$0.00' with a note 'Prorated for Apr 13th-May 12th'. The 'Billing information' section shows 'githhubbuch' as the account, with a 'Personal account' link and a 'Switch billing account' button. At the bottom, a green button reads 'Complete order and begin installation'. Below the button, a note says 'By clicking "Complete order and begin installation", you are agreeing to the Marketplace Terms of Service, Stale's Terms of Service and the Privacy Policy.' A small note at the very bottom right says 'Next: Authorize Stale to access your account.'

Abbildung 9-4: Die Bestellübersicht bei einer exemplarischen App

Klicke dort auf *Complete order and begin installation* (deutsch »Schließe Bestellung ab und beginne mit der Installation«), und du wirst aufgefordert, einen Installationsort für die App anzugeben (siehe Abbildung 9-5). Hier kannst du auswählen, ob die App für alle deine Repositories gelten soll oder nur für bestimmte. Wir wollen die App nur für ein Repository installieren. Wähle dafür *Only select repositories* (deutsch »Nur ausgewählte Repositories«). Klicke *Select repositories* (deutsch »ausgewählte Repositories«) an und wähle dein gerade erstelltes Repo aus. Es ist auch möglich, mehrere Repositories durch Anklicken auszuwählen. Unter dem Drop-down siehst du, welche Rechte diese App einfordert, um ihren Dienst zu tun. Da wir hier nichts weiter einstellen können, klicke auf den *Install*-Button.

Damit die App ihren Dienst aufnehmen kann, sind noch ein paar Dinge zu erledigen. Netterweise informiert sie uns direkt darüber (siehe Abbildung 9-6). Wir müssen eine Datei `.github/stale.yml` anlegen und diese mit dem abgebildeten Text befüllen. Diese Datei ist die Konfigurationsdatei der App und steuert ihr Verhalten. Führe die Schritte wie aufgeführt durch.

4 Warum auch immer dieser Plan so heißt

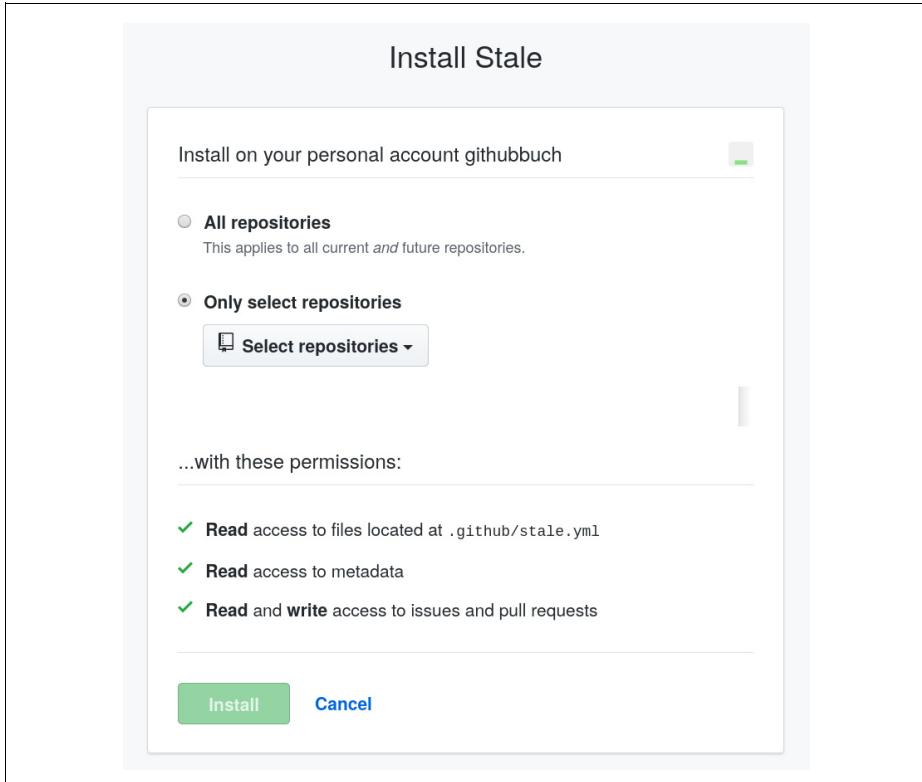


Abbildung 9-5: Apps kann man auf allen oder auch einzelnen Repositories installieren.

```
After installing the integration, create .github/stale.yml in the default branch to enable it:

# Number of days of inactivity before an issue becomes stale
daysUntilStale: 60
# Number of days of inactivity before a stale issue is closed
daysUntilClose: 7
# Issues with these labels will never be considered stale
exemptLabels:
  - pinned
  - security
# Label to use when marking an issue as stale
staleLabel: wontfix
# Comment to post when marking an issue as stale. Set to `false` to disable
markComment: >
  This issue has been automatically marked as stale because it has not had
  recent activity. It will be closed if no further activity occurs. Thank you
  for your contributions.
# Comment to post when closing a stale issue. Set to `false` to disable
closeComment: false
```

Abbildung 9-6: Um die App zum Laufen zu bringen, sind noch ein paar Handgriffe notwendig.



### Tipp: App für weitere Repositories nutzen

Falls du eine App auch für weitere Repositories einrichten möchtest, kannst du das auf der entsprechenden Marketplace-Seite machen (siehe auch Abbildung 9-7), indem du *grant this app access to your GitHub account* (deutsch »der App Zugriff auf deinen GitHub-Account geben«) anklickst. Du kommst dann auf die bereits bekannte Installationsansicht (siehe Abbildung 9-5).

The screenshot shows the GitHub Marketplace page for the 'Stale' application. At the top, it says 'Application' and shows a small icon of a blue robot head. The app name 'Stale' is prominently displayed. Below the name, there's a note: 'You have already purchased this app on GitHub Marketplace. To complete this installation, you must grant this app access to your GitHub account.' A green button labeled 'Set up a new plan' is visible, and a grey button labeled 'Edit your plan ▾' is next to it. An arrow points from the text 'grant this app access to your GitHub account.' to the 'Edit your plan' button.

Abbildung 9-7: Apps kann man auf ihrer Marketplace-Seite für weitere Repositories installieren.

## App anpassen

Wir wollen die App noch etwas an unsere Bedürfnisse anpassen. Wenn du dir die Konfigurationsdatei anschaugst, siehst du, welche Einstellungen möglich sind. Auf zwei davon möchte ich hier näher eingehen. Wir wollen die App in Aktion sehen, 60 Tage zu warten, bis irgendetwas passiert, ist allerdings etwas lang. Wir ändern daher die `daysUntilStale` wie folgt:

```
# Number of days of inactivity before an issue becomes stale  
daysUntilStale: 1
```

Damit verkürzen wir die Zeit drastisch auf einen einzigen Tag, die ein Issue hat, um als »abgestanden« zu gelten. Schau dir zudem noch die Einstellung `exemptLabels` (»befreite Labels«) an:

```
# Issues with these labels will never be considered stale  
exemptLabels:  
  - pinned  
  - security
```

Damit kannst du Ausnahmen festlegen, das heißt, Issues mit diesem Label werden von der App nicht angefasst. Das kann beispielsweise nützlich sein, um wichtige Sicherheitsrelevante Issues oder auch Issues, die eine längere Zeit für die Umsetzung benötigen, vor dem automatischen Schließen zu schützen.

Wir legen jetzt zwei Issues an, einen Issue ohne Label und einen Issue mit einem der beiden Labels `pinned` oder `security`. Die Labels musst du gegebenenfalls neu anlegen (das haben wir im Abschnitt »Einen Issue anlegen« auf Seite 32 in Kapitel 3 kennengelernt). Das Ganze sollte so ähnlich aussehen wie in Abbildung 9-8.

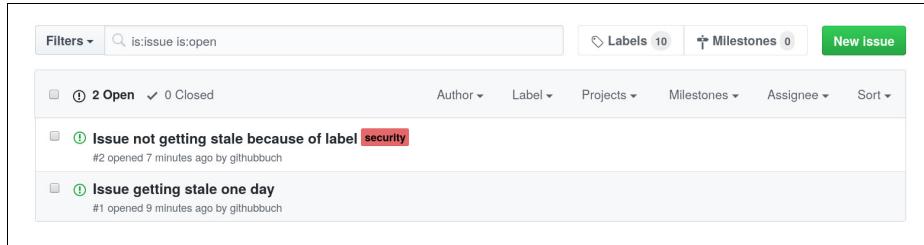


Abbildung 9-8: Zwei Issues zum Testen der App

Jetzt müssen wir uns einen Tag in Geduld üben, bis wir die Arbeitsergebnisse der App erhalten.<sup>5</sup> Nach Ablauf eines Tages sollten deine Issues wie in Abbildung 9-9 dargestellt aussehen. Der eine Issue hat ein neues Label bekommen, wie in der Konfiguration festgelegt ①, und einen zusätzlichen Kommentar ②, der ebenfalls in der Konfiguration festgelegt wurde (in unserem Fall der Standardkommentar, da wir ihn nicht angepasst haben). Da wir den Standardwert für das Schließen von »abgestandenen« Issues auf sieben Tage belassen haben, wird dieser Issue spätestens dann geschlossen werden. Die Konfiguration der App lässt sich jederzeit nach deinen Bedürfnissen in der Datei `.github/stale.yml` weiter anpassen.



Abbildung 9-9: Die zwei getesteten Issues: Issue mit der ID 1 hat korrekterweise automatisch ein neues Label bekommen.

Du hast damit erfolgreich eine App aus dem Marketplace installiert und auch gleich ausprobiert. Übrigens hast du beim Anlegen eines neuen Repos ab sofort die Möglichkeit, deine frisch installierte App auch für das neue Repo gleich mit zu installieren, siehe Abbildung 9-10.

5 Es hilft auch nicht, den Wert auf 0 zu setzen, ich habe es versucht.

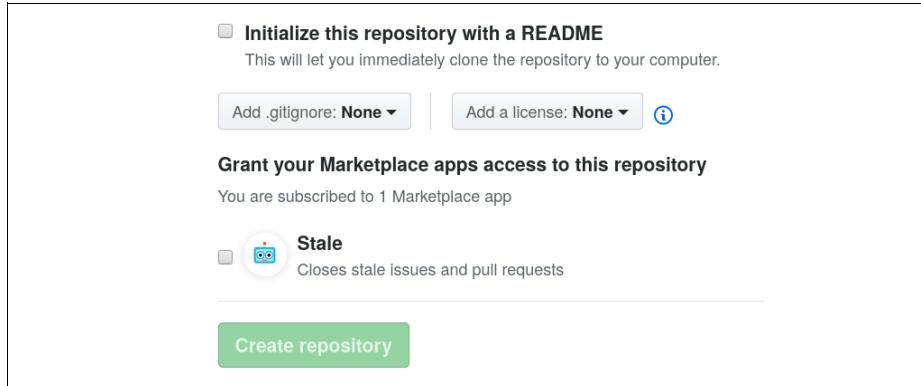


Abbildung 9-10: Beim Anlegen eines neuen Repos kann die App gleich dazustalliert werden.



#### Tipp: App-Konfiguration wiederfinden

Solltest du den Konfigurationsbereich von Apps suchen, findest du ihn unter *Settings/Integrations* im entsprechenden Repository. Dort werden alle installierten Apps aufgelistet und haben jeweils einen *Configure*-Button, der den Konfigurationsbereich der App öffnet (siehe auch Abbildung 9-11).

The screenshot shows the 'Settings' tab of a GitHub repository. On the left is a sidebar with links: 'Code', 'Issues 2', 'Pull requests 0', 'Actions', 'Projects 0', 'Wiki', 'Security', 'Insights', and 'Settings'. The 'Settings' link is underlined. The main area is titled 'Installed GitHub Apps' with the sub-instruction 'GitHub Apps augment and extend your workflows on GitHub with commercial, open source, and homegrown tools.' It lists the 'Stale' app from the previous screenshot, with a 'Configure' button to its right. The sidebar also has links for 'Options', 'Manage access', 'Branches', 'Webhooks', 'Notifications', and 'Integrations', where the 'Integrations' link is also underlined.

Abbildung 9-11: Die Konfiguration von GitHub-Apps sind unter Integrations versteckt.

## Eine Action aus dem Marketplace installieren

Nachdem wir eine App installiert haben, wollen wir uns jetzt einer Action widmen. Die Action, die wir nun installieren werden, heißt *WIP*<sup>6</sup> (das steht für *Work In Progress*, zu Deutsch »in Arbeit«). Sie wird auch *Do not Merge* (deutsch etwa »nicht mergen«) genannt, und es gibt sie auch als App<sup>7</sup>. Wir installieren aber die Action, um ein Gefühl dafür zu bekommen, worin der Unterschied zwischen einer App und einer Action besteht. Die Action macht Folgendes:

6 <https://github.com/marketplace/actions/wip>

7 <https://github.com/marketplace/wip>

- Wenn im Titel eines Pull-Requests *WIP* steht, kann er nicht gemerget werden (Ausnahme: Die Projekteignerin kann trotzdem mergen, falls wir nicht explizit über die Option *Include administrators* angegeben haben, dass sie es nicht darf).

Das kann insbesondere dann nützlich sein, wenn du (oder andere) zwar schon Pull-Requests anlegen möchtest, diese aber noch nicht fertig sind und daher noch nicht gemerget werden sollen – etwa weil du zunächst in dem Pull-Request mit deinen Maintainern den Inhalt diskutieren und gegebenenfalls noch weitere Commits machen möchtest. Damit verhinderst du, dass aus Versehen ein anderer Maintainer den Pull-Request mergt. Wir werden in drei Schritten vorgehen:

1. Wir installieren die Action.
2. Wir probieren sie aus und justieren noch etwas nach.
3. Wir schauen uns an, was die Action hinter den Kulissen macht.

## Action installieren

Wir beginnen auf der Website der Action<sup>8</sup>. Dort steht meistens, wie die Installation durchzuführen ist (siehe Abbildung 9-12). Meiner Erfahrung nach sind die Beschreibungen leider nicht immer optimal. Auch bei dieser Action ist noch etwas Luft nach oben.

### DO NOT MERGE – as an action.

build passing Greenkeeper enabled

This GitHub Action sets a pull request status to pending if the title includes "WIP".

An example workflow looks like this (switch to the [Edit new file](#) tab when creating a new workflow and paste the code below):

```
name: WIP
on:
  pull_request:
    types: [ opened, synchronize, reopened, edited ]

jobs:
  wip:
    runs-on: ubuntu-latest
    env:
      GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
    steps:
      - uses: wip/action@v1.0.0
```

Then to prevent PRs from being merged, enable the `WIP (action)` status check in your Settings > Branch > [Branch Name] > Protect matching branches > Require status checks to pass before merging

Abbildung 9-12: Auf der Action-Seite findet man in der Regel Informationen zur Installation.

8 <https://github.com/marketplace/actions/wip>

Lege für die Action ein neues Repository an. Um die Action zu installieren, begeben wir uns in unserem Repository zum Menüpunkt *Actions*. Dort wählst *set up a workflow yourself* (deutsch etwa »richte selbst einen Workflow ein«) aus, siehe Abbildung 9-13.

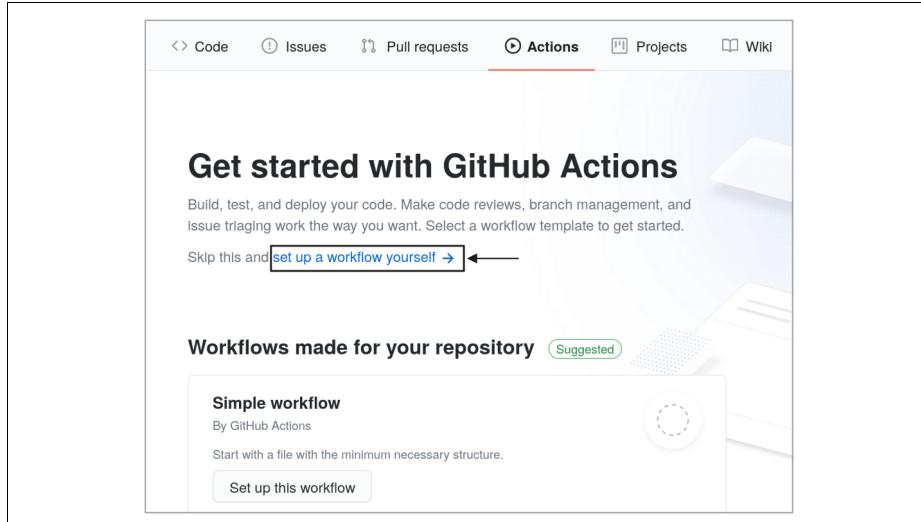


Abbildung 9-13: Neue Actions lassen sich unter anderem über den Menüpunkt Actions anlegen.

Wir sehen jetzt eine ganze Menge (siehe Abbildung 9-14). GitHub hat für uns in dem Unterordner `.github/workflows` eine neue Datei erstellt und diese `main.yml` genannt ①.

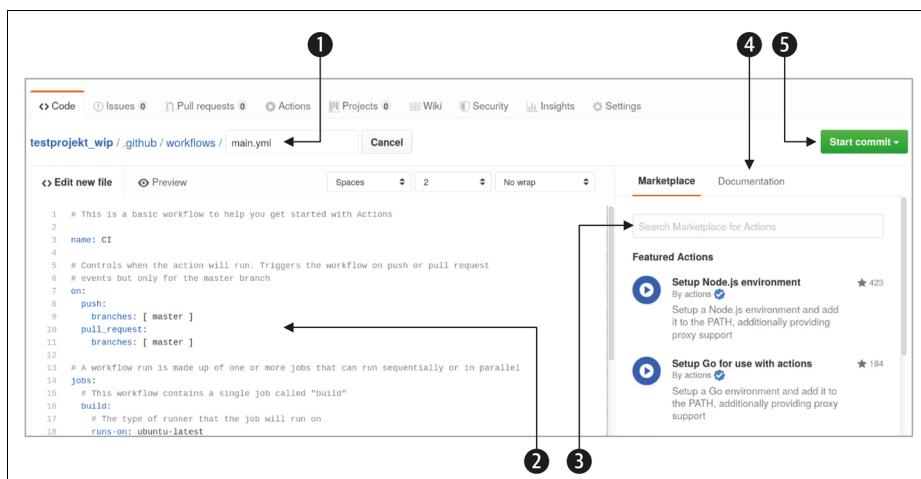


Abbildung 9-14: Beim Anlegen einer neuen Action befüllt GitHub diese zu Anfang mit einer Beispiel-Action.

Behalte die Ordnerstruktur so bei, die Datei darfst du gern umbenennen, aber die Dateiendung sollte weiterhin `.yml` lauten (eine Erläuterung zu dem Dateityp ist in der Erklärbarbox »YAML« zu finden). GitHub hat zudem diese Datei gleich mit einem Beispiel befüllt ❷. Diesen Text kannst du komplett löschen. Kopiere hier den Quelltext von der Action-Webseite hinein.

Falls du noch keine Action ausgewählt hast, kannstest du das über die Marktplatzsuche nachholen ❸. Für uns ist das gerade nicht relevant, da wir ja bereits eine Action ausgewählt haben. In der Dokumentation findest du zudem allgemeine Informationen zu der Syntax von Actions, etwa wie du einstellen kannst, dass eine Action beispielsweise alle acht Minuten durchgeführt wird ❹. Das ist für uns im Moment aber ebenfalls nicht relevant. Wähle `Start commit` und speichere die Datei ❺.

## YAML

YAML<sup>9</sup> steht für *YAML Ain't Markup Language* – grob übersetzt mit »YAML ist keine Auszeichnungssprache« – und ist ein sogenanntes rekursives Akronym. Es handelt sich dabei um eine für Menschen leicht lesbare Sprache, um Daten in eine bestimmte Ordnung zu bringen – man sagt auch, um diese zu serialisieren. Das wird gern für Konfigurationsdateien genutzt, um sogenannte Schlüssel-Wert-Paare aufzubauen. YAML kann dabei von unterschiedlichsten Programmiersprachen ausgelesen und verarbeitet werden (z.B. Java, C, C++, Perl, PHP).

YAML-Dateien sind leicht zu erkennen, da sie normalerweise die Endung `.yaml` oder `.yml` besitzen. Zuweisungen beginnen mit einem Doppelpunkt, gefolgt von einem Leerzeichen, um ein Schlüssel-Wert-Paar festzulegen (`:`). Einträge in Listen fangen mit einem Bindestrich und einem Leerzeichen an (`-`), Kommentare mit einer Raute (`#`). Möchte man langen Text übersichtlich darstellen, verwendet man das Symbol `>` – in der YAML-Datei kann man jetzt so viele Umbrüche machen, wie man für die bessere Übersicht benötigt, die Auswertung des Texts erfolgt, als wäre es eine lange Zeile. Möchte man das Gegenteil erreichen – mehrzeilig abgebildeter Text soll bitte auch mehrzeilig bleiben –, verwendet man das Symbol `|` (auch »Pipe« genannt). Eine YAML-Datei könnte beispielhaft so aussehen:

```
# Ich bin ein Kommentar
name: Klaus

hobbies:
  - Reiten
  - Schlafen

beschreibung: >
  Dieser Text wird bei der Auswertung
  in eine Zeile geschrieben und nur
  zur Übersicht in mehrere Zeilen umbrochen.
```

<sup>9</sup> <https://yaml.org/>

```
# Die Mailsignatur wollen wir natürlich nicht in einer Zeile haben  
mailsignatur: |  
  Klaus  
  Diplom-Reiter  
  klaus@klaus.de
```

Wichtig ist es, die Einrückungen mit Leerzeichen zu machen und keine Tabulatoren zu verwenden, da es je nach Parser ansonsten zu Fehlermeldungen kommen könnte.

Detaillierte Informationen zum Aufbau sind unter anderem der entsprechenden Spezifikation zu entnehmen.<sup>10</sup> Fast schon erwartungsgemäß hat YAML natürlich auch ein eigenes GitHub-Repo<sup>11</sup>.

## Action ausprobieren und feinjustieren

Jetzt wollen wir die Action in Aktion sehen. Erzeuge einen neuen Pull-Request (z.B. über das Anlegen eines neuen Branchs und dem Committen einer neuen Datei auf den Branch) und schreibe WIP in den Titel (z.B. »Fuege README.md hinzu – WIP«). Nach dem Anlegen sollte dein Pull-Request ähnlich aussehen wie der in Abbildung 9-15.

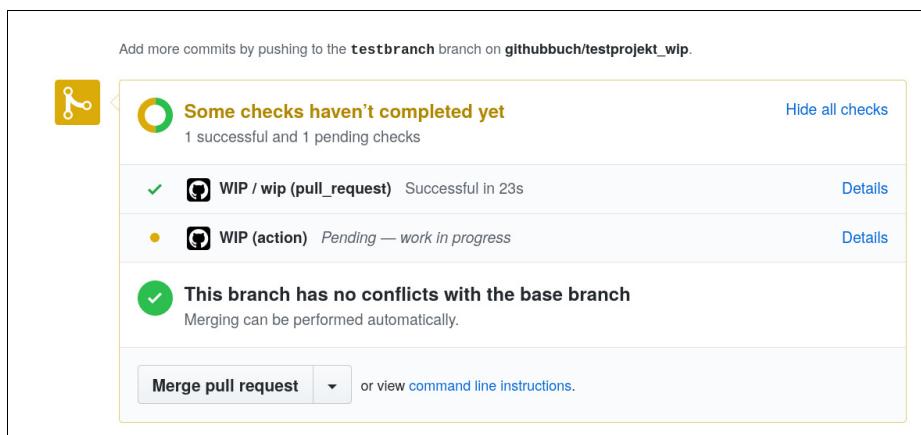


Abbildung 9-15: Nach Installation der Action durchlaufen alle Pull-Requests zwei Prüfungen.

Dort sind jetzt zwei Prüfungen durchgeführt wurden, einmal *WIP / wip (pull\_request)* und einmal *WIP (action)*. Die erste Prüfung war erfolgreich – gekennzeichnet durch einen kleinen grünen Haken vor dem Namen –, die zweite Prüfung nicht, dargestellt durch einen kleinen gelben Punkt. Meine Vermutung ist, dass die erste

10 <https://yaml.org/spec/1.2/spec.html>

11 <https://github.com/yaml/yaml>

Prüfung den Titel auf das Kürzel WIP hin untersucht und schlussendlich immer erfolgreich ist, da sie irgendwann »enthält WIP oder enthält kein WIP« sagen kann.<sup>12</sup> Die zweite Prüfung ist so gebaut, dass sie fehlschlägt, solange WIP im Namen auftaucht. Das wird markiert durch *Pending – work in progress* (deutsch »ausstehend – in Arbeit«).

Der Pull-Request lässt sich allerdings trotzdem mergen (der Button ist nicht ausgegraut), egal ob als Projekteignerin oder Maintainer. Das ist unpraktisch, falls jemand nicht so genau hinschaut und doch aus Versehen auf den Merge-Button klickt. Das können wir aber mithilfe einer Regel für einen *protected Branch* ändern. Wie das geht, hast du bereits in Kapitel 4 im Abschnitt »Gutes schützen – Protected Branches« auf Seite 67 kennengelernt. Erstelle für alle Branches die neue Regel *Require status checks to pass before merging* (deutsch etwa »Erfordert Statusprüfungen vor dem Mergen«) und wähle *WIP (action)* aus, wie in Abbildung 9-16 zu sehen. Der andere Punkt *wip* dürfte die Prüfung auf das Kürzel sein. Du kannst ihn auch mit auswählen, musst es aber nicht. Wenn du möchtest, kannst du noch *Include administrators* als weitere Option festlegen, um auch die Projekteignerin an die Regeln zu binden. Speichern nicht vergessen!

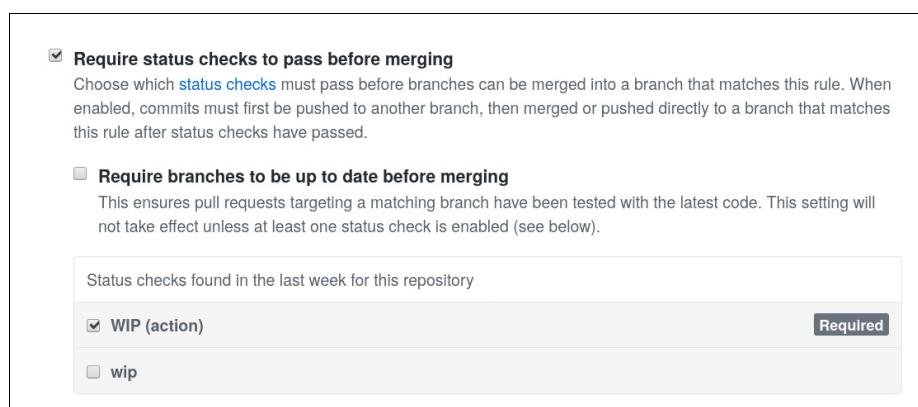


Abbildung 9-16: Versehentliches Mergen lässt sich mithilfe von *protected Branches* verhindern.

Schauen wir uns jetzt unseren Pull-Request an, sehen wir, dass die Regel des *protected Branchs* greift (siehe Abbildung 9-17). Ich habe *Include administrators* nicht aktiviert, weswegen die Projekteignerin alle Regeln ignorieren und trotzdem mergen kann.

<sup>12</sup> Das kannst du leicht selbst testen, indem du einen Pull-Request ohne WIP erstellst. Auch dann ist die Prüfung erfolgreich.

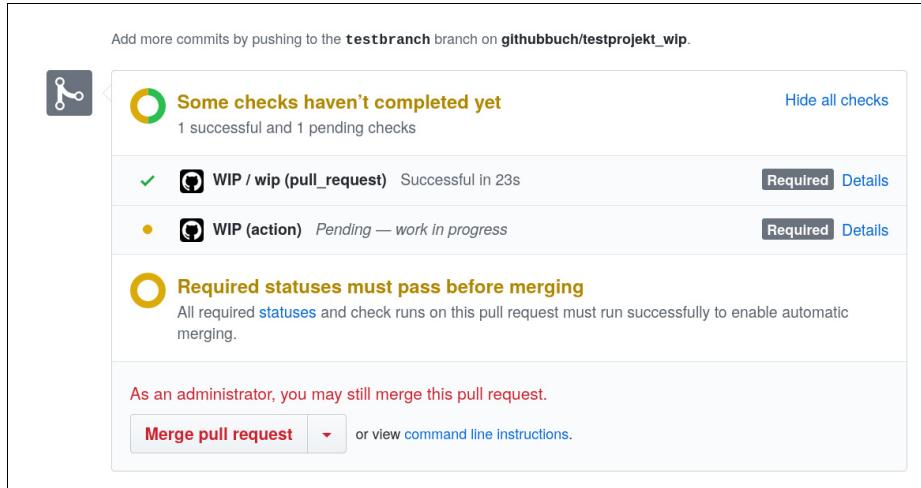


Abbildung 9-17: Protected Branches verhindern das versehentliche Mergen, die Projektleiterin kann es in diesem Fall trotzdem tun.

Damit hast du erfolgreich eine Action aus dem Marketplace installiert und konfiguriert.

## Hinter den Kulissen einer Action

Wenn es dich interessiert, was die Action genau gemacht hat, gibt es dafür ein Protokoll, das du dir auch anschauen kannst. Aus meiner Sicht ist die Menüführung an dieser Stelle jedoch etwas unschön, und deshalb kommentiere ich die Schritte kurz. Sieh dir dazu die Abbildung 9-18 und Abbildung 9-19 an.

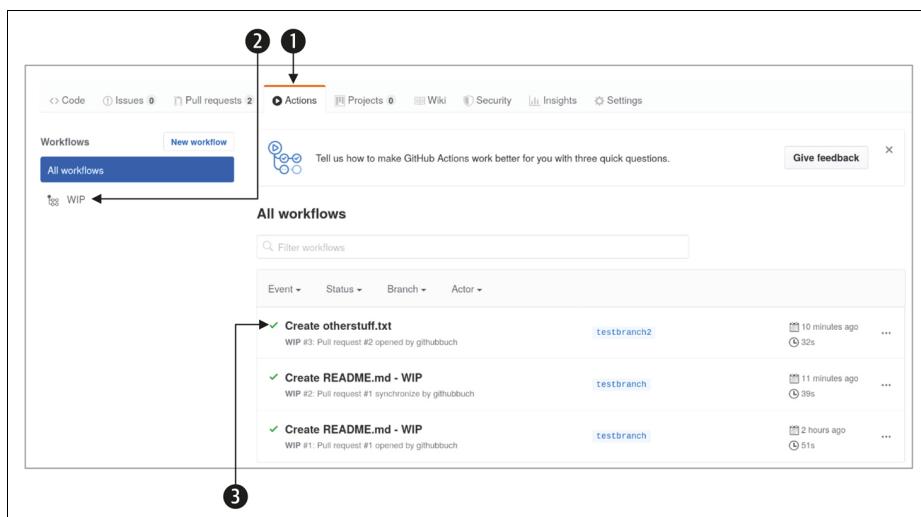


Abbildung 9-18: Das Anschauen einer Action-Protokolldatei benötigt mehrere Klicks (Teil 1).

Gehe auf *Actions* ① und wähle auf der linken Seite unter *Workflows* unseren *WIP* aus ②. Danach wähle rechts den Durchlauf aus, der dich interessiert ③. Anschließend wählst du *wip* aus ④ und kannst über die kleinen Pfeile ⑤ noch weitere Details sichtbar machen. Falls etwas nicht funktioniert, kannstest du hier auf Fehlersuche gehen. Das kann auch bei Actions aus dem Marketplace nützlich sein, falls du die Action nicht richtig konfiguriert hast oder der Action-Entwickler eine fehlerhafte Version hochgeladen hat.

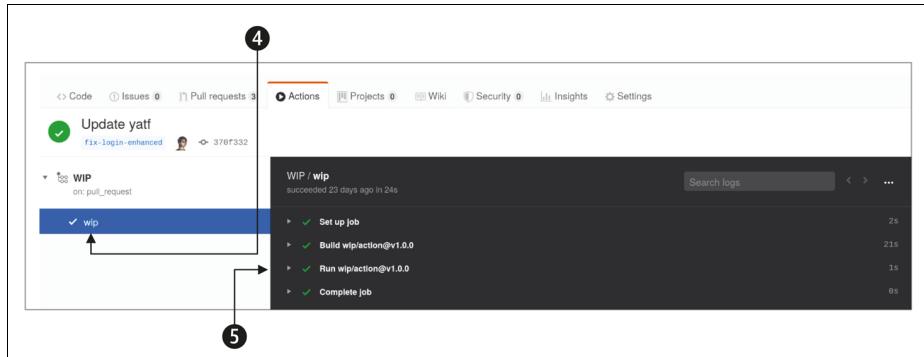


Abbildung 9-19: Das Anschauen einer Action-Protokolldatei benötigt mehrere Klicks (Teil 2).

Auf eine Sache möchte ich dich aufmerksam machen. Du kannst sehen, aus wie vielen Schritten (in GitHub-Sprech: *Steps*) der durchgeführte Workflow besteht. In Abbildung 9-19 sind es vier:

- *Set up job* – Aufsetzen des Workflows, diesen Schritt gibt es in jedem Workflow.
- *Build wip/action@v1.0.0* – eine Aktivität.
- *Run wip/action@v1.0.0* – eine andere Aktivität.
- *Complete job* – Aufräumarbeiten am Ende des Workflows, auch diesen Schritt gibt es in jedem Workflow.

Je nach Aufbau des Workflows können es nur drei oder auch deutlich mehr sein. Du siehst zudem, wie lange GitHub für das Durchführen jeder einzelnen Aktivität gebraucht hat. Im Abschnitt »Mit welchen Kosten muss ich rechnen?« auf Seite 6 in Kapitel 1 wurde erwähnt, dass öffentlichen Repositories eine bestimmte Anzahl an »Action-Minuten« pro Monat kostenlos zur Verfügung gestellt wird. Und bingo, genau hier wird das Thema relevant. Actions verbrauchen nämlich – je nach Komplexität – entsprechende Laufzeit auf den Servern von GitHub.

Die Laufzeit der einzelnen Aktivitäten ist aber auch dann interessant, wenn du planst, eigene Actions zu bauen, z.B. um Fehler zu finden. Wie das geht, schauen wir uns im nächsten Abschnitt genauer an.

# Eine eigene Action erstellen (für Fortgeschrittene)

Bisher haben wir von anderen programmierte Actions und Apps eingesetzt und diese gegebenenfalls noch etwas an unsere Bedürfnisse angepasst. In diesem Abschnitt zeige ich dir, wie du eine eigene Action bauen kannst. Hierfür solltest du ein grundlegendes Verständnis vom Programmieren haben. Ich werde zudem nicht auf alle (Programmier-)Details eingehen, sondern dir nur die grundlegenden Begriffe, Dateien und Werkzeuge zeigen, und zwar in der Art einer Skizze des Erstellungsprozesses, um dir einen ersten Einstieg zu ermöglichen. Das Thema ist nämlich so vielschichtig, dass locker ein eigenes Buch daraus entstehen könnte. Wir schauen uns in diesem Abschnitt Folgendes an:

1. Ein paar grundlegende Begriffe.
2. Die Anatomie einer Action, beispielsweise welche (Programmier-)Möglichkeiten es gibt, Actions zu bauen, und welche Dateien hierfür notwendig sind.
3. Einen Anwendungsfall, den wir gemeinsam implementieren.
4. Wir gehen den Anwendungsfall durch und versuchen zu verstehen, was genau passiert.

## Einige grundlegende Begriffe

Bevor wir mit dem Programmieren der Action loslegen können, wollen wir zunächst ein paar Begrifflichkeiten klären. Wir haben im vorherigen Abschnitt eine Action installiert und benutzt. Korrekterweise muss man sagen: Wir haben einen *Workflow* installiert, der die Action *WIP* genutzt hat. Workflows sind benutzerdefinierte automatisierte Prozesse, die mehrere Actions und/oder Kommandos ausführen können. Stell sie dir als eine Art Rezept für die Automatisierung einer Aufgabe vor.

Man kann beispielsweise festlegen, was der Auslöser für das Ausführen eines Workflows ist (z.B. das Anlegen eines Pull-Requests oder »jeden Samstag um 14 Uhr«), welche Actions und Kommandos in welcher Reihenfolge durchgeführt werden sollen, und man kann sogar Bedingungen festlegen (z.B. »Nur ausführen, wenn der Pull-Request die Datei *README.md* betrifft«).

Damit kennen wir aber bisher nur den großen, allumfassenden Workflow und einen ganz kleinen Baustein darin: die Action. Das ist in etwa so, als würde ich versuchen, dir den Aufbau eines Bürogebäudes zu erklären, und sagen: »Da ist das Haus, und hier ist ein Schreibtisch.« Damit würde ich die einzelnen Büroräume ausblenden. Ähnlich ist es mit Workflow (Gebäude) und Action (Schreibtisch).

Sieh dir hierzu Abbildung 9-20 an:

- Ein *Workflow* (Gebäude) kann ein oder mehrere Jobs (Büros) haben – Jobs werden gleichzeitig abgearbeitet, also Job1 und Job2 laufen parallel. Ein Nacheinander lässt sich aber auch festlegen. Jobs können dabei auf verschiedenen Betriebssystemen laufen, um beispielsweise Softwaretests einmal unter Windows und einmal unter Linux durchzuführen.

- Jeder *Job* besteht aus einem oder mehreren Steps (Büroausstattung).
- Jeder *Step* ist entweder eine Action (Schreibtisch) oder ein Kommando (Schrank).

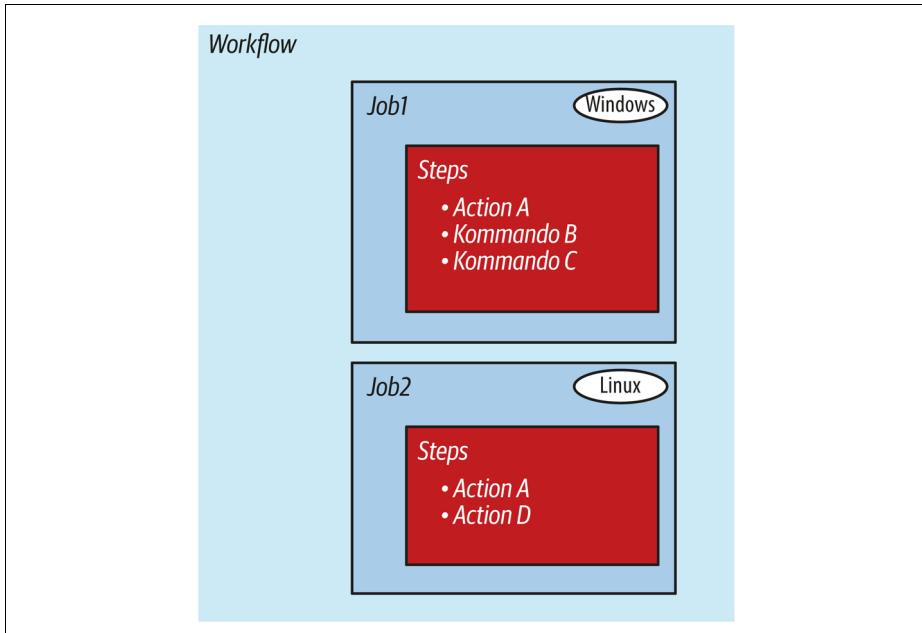


Abbildung 9-20: Actions sind der kleinste Baustein in einem Workflow.



#### Achtung: Limitierungen bei Workflows

Für den Einsatz von Workflows gibt es einige Limitierungen seitens GitHub, die durch den Einwurf kleiner Münzen auch reduziert werden können. Begrenzt sind beispielsweise die (bereits bekannten) Ausführungszeiten von Actions, aber auch die Anzahl der gleichzeitig laufenden Jobs oder wie lange ein Workflow insgesamt laufen darf.<sup>13</sup>

Sobald du mehrere und komplexere Workflows im Einsatz hast, solltest du das im Hinterkopf behalten.

Ohne jetzt alles durchzugehen, wollen wir noch einmal die Workflow-Datei aus dem vorherigen Abschnitt anschauen und einige Details betrachten:

```
name: WIP
on:
  pull_request:
    types: [ opened, synchronize, reopened, edited ]
```

13 <https://docs.github.com/en/free-pro-team@latest/actions/reference/usage-limits-billing-and-administration#usage-limits>

```
jobs:  
  wip:  
    runs-on: ubuntu-latest  
    env:  
      GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}  
    steps:  
      - uses: wip/action@v1.0.0
```

Das Schlüsselwort `on` legt fest, was den Workflow auslöst. Dieser Workflow wird von einem Pull-Request ausgelöst, der entweder neu eröffnet, synchronisiert, wiedereröffnet oder editiert wurde:

```
on:  
  pull_request:  
    types: [ opened, synchronize, reopened, edited ]
```

Es gibt auch noch andere Auslöser für Workflows, wie beispielsweise einen Push auf das Repository. Weitere Informationen dazu sind in der GitHub-Dokumentation zu finden.<sup>14</sup> Unser Workflow hat einen einzigen Job namens `wip`, der auf der letzten Version von Ubuntu-Linux läuft:

```
jobs:  
  wip:  
    runs-on: ubuntu-latest
```



#### Tipp: Zeitplanung ganz einfach

Workflows können auch nach einem Zeitplan ausgeführt werden. Nachfolgendes führt den Workflow alle fünf Minuten im August aus:

```
on:  
  schedule:  
    - cron: '*/5 * * * 8'
```

Wer sich mit der dahinterliegenden Syntax schwertut, findet unter <https://crontab.guru> ein einfache zu bedienendes Werkzeug.

Es gibt nur einen Schritt, nämlich die Action `wip/action@v1.0.0`:

```
steps:  
  - uses: wip/action@v1.0.0
```

Über die Versionsnummer einer Action kannst du übrigens festlegen, welche Version einer Action du in deinem Workflow haben möchtest. Das kann nützlich sein, wenn die neuere Version einer Action noch nicht ganz fehlerfrei ist. Dann kannst du hierüber auf eine ältere Version zugreifen. Es könnten jetzt noch andere Actions als weitere Schritte aufgeführt sein, entweder aus dem Marketplace oder eigene. Das sehen wir uns weiter unten genauer an.

---

<sup>14</sup> <https://docs.github.com/en/free-pro-team@latest/actions/reference/events-that-trigger-workflows>



### Tipp: Berechtigungen für Actions einrichten

Wer für ein Repository festlegen möchte, welche Berechtigungen Actions haben dürfen, kann dies unter *Settings* und dann *Actions* tun (siehe Abbildung 9-21). An derselben Stelle lässt sich auch u.a. einrichten, wie lange Log-Dateien aufgewahrt werden sollen.

Abbildung 9-21: Für jedes Repository lassen sich die Berechtigungen für Actions einstellen

## Anatomie einer Action

Um eigene Actions zu erstellen, gibt es derzeit zwei verschiedene Arten von Actions in GitHub:

- JavaScript-Actions
- Docker-Container-Actions

Das heißt, du kannst eine Action entweder mit JavaScript programmieren, oder du kannst einen sogenannten Docker-Container bauen, in dem du deine Action mit einer beliebigen anderen Programmiersprache programmiert hast. Ich werde dir nachher den Weg über JavaScript zeigen.

Wir haben schon gesehen, dass Actions im jeweiligen Repository unter *.github/workflows* abgespeichert werden und sogenannte YAML-Dateien sind. Die Datei kann dabei grundsätzlich beliebig heißen, nur die Endung *.yml* ist vorgegeben. Es ist möglich, mehrere Workflows zu haben, z.B. einen Workflow, der bestimmte Aktivitäten bei Pull-Requests auslöst, und ein anderer, der bestimmte Aktivitäten bei neuen Issues durchführt. Insofern solltest du deine Workflows<sup>15</sup> auf jeden Fall gut und wiedererkennbar benennen.

<sup>15</sup> Es gibt Menschen, die nennen Workflows auch »Workflöhe« :)

Eine JavaScript-Action, die in einem Workflow laufen soll, besteht aus drei Teilen, zu finden in Tabelle 9-1.

Tabelle 9-1: Notwendige Dateien für eine JavaScript-Action

Datei(en)	Bedeutung	Vorgaben
JavaScript-Dateien	Programmierlogik der Action	in der Regel Dateien mit der Endung <code>.js</code> , gegebenenfalls noch benötigte Erweiterungsmodule
Workflow-Datei	Festlegung von Workflow-Auslöser, die durchzuführenden Aktivitäten (Steps) und Bedingungen	Verzeichnis <code>.github/workflows/</code> , Datei mit der Endung <code>.yml</code>
Action-Datei	Festlegung, welche JavaScript-Dateien ausgeführt werden sollen und welche Ein- und Ausgabeparameter es gibt (sie wird manchmal auch <i>action metadata file</i> genannt)	Verzeichnis <code>.github/actions/ACTIONNAME/</code> , Datei muss zwingend <code>action.yml</code> heißen



#### Tipp: Workflow löschen

Möchte man einen Workflow wieder löschen, geht das (bisher) nur über das Löschen der entsprechenden YAML-Datei aus dem Ordner `.github/workflows`. Die Historie des entsprechenden Workflows bleibt aber weiterhin unter dem Menüpunkt *Actions* erhalten. GitHub hat bereits angekündigt, hier eine Löschfunktion einzurichten.



#### Tipp: Templates für Actions

GitHub bietet einige Templates für Actions und deren Workflows an,<sup>16</sup> beispielsweise für JavaScript-Actions oder Actions, die mit einer bestimmten Python-Version laufen sollen.

## Unseren Anwendungsfall einrichten

Ich habe einen relativ einfachen Anwendungsfall gewählt, der hoffentlich gut nachvollziehbar ist. Jedes Mal, wenn jemand einen neuen Issue erstellt, soll dieser automatisch das Label *new* bekommen. Einen solchen oder ähnlichen Workflow kannst du natürlich auch aus dem Marketplace beziehen.<sup>17</sup> Da der Anwendungsfall aber relativ einfach ist, bietet er sich als selbst gebautes Beispiel zum Üben gut an.

Ein paar Voraussetzungen auf deinem lokalen Rechner brauchen wir, um unseren Anwendungsfall durchspielen zu können:

1. Ein installiertes Git, sollte spätestens seit Kapitel 7 installiert sein.
2. Ein installiertes Node.js<sup>18</sup>.

<sup>16</sup> <https://github.com/actions>

<sup>17</sup> Beispielsweise <https://github.com/marketplace/actions/issue-labeler>.

<sup>18</sup> Siehe Erklärbärbox »Node.js«, alle Informationen zur Installation gibt es hier: <https://nodejs.org/>.

## Node.js

Node.js ist ein Open-Source-Framework für JavaScript-Programme, das 2009<sup>19</sup> das Licht der Welt erblickte und auf GitHub<sup>20</sup> weiterentwickelt wird. Um Node.js besser verstehen zu können, lohnt es sich, zunächst kurz in die Historie von JavaScript einzusteigen.

JavaScript wurde 1995 ursprünglich als Programmiersprache für den Browser entwickelt und war dafür gedacht, die Möglichkeiten von Websites zu erweitern. Beispielsweise können Websites damit deutlich interaktiver gestaltet und Benutzerinteraktionen leichter ausgewertet werden. JavaScript-Programme waren aber auf den Browser als Laufzeitumgebung angewiesen. Node.js bietet eine solche Laufzeitumgebung auch außerhalb des Browsers an.

Ich werde unseren Anwendungsfall ganz innovativ *issuelabeler* nennen. Immer wenn du diesen Textbaustein siehst, weißt du, dass du den Namen anpassen kannst. Ich werde im Beispiel die notwendigen Dateien über GitHub erstellen, du kannst das aber auch über die Konsole machen. Den Inhalt der Dateien findest du entweder in meinem GitHub-Repository<sup>21</sup> oder im Anhang Anhang B. Auf GitHub musst du Folgendes tun:

1. Erzeuge ein neues Repository auf GitHub.
2. Erzeuge die Datei `.github/workflows/issuelabeler.yml` (Inhalt siehe Anhang B).
3. Erzeuge eine Datei `.github/actions/issuelabeler/action.yml` (Inhalt siehe Anhang B) – der Name `action.yml` ist zwingend vorgegeben!
4. Erzeuge eine Datei `.github/actions/issuelabeler/index.js` (Inhalt siehe Anhang B).
5. Optional: Lege das Label *new* auf GitHub an. Wenn du es nicht anlegst, wird es automatisch beim ersten Durchlauf erzeugt, aber gegebenenfalls nicht in der gewünschten Farbe.

Wechsle jetzt auf die Konsole, sofern du dort nicht bereits bist. Wir müssen noch etwas installieren, und das geht am einfachsten über die Konsole. Klone dein Repository und wechsle in das Unterverzeichnis deiner Action:

```
$ git clone https://github.com/USERNAME/REPONAME.git  
$ cd REPONAME/  
$ cd .github/actions/issuelabeler
```

Führe nachfolgende Installationsschritte aus:

```
$ npm init -y  
$ npm install --save @actions/core@1.2.4 @actions/github@2.2.0
```

19 <https://nodejs.dev/learn/a-brief-history-of-nodejs>

20 <https://github.com/nodejs/node>

21 <https://github.com/githhubbuch>

Die Software *npm* (*Node Package Manager*) ist der Standardpaketmanager für Node.js und erlaubt uns, benötigte Bibliotheken zu installieren. Mit dem ersten Befehl initialisieren wir ein neues Projekt, und mit dem zweiten Befehl installieren wir die benötigten Bibliotheken, die wir später in unserem JavaScript-Code nutzen. Wenn du die konkreten Versionsnummern weglässt (@1.2.4 bzw. @2.2.0), wird automatisch die aktuelle Version installiert. Mit neueren Versionen kann es aber passieren, dass unser Code nicht wie gewünscht funktioniert, da sich eventuell Schnittstellen verändert haben. Ich empfehle daher, die von mir angegebenen Versionen zu nutzen.

Ist alles erfolgreich durchgelaufen, sollte eine neue Datei *package.json* erzeugt worden sein (Befehl 1) und ein Verzeichnis *node\_modules* mit einigen Dateien (Befehl 2). Diese müssen wir noch auf GitHub hochladen:

```
$ git add .  
$ git commit -m "NACHRICHT"  
$ git push
```



#### Tipp: Actions auf der Kommandozeile

Falls du generell lieber auf der Kommandozeile unterwegs bist: Das Projekt *actions-cli*<sup>22</sup> hat eine Schnittstelle von der Kommandozeile zu GitHub-Actions implementiert.

The screenshot shows a GitHub Actions workflow named 'labeling\_workflow' triggered on issues. The workflow has a single step labeled 'label\_issue'. The log for this step shows a successful execution. The log details the following steps:

- Set up job
- Run actions/checkout@v1
- Print Hello world
  - Run echo Hello world!
  - Hello world!
- label-step
  - Run ./github/actions/issuelabeler
- Complete job

The entire workflow status is labeled 'succeeded 8 days ago in 3s'.

Abbildung 9-22: Die Action wurde in allen Punkten erfolgreich abgearbeitet.

Sobald du alle Dateien hochgeladen hast, kannst du deine Action ausprobieren. Erstelle einen Issue und warte ab, was passiert. Der Issue sollte nach einer Weile mit dem Label *new* versehen sein. Im Protokoll der Action kannst du nachverfolgen, welche Aktivitäten GitHub durchgeführt hat (das Protokoll hast du im Ab-

22 <https://github.com/remorses/actions-cli>

schnitt »Hinter den Kulissen einer Action« auf Seite 210 bereits kennengelernt, siehe hierzu auch Abbildung 9-22).

## Unseren Anwendungsfall verstehen

Bis hierhin hast du nichts weiter gemacht, als eine Action zu kopieren und auszuprobiieren – was grundsätzlich auch nichts anderes ist, als eine Action über den Marketplace zu installieren. Ich werde dir jetzt Schritt für Schritt zeigen, was genau passiert, sodass du zukünftig befähigt bist, deine eigenen Actions zu bauen.

Wir betrachten uns nacheinander die wichtigen Dateien. Auf die jeweilige Syntax werde ich nicht im Detail eingehen, diese ist sehr gut auf den GitHub-Hilfeseiten dokumentiert.<sup>23</sup> Wir fangen mit der Workflow-Datei `.github/workflows/issuelabeler.yml` an.

### **.github/workflows/issuelabeler.yml**

Das Erste, das wir in der `.github/workflows/issuelabeler.yml` machen, ist, dem Workflow einen Namen zu geben, um ihn wiederzufinden. Der Name ist frei wählbar:

```
name: labeling_workflow
```

Es gilt wie immer: Sprechende Namen sind sinnvoll. Als Nächstes wird der (oder werden die) Auslöser des Workflows festgelegt (wieder eingeleitet durch das Schlüsselwort `on`). Bei uns sind das alle Issues, die neu sind, editiert werden oder wieder eröffnet werden:

```
on:  
  issues:  
    types: [opened, edited, reopened]
```

Falls du beispielsweise den Workflow nur bei komplett neuen Issues laufen lassen möchtest, entferne `edited`, `reopened` aus dem Quelltext. Möchtest du weitere Auslöser haben (etwa Pull-Requests), kannst du sie hier ergänzen, beispielsweise:

```
on:  
  issues:  
    types: [opened]  
  pull_request:  
    types: [opened]
```

Die Einrückungen im Quelltext sind übrigens nicht zufällig. Der Editor auf der GitHub-Weboberfläche schlägt diese genau so vor. Das erhöht natürlich ungemein die Lesbarkeit, wir werden sie daher beibehalten. Als Nächstes wird das Grundgerüst unseres einzigen Jobs festgelegt:

```
jobs:  
  label_issue:  
    runs-on: ubuntu-latest
```

---

23 <https://help.github.com/en/actions/reference/workflow-syntax-for-github-actions>

Der Text `label_issue` ist auch hier ein frei wählbarer Name. Die letzte Zeile legt wieder das Betriebssystem fest, auf dem der Job laufen sollen. Das wird in der GitHub-Terminologie auch als *Runner* bezeichnet (ins Deutsche eher unzureichend übersetzt mit »Läufer«). GitHub bietet neben dem bei uns festgelegten Ubuntu-Linux auch die Job-Abarbeitung über Windows und macOS an.<sup>24</sup> Das kann beispielsweise nützlich sein, wenn du eine Software auf verschiedenen Betriebssystemen kompilieren möchtest, um zu sehen, ob es auf allen klappt.

Nun werden die Steps festgelegt, wir haben dabei drei im Einsatz: eine fremde Action (`actions/checkout@v1`), ein Kommando (`echo Hello world!`) und unsere eigene Action (`./.github/actions/issuelabeler`), jeweils eingeleitet durch einen vorangestellten Strich (-):

```
steps:
  - uses: actions/checkout@v1

  - name: Print Hello world
    run: echo Hello world!

  - name: label-step
    uses: ./.github/actions/issuelabeler
    with:
      repo-token: ${{secrets.GITHUB_TOKEN}}
```

Die fremde Action `actions/checkout@v1` ist eine von GitHub bereitgestellte Action, die den aktuellen Zustand des Repositorys bei Auslösung des Workflows klont. Das soll sicherstellen, dass der Inhalt des Repositorys innerhalb des Jobs zur Verfügung steht und gegebenenfalls nachträglich ausgeführte Commits nicht dazwischenfunken (beispielsweise beim Kompilieren oder beim Testen). Es ist grundsätzlich eine gute Idee, diese Action mit einzubinden.

Das Kommando `echo Hello world!` gibt den Satz »Hello world!« aus, nachlesbar im Protokoll (siehe Abbildung 9-22). Das Kommando hätten wir für unsere Action nicht gebraucht, es ist mehr aus didaktischen Gründen drin :). Es wäre aber auch möglich, andere Kommandos auszuführen, beispielsweise die Installation von zusätzlicher Software in der virtuellen Betriebssystemumgebung, wenn diese für Tests oder Ähnliches gebraucht wird.

Unsere eigene Action `./.github/actions/issuelabeler` hat noch eine kleine Besonderheit, nämlich die Festlegung des Parameters `repo-token`, den wir nachher im JavaScript-Quelltext wieder aufgreifen werden. Der Inhalt des Parameters ist ein von GitHub vorgegebener spezieller Token (deutsch etwa »Wertmarke«)<sup>25</sup>, der es unserer Action später ermöglicht, überhaupt Änderungen am Repository (in unse-

---

<sup>24</sup> Eine Übersicht der Möglichkeiten gibt es hier: <https://help.github.com/en/actions/reference/virtual-environments-for-github-hosted-runners>.

<sup>25</sup> Details dazu sind hier zu finden: [https://help.github.com/en/actions/configuring-and-managing-workflows/authenticating-with-the-github\\_token](https://help.github.com/en/actions/configuring-and-managing-workflows/authenticating-with-the-github_token).

rem Fall: Editieren der Issue-Labels) vornehmen zu dürfen. Es sind aber auch andere Parameter denkbar, beispielsweise der Username, der den Workflow aufruft, oder der aktuelle Repository-Name.<sup>26</sup>

### .github/actions/issuelabeler/action.yml

Wir wechseln jetzt die Datei, und zwar gehen wir zu unserer Action-Metadatendatei `.github/actions/issuelabeler/action.yml`. Diese Datei ist inhaltlich wenig spektakulär. Wir legen dort fest, mit welcher Node.js-Version wir arbeiten wollen und wo der Einstiegspunkt (*Entry point*) für den Quellcode ist, in unserem Fall die Datei `index.js`:

```
runs:  
  using: "node12"  
  main: "index.js"
```

Du kannst dich hier noch als Autorin verewigen oder Variablen und ihre Standardwerte festlegen. Eine Übersicht der Möglichkeiten gibt wieder die GitHub-Hilfe<sup>27</sup>.

### .github/actions/issuelabeler/index.js

Erneut wechseln wir die Datei und schauen uns die letzte im Bunde an, unsere JavaScript-Datei `.github/actions/issuelabeler/index.js`. Zu Beginn binden wir zwei Bibliotheken ein, die wir zuvor über die Konsole installiert haben<sup>28</sup>. Diese helfen uns, bestimmte Aktivitäten leichter durchzuführen:

```
const core = require("@actions/core");  
const github = require("@actions/github");
```

Danach legen wir eine Reihe von Konstanten fest, die wir benötigen: beispielsweise wie das Label heißen soll und was wir vergeben wollen. Aber auch unserem `repo-token` aus der Workflow-Datei begegnen wir hier wieder:

```
const [owner, repo] = process.env.GITHUB_REPOSITORY.split("/");  
const token = core.getInput("repo-token");  
const issue = github.context.payload.issue.number;  
const octokit = new github.GitHub(token);  
  
const label = ["new"];
```

---

26 Eine Übersicht findest du hier: <https://help.github.com/en/actions/reference/context-and-expression-syntax-for-github-actions>.

27 <https://help.github.com/en/actions/building-actions/metadata-syntax-for-github-actions>

28 <https://github.com/actions/toolkit/tree/master/packages/core> und  
<https://github.com/actions/toolkit/tree/master/packages/github>

Wenn du möchtest, dass ein neuer Issue mehrere Labels bekommt, kannst du const label erweitern, beispielsweise so:

```
const label = ["new", "brand-new"];
```

Die eigentliche »Magie« geschieht aber in diesem Quelltextabschnitt:

```
// Label issue
const response = await octokit.issues.addLabels({
  owner,
  repo,
  issue_number: issue,
  labels: label
});
```

Hier verwenden wir eine von GitHub bereitgestellte Funktion addLabels, um dem Issue sein neues Label zu verpassen.<sup>29</sup>



#### Achtung: Actions bei geforkten Repositories

Wenn Aktivitäten aus einem geforkten Repository einen Workflow auf dem Originalprojekt anstoßen (wie beispielsweise die Aktivität Pull-Request), funktionieren viele Actions aufgrund von fehlenden Schreibrechten nicht. Das ist (derzeit) so gewollt, wenngleich dieser Zustand von der Open-Source-Community durchaus kritisch gesehen wird.<sup>30</sup> GitHub hat das mittlerweile auch erkannt. Eine entsprechende Anpassung steht auf ihrer Roadmap und ist gegebenenfalls zur Drucklegung des Buchs bereits implementiert.

Das soll es auch erst einmal gewesen sein mit unserer eigenen Action. Für intensive Recherche und eigene Experimente empfehle ich dir auf jeden Fall, bei anderen Actions (z.B. aus dem Marketplace) zu »spicken«<sup>31</sup> und dich inspirieren zu lassen. Bevor wir die Welt der Actions verlassen, möchte ich dir im nächsten Abschnitt noch zeigen, welche Möglichkeiten GitHub für geheime, aber benötigte Informationen im Quelltext (beispielsweise Passwörter) anbietet.

## Passwörter geheim halten – GitHub Secrets

Selbst bei Open Source gibt es Dinge, die man nicht unbedingt mit aller Welt teilen möchte. In der Regel sind das Passwörter beispielsweise für den Zugang zu einer Datenbank, aber auch E-Mail-Adressen oder andere Informationen sind denkbar. Dafür gibt es bei GitHub die sogenannten *Secrets* (siehe Abbildung 9-23).

29 Weitere Funktionen sind hier zu finden: <https://octokit.github.io/rest.js/v18>.

30 Unter anderem auch von mir. Ich habe tagelang damit zugebracht, einen Workflow mit einer von mir geschriebenen Action für ein Open-Source-Projekt zu debuggen, bis ich festgestellt habe: »It's not a bug, it's a feature!«

31 »Learning by Abgucken«

The screenshot shows the GitHub repository settings page for a specific repository. The top navigation bar includes links for Code, Issues (2), Pull requests (0), Actions, Projects (0), Wiki, Security, Insights, and Settings. The Settings tab is active. On the left, a sidebar menu lists Options, Manage access, Branches, Webhooks, Notifications, Integrations, Deploy keys, Secrets (which is selected and highlighted in orange), and Actions. The main content area is titled "Secrets". It contains a note stating that secrets are environment variables encrypted and only exposed to selected actions, available to anyone with collaborator access. It also notes that secrets are not passed to workflows triggered by pull requests from forks. A table lists a single secret named "DatenbankPasswort" with a "Remove" button. A link "Add a new secret" is at the bottom of the table.

Abbildung 9-23: GitHub Secrets findet man beim jeweiligen Repository unter den Settings.

Hier kannst du geheime Informationen speichern und über den dazugehörigen Namen (auch Schlüssel genannt) ansprechen. Das bedeutet, in einem öffentlich einsehbaren Quelltext steht dann nicht mehr:

```
password: 89Gehe!m
```

sondern:

```
password: ${{ secrets.DatenbankPasswort }}
```

Wobei DatenbankPasswort der Name des entsprechenden Geheimnisses ist. Diese Syntax hast du schon im Rahmen der eigenen Action im Abschnitt zuvor kennengelernt. Jedes Repository kann bis zu 100 solcher Geheimnisse beinhalten, die Namen müssen jeweils eindeutig sein und am besten auch noch so sprechend, dass du sie leicht wiedererkennst.



#### Mehr Infos über den Ablauf einer Action

Die Informationen, die GitHub standardmäßig zu den einzelnen Actions ausspuckt, reichen nicht? Unter *Secrets* lassen sich die beiden Secrets *ACTIONS\_RUNNER\_DEBUG* und *ACTIONS\_STEP\_DEBUG* aktivieren. Sobald man diese auf *true* stellt, erhält man weitere Informationen.

Wir verlassen jetzt den Bereich von Actions und Apps und schauen uns an, was GitHub noch zu bieten hat.



## KAPITEL 10

# Pimp my Repo – Weitere GitHub-Features

Dieses Kapitel stellt weitere Features von GitHub vor, die im ersten Schritt zwar nicht »überlebenswichtig« sind, aber die weitere Arbeit erleichtern oder sogar verschönern können.



### Am Ende des Kapitels kannst du ...

- aus deinem Repository automatisch eine Website erstellen lassen (*GitHub Pages*).
- diese Website mit vorgefertigten Themes verschönern und eigene Anpassungen vornehmen, wie beispielsweise eine Navigation hinzufügen.
- Templates für Repositories anlegen und nutzen.
- dir mithilfe von *Projektboards* einen Überblick über die vielfältigen anstehenden Aufgaben deines Projekts verschaffen.

## Websites aus GitHub generieren (GitHub Pages)

Wer ein cooles Projekt hat, möchte dieses der Welt auch irgendwann mitteilen. Wie du ja vielleicht schon selbst mitbekommen hast, sind Projektseiten auf GitHub – selbst mit einer vorbildlichen *README.md* – optisch nicht gerade das Gelbe vom Ei. Du kannst jetzt natürlich auf dem üblichen Weg eine aussagekräftige Homepage für dein Projekt erstellen, aber es geht auch einfacher (und eventuell sogar schneller), und zwar direkt über GitHub mit den sogenannten *GitHub Pages*<sup>1</sup>.



### Achtung: GitHub und Git LFS

Zur Erinnerung (siehe Abschnitt »Git LFS einrichten« auf Seite 158, Kapitel 7): Git LFS funktioniert nicht zusammen mit GitHub Pages.

<sup>1</sup> <https://pages.github.com/>

## GitHub Pages einrichten

Wir testen das am besten einfach. Nimm ein bestehendes oder neues Repository, in dem eine (ein bisschen) gefüllte *README.md* existiert. Gehe auf die *Settings* beim entsprechenden Repository und wähle *Options* aus. Wenn du dort nach unten scrollst, kannst du unter der Überschrift *GitHub Pages* diese einschalten, indem du bei *Source* (deutsch »Quelle«) in dem Drop-down eine Option auswählst (siehe Abbildung 10-1). Du findest hier die Auswahl:

- *master branch* – der master-Branch als Quelle für GitHub Pages.
- *master branch /docs folder* – wie oben, beschränkt sich aber auf den Ordner *docs*. Solltest du bisher keinen solchen Ordner angelegt haben, ist diese Option nicht auswählbar.
- *None* – deaktiviert GitHub Pages (Starteinstellung).

The screenshot shows the 'GitHub Pages' settings for a repository. At the top, it says 'GitHub Pages' and describes its purpose: 'GitHub Pages is designed to host your personal, organization, or project pages from a GitHub repository.' Below this, there's a 'Source' section which is currently disabled. It asks the user to select a source to enable GitHub Pages. A dropdown menu is open, showing 'None' as the selected option. There's also a link to 'Learn more'. Below the source section is a 'Theme Chooser' section, which also asks the user to select a theme using the master branch. A button labeled 'Choose a theme' is present. The entire interface is contained within a light gray box with rounded corners.

Abbildung 10-1: GitHub Pages richtet man im jeweiligen Repository unter »Settings« ein.



### Tipp: GitHub Pages auf eigenem Branch

Wenn du die Website lieber auf einem eigenen Branch haben möchtest, um beispielsweise Softwarecode und Website irgendwie zu trennen, kannst du einen neuen Branch *gh-pages* anlegen. Danach wählst du diesen in den *Settings* als Quelle aus. Andere Branchnamen funktionieren aber nicht!

Den darunterliegenden *Theme Chooser* schauen wir uns gleich noch an. Wähle die obersten Option, den *master branch*, und die Seite lädt neu. Jetzt zeigt dir GitHub an, wo deine GitHub Page veröffentlicht wurde, in der Regel ist das `USERNAME.github.io/REPONAME`. Dorthin wechseln wir jetzt mal, und du solltest den Inhalt deiner *README.md* sehen (siehe Abbildung 10-2). Gegebenenfalls braucht GitHub ein paar Minuten, bevor du unter der Webadresse etwas angezeigt bekommst, also nur Geduld.

Die von GitHub erstellte Website ist eine sogenannte Projekt-Website, die auf einem (Projekt-)Repository aufbaut. Davon kannst du grundsätzlich beliebig viele haben, pro Repository eine. Du kannst aber auch eine User-Website einrichten, von der du genau eine pro User-Account haben kannst. Erstelle dafür ein Repository nach dem Schema `USERNAME.github.io`. Die Website ist dann automatisch über <https://USERNAME.github.io> erreichbar. Besonders groß ist der Unterschied zwischen den beiden nicht, abgesehen von der URL. Im Nachfolgenden werde ich immer von einer Projekt-Website ausgehen.

The screenshot shows a GitHub Pages site with the following structure:

- testprojekt\_ghpages**
- Reponame**
- Willkommen in meinem Repo**  
Viel wichtiger Text, der beschreibt, was man hier so sehen und machen kann.
- Meine ToDo-Liste**  
Hier stehen alle meine Todos:
  - GitHub-Pages einrichten
  - GitHub-Pages schön machen

Abbildung 10-2: Mit GitHub Pages lassen sich leicht Websites aus einer `README.md` erstellen.

Schön ist die Darstellung der Seite nicht, aber es fehlt zumindest das ganze ablenkende »GitHub-Klimbim« drum herum. Wir sind jedoch anspruchsvoll, was das Design anbelangt, und deswegen schauen wir im nächsten Abschnitt, wie wir die Seite etwas »aufhübschen« können.

## GitHub Pages verschönern mit dem Theme Chooser

Um die Website zu verschönern, können wir den zuvor ignorierten *Theme Chooser* nutzen, der einige Vorlagen bietet. Klicke auf *Choose a theme*, und du landest in der Theme-Auswahl (siehe Abbildung 10-3).

Unter dem GitHub-Menüband siehst du die unterschiedlichen Themes (*Cayman*, *Slate*, *Merlot* etc.)<sup>2</sup>, die du dir durch Anklicken anschauen kannst. Dargestellt werden sie in einer Art Vorschau im unteren Bereich (im Beispiel steht dort *Cayman theme*). Wähle eine Vorlage aus, die dir gefällt, klicke auf *Select theme* und lade deine GitHub Page `USERNAME.github.io/REPONAME` neu. Auch hier kann es eventuell ein paar Minuten dauern, bis die Änderungen sichtbar werden. Die Seite sollte jetzt im gewählten »Look« erstrahlen.

2 Eine Liste aller unterstützten Themes ist hier zu finden: <https://pages.github.com/themes/>.

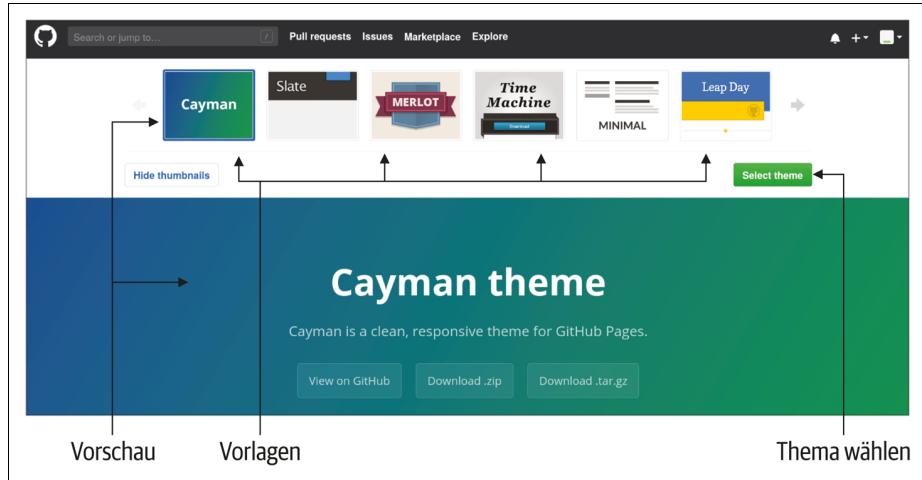


Abbildung 10-3: GitHub bietet verschiedene Vorlagen für GitHub Pages an.

Wenn du nun in dein Repository schaust, wirst du unter *Code* eine neue Datei namens *\_config.yml* vorfinden. Es handelt sich hierbei um eine sogenannte YAML-Datei (siehe auch Erklärbarbox »YAML«). GitHub nutzt die Software Jekyll<sup>3</sup> für die Webseitererstellung, und diese Datei ist die dafür notwendige Konfigurationsdatei. Wenn du sie öffnest, findest du dort das von dir gewählte Theme (in meinem Fall der technische Name für das Cayman-Theme):

```
theme: jekyll-theme-cayman
```

Du kannst diese Datei zukünftig auch direkt editieren, wenn du das Theme wechseln möchtest. Dafür musst du natürlich dessen korrekten Namen kennen. Generell wird diese Datei verwendet, um deine Website zu konfigurieren. Du kannst beispielsweise den Titel der Website über die Konfiguration festlegen, indem du die *\_config.yml* wie folgt anpasst:

```
theme: jekyll-theme-cayman
title: Meine wunderbare Website!
```

## Jekyll

Jekyll ist sowohl eine Romanfigur<sup>4</sup> als auch eine Software. GitHub nutzt natürlich die Software für das Generieren von Webseiten. Jekyll generiert statische Webseiten aus HTML- oder Markdown-Dateien. Du legst einmalig ein Layout fest, und wie durch Zauberei entsteht aus einer einfachen Markdown-Datei eine vollwertige, im besten Fall gut aussehende Website. Jekyll ist nicht an GitHub gebunden und

3 <https://jekyllrb.com/>

4 Siehe auch »Der seltsame Fall des Dr. Jekyll und Mr. Hyde« von Robert Louis Stevenson.

kann auch unabhängig davon genutzt werden. Mit Jekyll lassen sich nicht nur Websites, sondern auch Blogs sehr einfach realisieren.

Du hast jetzt gesehen, dass sich mithilfe von GitHub Pages und dem *Theme Chooser* sehr leicht einigermaßen hübsche Websites erstellen lassen. Im nächsten Abschnitt wollen wir ein bestehendes Theme an unsere Bedürfnisse anpassen.



#### Tipp: Startseite von GitHub Pages

Standardmäßig wird die *README.md* als Startseite deiner Website genommen. Wenn du aber eine Datei namens *index.md* erstellst, ist diese zukünftig die Startseite.

## GitHub Pages ausbauen – die Navigation einrichten

Im vorherigen Abschnitt haben wir aus unserer *README.md* eine einzelne Webseite mit einem festgelegten Theme erstellen lassen. Websites bestehen in der Regel aber natürlich aus mehreren Seiten. In diesem Abschnitt wollen wir eine weitere Seite erzeugen und mit einer zentral pflegbaren Navigation erreichbar machen.

### Weitere Seiten und die Navigation erstellen

Eine weitere Seite zu erzeugen, ist kinderleicht: Lege eine weitere Datei an, beispielsweise *Kontakt.md*. Hilfreich ist, wenn du zumindest etwas Text, etwa eine Überschrift, in die Datei schreibst. Als Website ist sie dann über die URL nach dem Schema *USERNAME.github.io/REPONAME/DATEINAME* erreichbar, in diesem Fall also *USERNAME.github.io/REPONAME/Kontakt*. Für das folgende Beispiel werde ich noch eine weitere Seite namens *Seite.md* erzeugen.

Eine Navigation zu bauen, die alle Seiten miteinander verbindet, ist ein wenig aufwendiger, aber nicht schwierig. Für die Navigation an sich nutzen wir HTML<sup>5</sup>. Falls du (noch) kein HTML kannst: Keine Sorge, das Beispiel ist rudimentär und lässt sich problemlos an deine Bedürfnisse anpassen.

Erzeuge in deinem Repository eine Datei *\_includes/nav.html* (Unterstrich beim Verzeichnis beachten!). Du weißt ja bereits, dass GitHub keine leeren Verzeichnisse erzeugen kann, deshalb werden wir die Datei nebst Verzeichnis in einem Rutsch erstellen. Befülle die Datei mit folgendem Inhalt:

```
<p style="text-align: center; background-color: white;">
  <a href="{{site.baseurl}}>Home</a> |
  <a href="{{site.baseurl}}/Kontakt">Kontakt</a> |
  <a href="{{site.baseurl}}/Seite.html">Andere Seite</a> |
</p>
```

<sup>5</sup> Das steht für *Hypertext Markup Language* und ist eine Auszeichnungssprache zum Erzeugen von Webseiten.

In der ersten Zeile legen wir ein paar (optionale) Stilelemente fest (alles was hinter `style` folgt), und in den nächsten drei Zeilen setzen wir die Links für unsere drei erzeugten Seiten `README.md`, `Kontakt.md` und `Seite.md`. Bitte beachte den unterschiedlichen Aufbau der beiden unteren Links `/Kontakt` und `/Seite.html`. Darauf werde ich später noch kurz eingehen.

Die Idee ist, die gesamte Navigation in einer Datei zu bündeln. Du kannst dann diese Navigationsdatei in alle deine Seiten einbinden, siehe Abbildung 10-4. Ändert sich etwas an der Navigation, musst du nur die `nav.html` anpassen, und alle anderen Seiten werden automatisch aktualisiert.

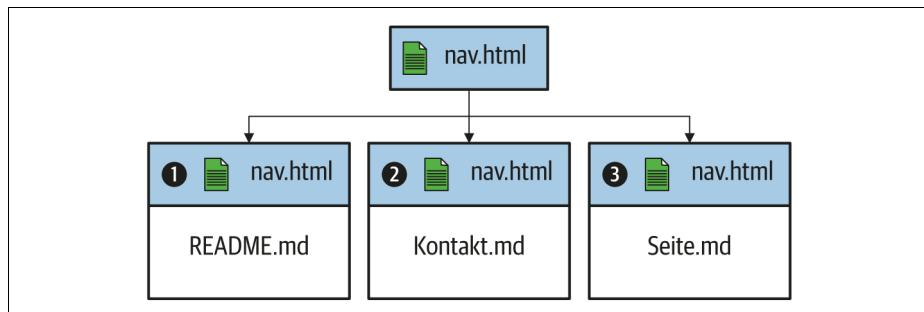


Abbildung 10-4: Alle drei Seiten nutzen dieselbe Navigationsdatei.

## Die Navigation einbinden

Bis hierhin haben wir nur festgelegt, wie die Navigation aussehen soll, wir haben sie aber noch nicht benutzt. Um sie einzusetzen, müssen wir das Layout der Seite entsprechend anpassen. Da wir ein Theme von GitHub nutzen, müssten wir also dort Anpassungen vornehmen. Wie du dir vielleicht denken kannst, haben wir im Repository des Themes keine Schreibrechte, es geht aber zum Glück auch ohne.

Ich zeige dir anhand des Themes *Cayman*, wie du das Layout für deine Website anpassen kannst. Wir kopieren nämlich einfach die benötigte Datei vom Theme-Repository in das unsrige:

1. Suche das Repository des Themes. Das geht am einfachsten über die Übersicht aller Themes.<sup>6</sup>
2. In dem Theme-Repository suchst du die Datei `default.html` im Ordner `_layouts` und kopierst dessen Inhalt (das geht am einfachsten über den Button `Raw`).
3. Gehe auf dein Repository und erstelle eine Datei nebst Verzeichnis, `_layouts/default.html`.
4. Kopiere den Inhalt aus dem Theme-Repository in deine gerade erstellte Datei hinein und speichere.

<sup>6</sup> <https://pages.github.com/themes/> und dort »Cayman« anklicken:  
<https://github.com/pages-themes/cayman>

Die Datei `default.html` ist die Standardlayoutdatei des genutzten Themes. Das bedeutet, dass alle Webseiten, die wir aus Markdown-Dateien generieren, dieses Standardlayout als Grundlage nehmen. Diese Datei liegt jetzt in unserem eigenen Repository, auf dem wir Schreibrechte haben. GitHub wird zukünftig diese Datei für das Generieren der Website verwenden. Jetzt können wir Anpassungen vornehmen und unsere zuvor erstellte Navigation einbauen. Öffne die Datei `_layouts/default.html` und suche nach folgender Stelle:

```
<main id="content" class="main-content" role="main">
  {{ content }}
```

Sie dürfte relativ weit unten zu finden sein.<sup>7</sup> Texte in geschweiften Klammern (hier: `{{ content }}`) sind in der Regel Befehle oder Platzhalter für den Zusammenbau der späteren Webseite. Wir wollen an dieser Stelle eine neue Zeile einfügen:

```
<main id="content" class="main-content" role="main">
  {% include nav.html %}
  {{ content }}
```

Damit inkludieren wir unsere zuvor erstellte Navigationsdatei in jede Seite, die aus der `default.html` heraus generiert wird. Solange wir das nicht explizit anders angeben, ist dies im Augenblick jede generierte Seite. Info am Rande: Wenn wir die Datei `nav.html` nicht in dem Ordner `_includes` abgelegt hätten, würde GitHub sie nicht finden. Lädst du jetzt deine Website neu, solltest du etwas Ähnliches wie das in Abbildung 10-5 Gezeigte sehen.

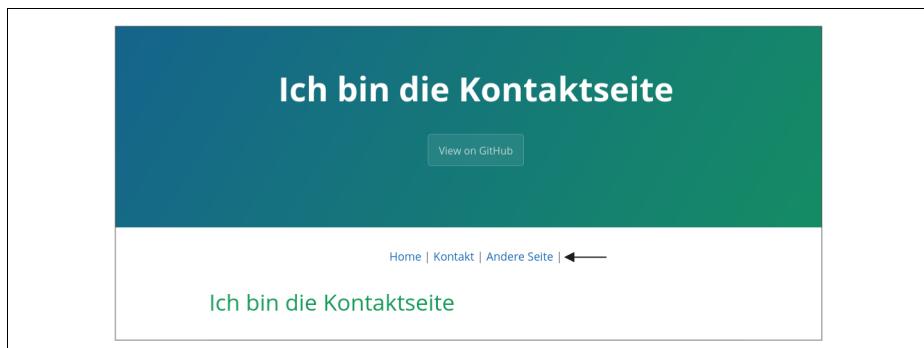


Abbildung 10-5: Mit einigen Handgriffen lässt sich eine Navigation hinzufügen.

## Die Navigation debuggen

Wenn du deine Navigation ausprobierst, wird an einer Stelle ein Fehler auftauchen, und zwar immer dann, wenn du auf den Link `Andere Seite` klickst. GitHub wird dir sagen, dass es diese Seite nicht finden kann. Wenn wir uns den Aufbau der Links anschauen, sehen wir auch, dass GitHub eine Datei `Seite.html` sucht:

<sup>7</sup> Sollte es diese Zeilen gar nicht mehr geben: Auch nicht schlimm, such dir eine Stelle, von der du glaubst, dass sie ebenfalls geeignet wäre – ansonsten macht »Versuch kluch« :).

```
<a href="{{site.baseurl}}/Kontakt">Kontakt</a> |  
<a href="{{site.baseurl}}/Seite.html">Andere Seite</a> |
```

Wir haben aber nur eine *Seite.md* angelegt. Ich habe diesen Fehler absichtlich eingebaut, um dir zu zeigen, dass du auch Seiten mit der Endung *.html* verlinken kannst, selbst wenn diese eigentlich Markdown-Dateien sind. Der Fehler ist »heilbar« auf zwei Arten:

1. Du änderst den Link innerhalb der *nav.html* und entfernst das *.html* – genau wie bei *Kontakt*.
2. Du editierst die Seite *Seite.md*, indem du ganz zu Beginn der Datei zwei dreifach gestrichelte Linien setzt:

```
---
```

```
---
```

```
# Andere Seite/weiterer Inhalt
```

Diese Striche sorgen dafür, dass Jekyll die *Seite.md* auf jeden Fall bearbeitet und eine *Seite.html* verfügbar macht. Danach funktionieren alle unsere definierten Links in der Navigation. Darüber hinaus hast du so die Möglichkeit, weitere Parameter einzusetzen, beispielsweise:

```
---
```

```
title: Sehr coole Seite
```

```
---
```

```
# Andere Seite/weiterer Inhalt
```

Du kannst sogar eigene Parameter festlegen und diese dann in der Seite verwenden:

```
---
```

```
food: Pizza
```

```
---
```

```
Mein Lieblingsessen: {{ page.food }}
```

Das Ganze nennt sich *YAML Front Matter* (deutsch »YAML-Titelei«<sup>8</sup>) und bietet dir weitere Möglichkeiten, dein Layout zu individualisieren. Falls du hier tiefer ein tauchen möchtest, empfehle ich dir die Dokumente auf der Jekyll-Website.<sup>9</sup>

### Cooler Gimmick: 404-Seite erzeugen

Eine coole Kleinigkeit machen wir noch zum Abschluss. Wir erzeugen eine individuelle Fehlerseite, die immer dann angezeigt wird, wenn jemand versucht, auf deiner Website eine nicht vorhandene Unterseite aufzurufen.

Eine solche Fehlerseite wird auch »404« genannt, weil das genau der Fehlercode ist, den der Browser in einem solchen Fehlerfall anzeigt.<sup>10</sup> Gehe in das Repository, das die 404-Seite bekommen soll, erzeuge eine Datei *404.html* oder *404.md* und

<sup>8</sup> Titelei ist ein Begriff aus dem Buchwesen. Damit werden Seiten bezeichnet, die dem eigentlichen Inhalt vorangestellt sind.

<sup>9</sup> <https://jekyllrb.com/docs/front-matter/>

<sup>10</sup> Für Interessierte zum tieferen Einsteigen: <https://de.wikipedia.org/wiki/HTTP-Statuscode>.

entscheide dich für einen Text, der beim Aufruf einer fehlenden Unterseite angezeigt werden soll.

Solltest du dich für *404.md* entschieden haben, musst du erneut YAML Front Matter zu Beginn der Datei einsetzen, damit Jekyll diese Seite auf jeden Fall bearbeitet:

```
---
```

```
permalink: /404.html
```

```
--
```

Abspeichern, ausprobieren und freuen!

Du hast jetzt eine Website aus deinem Repository erzeugt sowie ein vorgefertigtes Theme benutzt und es angepasst. Was aber machst du, wenn dir die Themes von GitHub nicht gefallen? Zum Glück gibt es andere Möglichkeiten.

## GitHub Pages – weitere Themes

GitHub Pages kann man auch mit anderen Themes nutzen als mit denen, die GitHub als Vorlage bereithält. Für deren Einsatz wird teilweise ein tieferes Verständnis von Jekyll, der Programmiersprache Ruby und Ruby-Bibliotheken (sogenannte *Ruby Gems*)<sup>11</sup> benötigt, sodass ich hier nur skizzenhaft so weit darauf eingehen werde, dass du einen ersten Startpunkt hast für eigene tiefer gehende Recherchen. Um deine Website individueller zu gestalten, hast du die folgenden Möglichkeiten:

- Remote Themes
- Theme-Repositories forken
- eigene Themes bauen

Diese gehen wir nacheinander kurz durch.

### Remote Themes

Die Theorie besagt, dass sich sogenannte Remote Themes über die Konfigurationsdatei *\_config.yml* relativ einfach einrichten lassen – ähnlich wie die von GitHub vorgegebenen Themes. Habe ich ein Repository gefunden, das ein ansprechendes Remote Theme anbietet, kann ich die Konfigurationsdatei um folgenden Zusatz ergänzen (Unterstrich \_ beachten!):

```
remote_theme: OWNER/REPONAME
```

Ein (halbwegs funktionierendes) Beispiel, das man auch auf den Hilfeseiten von GitHub findet, ist folgendes:

```
remote_theme: benbalter/retlab
```

In der Praxis haben die Remote Themes bei mir aber leider meistens nicht sonderlich gut funktioniert, obwohl es diese Funktion bereits seit 2017 gibt. Ich persön-

---

<sup>11</sup> Nur fürs Protokoll: Jekyll ist ein sogenanntes Ruby Gem.

lich würde daher eher davon abraten.<sup>12</sup> Wo finde ich solche Remote Themes? Eine gute Möglichkeit ist, über die Topics (kennen wir aus Abschnitt »Besser gefunden werden auf GitHub: Topics« auf Seite 102 in Kapitel 6) nach jekyll-theme zu suchen,<sup>13</sup> ansonsten gibt es auch viele Websites, die Jekyll-Themes sammeln.<sup>14</sup>

### Theme-Repositories forken/klonen

Als Variante zwei gibt es die Möglichkeit, Theme-Repositories zu forken oder zu klonen und dann entsprechend anzupassen. Für das Anpassen ist aber meist die lokale Installation von Jekyll, Ruby, Ruby Gems und gegebenenfalls anderen Komponenten nötig.

In Abbildung 10-6 sieht man eine typische Installationsanleitung für ein Theme, hier exemplarisch für das Theme *jekyll-uno*.<sup>15</sup> Dieses Beispiel zeigt, wie aufwendig das werden kann, und macht deutlich, dass du für das Nutzen von anderen Themes zumeist dein Repository lokal bearbeiten und auch einige Softwarekomponenten installieren musst.

#### Install and Test

1. Download or clone repo `git clone git@github.com:joshgerdes/jekyll-uno.git`
2. Enter the folder: `cd jekyll-uno/`
3. If you don't have bundler installed: `gem install bundler`
4. Install Ruby gems: `bundle install`
5. Start Jekyll server: `bundle exec jekyll serve --watch`

Access via: <http://localhost:4000/jekyll-uno/>

Abbildung 10-6: Typische Installationsanleitung für ein Theme außerhalb der von GitHub bereitgestellten Vorlagen.

### Eigene Themes bauen

Du kannst natürlich auch eigene Themes bauen. Hierfür ist ebenfalls wieder die lokale Installation von Jekyll, Ruby, Ruby Gems und eventuell anderen Komponenten notwendig. Wenn du dich intensiver mit dem Erstellen von Jekyll-Themes beschäftigen möchtest, empfehle ich dir eine entsprechende Recherche.<sup>16</sup> Viel Erfolg!

12 Ich hatte bei maximal 10% der Themes überhaupt das Glück, dass irgendetwas funktioniert hat.

13 <https://github.com/topics/jekyll-theme>

14 z.B. <https://jekyll-themes.com/free/>

15 <https://github.com/joshgerdes/jekyll-uno>

16 Starte am besten hier: <https://jekyllrb.com/docs/themes/>.

# Angriff der Klone – Repo-Templates anlegen

Wir haben schon eine Menge unterschiedlicher Vorlagen kennengelernt: von Vorlagen für Issues und Pull-Requests (unter »Vorlagen für Issues« auf Seite 75 und »Vorlagen für Pull-Requests« auf Seite 79, beide in Kapitel 4) bis hin zu Vorlagen für Webseitenlayouts (vorheriger Abschnitt). Wenn du häufig neue Repositories anlegst und zudem noch eine bestimmte Datei- und Verzeichnisstruktur benötigst, lohnt es sich, auch für Repositories eine Vorlage zu erstellen.

Betrachte hierzu einmal Abbildung 10-7. Das ist im Grunde nichts anderes, als ein Repository anzulegen, das mit *Template* markiert wird ① und einen zusätzlichen Button enthält ②. Jeder mit Leseberechtigung für ein solches Template-Repository kann es als Vorlage verwenden.

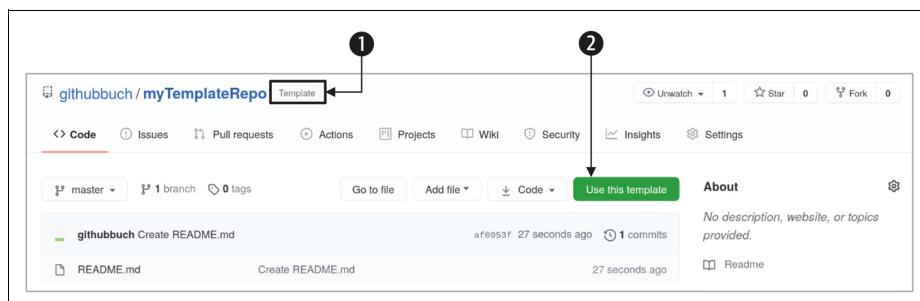


Abbildung 10-7: Wenn du auf einem Template-Repo unterwegs bist, kannst du von dort direkt ein neues Repo mit dem entsprechenden Template erstellen.

Ein Template beinhaltet die Datei- und Verzeichnisstruktur sowie die Actions. Nicht übernommen werden Issues, Pull-Requests, Branches oder das Wiki. Du kannst selbst ein Template erstellen, indem du in einem Repository, das du als Vorlage verwenden willst, unter *Settings* einen Haken bei *Template repository* setzt (siehe Abbildung 10-8).

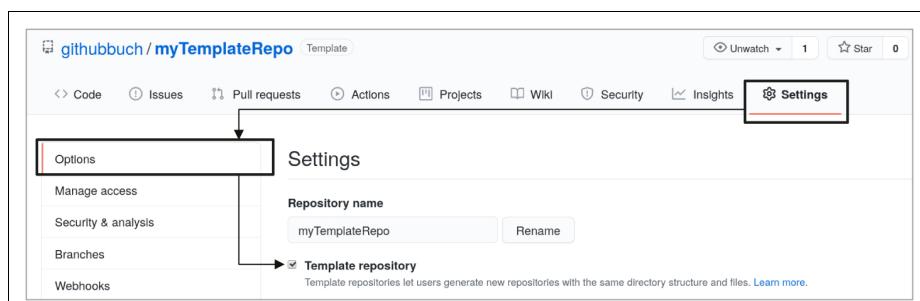


Abbildung 10-8: Um aus einem Repository ein Template zu machen, müssen wir über die Settings des entsprechenden Repositorys gehen.

Sobald du ein eigenes Template angelegt hast, kannst du im Dialog für das Erstellen eines neuen Repositorys jetzt auch dieses Template auswählen (siehe Abbildung 10-9).

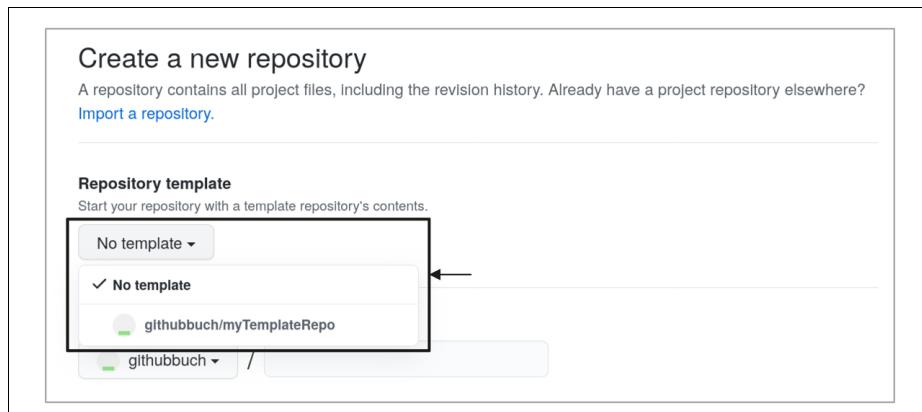


Abbildung 10-9: Wenn ein eigenes Template existiert, hast du beim Erstellen eines neuen Repos eine entsprechende Auswahlmöglichkeit.



#### Tipp: Repository aus Template-Repository via Webadresse erzeugen

Du kannst ein neues Repository aus einem Template-Repository einfach erstellen, indem du an die Webadresse des Template-Repositories einfach ein *generate* anhängst, z. B. <https://github.com/USER-NAME/TEMPLATENAME/generate>.

Ich persönlich habe erst wenige Templates in freier Wildbahn gesehen. Das mag aber auch daran liegen, dass das Feature erst seit Juni 2019 im Einsatz ist und sich noch nicht so durchgesetzt hat.

## Eigene Projektboards – mit Projects den Überblick behalten

Wer mit anderen Menschen zusammenarbeitet, benötigt häufig Werkzeuge, die die Zusammenarbeit vereinfachen oder mit denen man die anstehenden Aufgaben gut im Blick behalten kann. Hierfür bietet GitHub *Projects* an. Ich persönlich finde den Begriff etwas unglücklich gewählt, schließlich ist ja unsere Software (unser Benutzerhandbuch, unsere GitHub-Website etc.) das Projekt. Ich nenne das Werkzeug daher *Projektboard*, weil ich diesen Begriff treffender finde.

### Grundlegendes zu Projektboards

Was leistet so ein Projektboard? Ein Projektboard erlaubt es, offene Aufgaben im Blick zu behalten und zu verfolgen – im Team oder auch allein. Das macht man,

indem man die Aufgaben auf sogenannte Karten (*Cards*) schreibt und sie unter festgelegten Überschriften aufhängt. Auf den Karten sind alle relevanten Informationen zu finden, wie beispielsweise: Worum geht es? Wer macht es? Ist es ein Fehler oder eine Erweiterung? Mit welcher Detailtiefe man die Aufgaben auf den Karten notiert, ist projektabhängig.<sup>17</sup>

Die Überschriften können frei gewählt werden und sind ebenfalls von der Art des Projekts abhängig. Üblich sind drei bis fünf Überschriften wie *Noch offen*, *In Arbeit* und *Abgeschlossen* (siehe Abbildung 10-10) oder auch *Offen*, *In Entwicklung*, *Getestet* und *Deployt*. Man kann die Überschriften z. B. nach Modulen oder nach Prozessschritten wählen. Letzteres ist am häufigsten »in der freien Wildbahn« zu sehen und Kanban ein bekanntes Beispiel dafür.

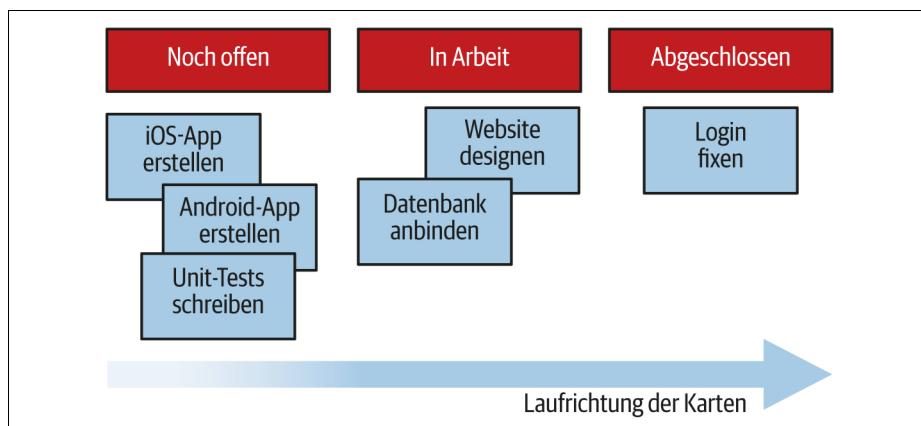


Abbildung 10-10: Um den Überblick zu behalten, werden Aufgaben auf Karten festgehalten und unter die jeweilige Überschrift gehängt.

Nur um eine Idee zu bekommen: Bei der Fertigung eines einzelnen Autos könnte man folgende Überschriften wählen:

- **Große Baugruppen:** »Motor«, »Innenausstattung« und »Fahrwerk«
- **Kleinere Baugruppen:** »Motor«, »Sitze«, »Radio«, »Lenkrad«, »Klimaanlage« und »Fahrwerk«
- **Prozessorientiert:** »Bauteil designen«, »Bauteil fertigen« und »Bauteil montieren«

Der Fantasie sind hier keine Grenzen gesetzt. Ich würde nur empfehlen, nicht zu viele Überschriften zu wählen, da ansonsten das Ganze recht unübersichtlich werden kann. Netterweise bietet uns GitHub ein paar Vorlagen an (dazu später mehr).

17 In der nicht digitalen Welt werden häufig Moderationskarten oder »Klebezettel« dafür eingesetzt, die an (Metaplan-)Wände gehetztet werden, um so einen Überblick über die Aufgaben zu bekommen.

Wie funktioniert das generell? Da jede Karte eine offene Aufgabe ist, sehe ich, wie viele Aufgaben unter welcher Überschrift noch offen sind. Erfülle ich eine Aufgabe, kann ich die dazugehörige Karte entfernen. Wann etwas wirklich fertig ist, führt häufig zu wilden Diskussionen. Ich empfehle dir daher, dich mit dem Thema *Definition of done* (deutsch etwa »Definition, wann etwas fertig ist«, siehe auch Erklärbox »Definition of done«) zu beschäftigen und für jede Aufgabe diese Definition explizit zu formulieren.

## Definition of done

Wer in Teams oder für Kunden arbeitet, sollte definieren, wann etwas wirklich fertig (*done*) ist. Warum? Menschen haben darüber häufig unterschiedliche Auffassungen, und eine entsprechende Definition hilft hier, Ärger zu vermeiden. In der Regel definiert man den »Fertig-Status« über Akzeptanzkriterien, was nichts weiter ist als eine Art Checkliste für festgelegte Bedingungen. Diese sollten natürlich möglichst eindeutig und einfach sein.

Um mal wieder ein Bild zu bemühen: Stell dir beispielsweise vor, dein pubertierendes Kind hätte seine vollgefüllte Müslischale in der Küche fallen lassen. Auf dem Boden liegen (viele) Teile des Mülsis, Milch ist auf den Boden und an Schränke gespritzt. Die Aufgabe an das Kind ist klar: »Mach den Dreck weg!« (Wir übersetzen das mal mit: »Mach es fertig.«). Wie es manchmal gerade bei Jugendlichen kommt:<sup>18</sup> Es wird nur das Nötigste gemacht. Die Müsliteile vom Boden werden aufgesammelt, und die Milch auf dem Boden wird mit einem Lappen weggeschwommen. Aus Sicht des Jugendlichen ist »der Dreck weg«. Als Elternteil sieht man das eventuell anders, die Milchspritze an den Schränken sind noch da, und der milchgetränkte Lappen wurde nicht ausgespült und liegt zerknüppelt irgendwo rum. »Der Dreck ist noch (teilweise) da«, die Aufgabe damit noch nicht fertig. Ein Streit ist vorprogrammiert.

In der Softwareentwicklung hat man dieses Problem längst erkannt und nutzt häufig das Muster »Vorausgesetzt – Wenn – Dann (– Und)«, um Akzeptanzkriterien für Software-Features festzulegen. Damit sollen möglichen Streitigkeiten der Nährboden entzogen werden. Angewandt auf unser Beispiel, könnte das wie folgt aussehen:

- **Vorausgesetzt:** die volle Müslischale ist dem Kind aus der Hand gefallen und der Inhalt befindet sich auf Boden, Wänden und Schränken.
- **Wenn:** das Kind »den Dreck weggemacht hat« ...
- **Dann:** sind alle verschütteten Inhalte der Müslischale nicht mehr auf Boden, Wänden und Schränken.
- **Und:** alle dafür genutzten Reinigungsgeräte sind wieder sauber und liegen wieder am Ursprungsort.

<sup>18</sup> Das soll kein Jugendliche-Bashing werden, ich war früher so ...

Auch dieses Beispiel hat Lücken. Es wird beispielsweise nicht der Fall abgedeckt, dass die Müslischüssel zerspringt und die Reste entsprechend entsorgt werden müssen. Falls Milchspritzer bis zur Lampe reichen würden, wäre deren Reinigung auch nicht mit abgedeckt etc. Ich denke aber, es ist klar, worum es geht.

Wer sein Projektboard prozessorientiert aufbaut, kann es darüber hinaus dazu nutzen, den Arbeitsfluss abzubilden. Was heißt das genau? Aufgaben durchwandern in der Regel das Projektboard vom ersten Prozessschritt (zumeist ganz links auf dem Board) bis zum letzten (zumeist ganz rechts auf dem Board). Habe ich bei einer Aufgabe einen der Schritte erledigt (z.B. »Prototyp entwickeln«), hänge ich die Karte um in den nächsten Prozessschritt »Prototyp testen«. Wenn sehr viele Karten in einem Prozessschritt hängen, weiß ich, dass ich irgendwo einen Engpass habe, z.B. weil meine Prototypenentwicklerin gerade krank ist. Dann kann ich entsprechende Gegenmaßnahmen einleiten.

Auf GitHub ist es möglich, für jedes Repository mehrere Projektboards einzurichten. Bei vielen größeren Projekten sieht man häufig, dass diese pro Release ein eigenes Projektboard aufbauen. Aber auch unterschiedliche Projektboards pro Arbeitsgruppe sind denkbar, beispielsweise eins für die Website-Gestaltung, eins für die App-Entwicklung und eins für Marketingmaßnahmen. Was und wie du schlussendlich Projektboards nutzt, dafür musst du selbst ein Gefühl entwickeln.

## Ein eigenes Projektboard erstellen

Wir wollen nun ein Projektboard erstellen. Gehe in einem eigenen Repository deiner Wahl auf das Register *Projects* und wähle dann *Create a project* (siehe Abbildung 10-11).

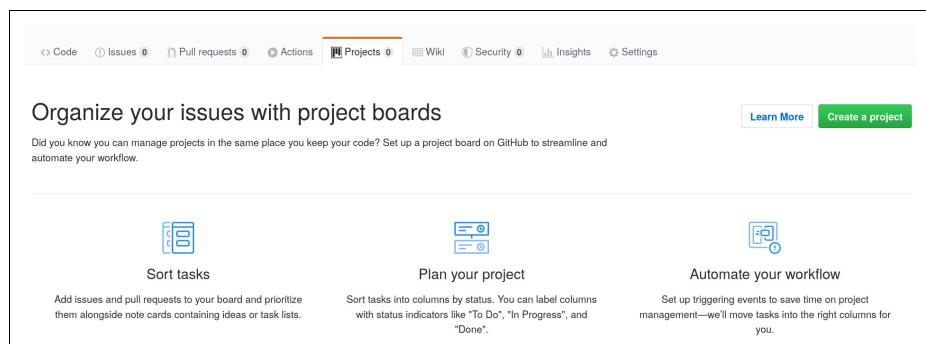


Abbildung 10-11: Unter »Projects« kann man mehrere Projektboards anlegen.

Neben einem möglichst aussagekräftigen Namen kannst du auch eine Vorlage auswählen (siehe Abbildung 10-12).

### Create a new project

Coordinate, track, and update your work in one place, so projects stay transparent and on schedule.

**Project board name**

**Description (optional)**

**Project template**

Save yourself time with a pre-configured project board template.

Template: **Basic kanban** ▾

**Create project**

Abbildung 10-12: GitHub bietet Vorlagen für Projektboards an.

Aktuell gibt es die folgenden Vorlagen:

- **Basic kanban** – dreispaltiges Projektboard mit den Spalten *To do*, *In progress* und *Done*.
- **Automated kanban** – wie *Basic kanban*, jedoch mit Automatisierungsmöglichkeiten für jede Spalte, um die Karten zwischen den Spalten hin und her zu bewegen.
- **Automated kanban with review** – wie *Automated kanban*, es gibt nur weitere Optionen, um auf den Review-Status eines Pull-Requests zu reagieren.
- **Bug triage** (deutsch »Fehlersichtung«) – vier-spaltiges Projektboard mit den Spalten *To do*, *High priority*, *Low priority* und *Closed* und Automatisierungsmöglichkeiten auf der ersten und der letzten Spalte.

Wähle *Basic kanban* und erstelle das Projektboard durch Klicken von *Create Project*. Du wirst dann auf dein neues Projektboard geleitet, das ähnlich aussehen sollte wie das in Abbildung 10-13.

Wir sehen zunächst die drei vorgefertigten Spalten *To do*, *In progress* und *Done* ❶. GitHub hat unser Projektboard gleich mit den ersten exemplarischen Karten bevölkert. Diese kannst du ganz einfach per Drag-and-drop von Spalte zu Spalte bewegen. Über das Pluszeichen ❷ kannst du neue Karten hinzufügen (GitHub nennt diese *note* für »Notiz«). Unter ❸ hast du die Möglichkeit, die Spalten zu editieren oder eine Automatisierung einzustellen. Das wollen wir uns gleich noch näher anschauen.

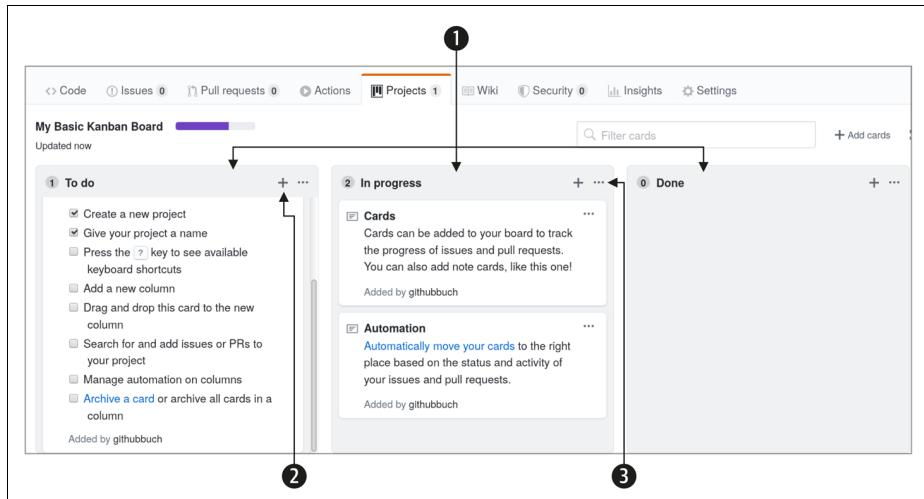


Abbildung 10-13: Die weißen Karten lassen sich per Drag-and-drop den einzelnen Spalten zuordnen.

Du kannst in solchen Projektboards neben den bereits erwähnten »Notizen« auch Issues und Pull-Requests als Karten anlegen und hast so den Überblick über alle wichtigen Aktivitäten innerhalb deines Projekts.

## Projektboard automatisieren

GitHub bietet einige Automatisierungsmöglichkeiten für Projektboards an, von denen wir ein paar ausprobieren werden. Eine kleine Warnung vorweg: So toll das zunächst klingt, es gibt die eine oder andere Einstellung, die vielleicht etwas enttäuschend ist, da sie ein anderes Verhalten als erwartet an den Tag legt. Dazu gleich mehr.

Wähle in der Spalte *To do* die drei Punkte aus (siehe Punkt ③ in Abbildung 10-13) und klicke auf *Manage automation* (deutsch »Verwalte Automatisierung«). Du solltest eine Maske ähnlich wie die in Abbildung 10-14 zu sehen bekommen. Wähle als *Preset* (deutsch »Voreinstellung«) *To do* aus und aktiviere bei *Move issues here when...* die Checkbox *Newly added*. Danach klicke auf *Update automation* (»Aktualisiere Automatisierung«).

Danach solltest du im unteren Bereich der Spalte einen Hinweis dazu bekommen, dass diese Spalte automatisiert wird (siehe Abbildung 10-15). Über das Anklicken von *Manage* kommst du zurück zu den Einstellungen für die Automatisierung.

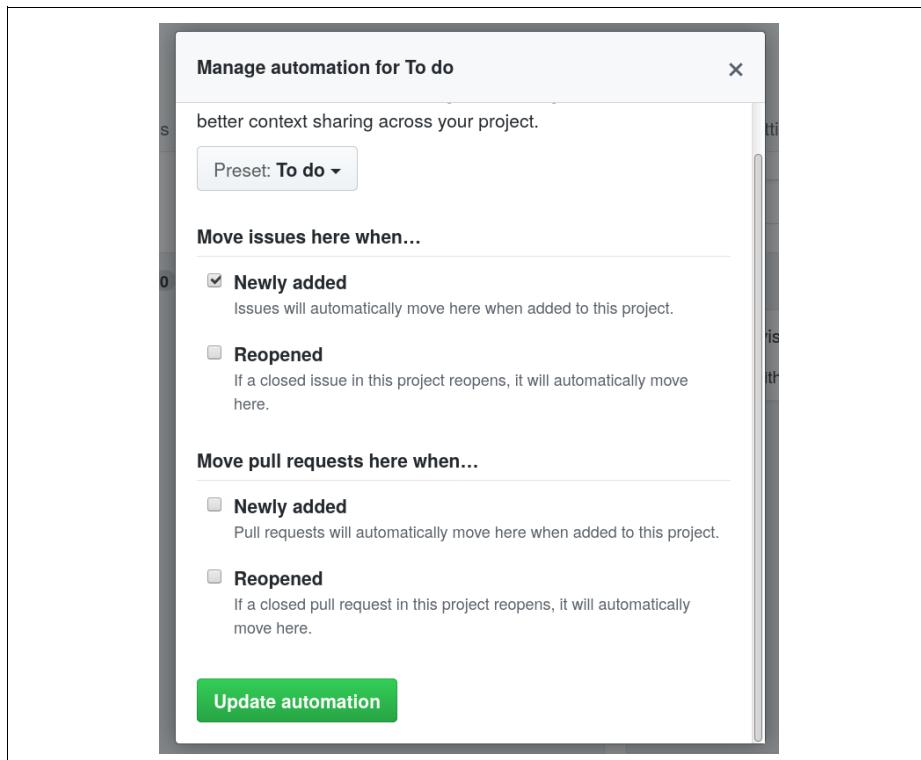


Abbildung 10-14: GitHub bietet verschiedene Vorlagen für die Automatisierung an.

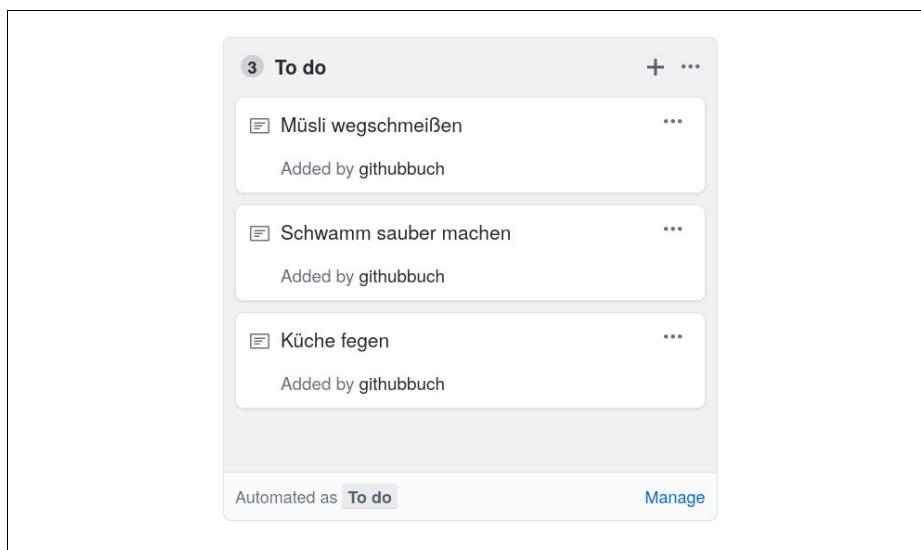


Abbildung 10-15: Ist eine Automatisierung eingestellt, sieht man das in der jeweiligen Spalte am unteren Rand.

Wenn du jetzt ähnlich wie ich »Ab sofort wird also jeder neue Issue automatisch in das Projektboard gehängt« denkst, liegst du leider, genau wie ich, falsch. Was automatisiert wird, ist Folgendes: Sobald du einen Issue einem Projekt manuell zuordnest, wird der Issue automatisch im entsprechenden Projekt angezeigt.

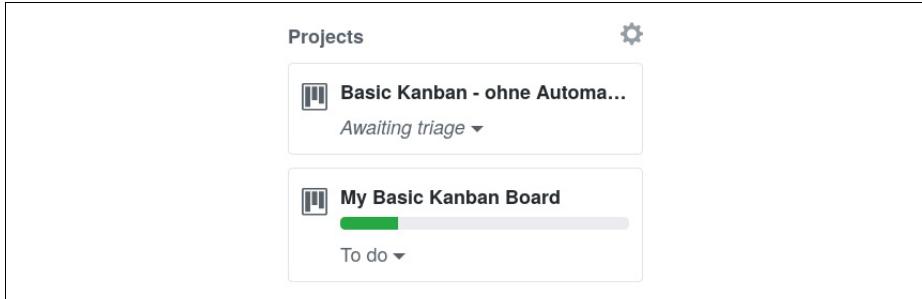


Abbildung 10-16: Die Automatisierung eines Projektboards greift erst dann, wenn man einen Issue einem Projektboard manuell zuordnet.

Durch Abbildung 10-16 wird das etwas klarer. Ich habe einen Issue erstellt und ihn zwei Projekten manuell zugeordnet. Der Issue ist oben einem Projekt ohne Automatisierung zugeordnet und wartet noch auf eine Sichtung (*Awaiting triage*). Ich müsste die Zuordnung zu dem Projektboard noch manuell anstoßen. Eine noch ausstehende Sichtung zeigt auf dem Projektboard übrigens ein blauer Punkt an (siehe Abbildung 10-17). Der Issue ist unten bereits einem Projektboard mit Automatisierung zugeordnet worden.



#### Tipp: App für Projektboard-Automatisierung

Im Marketplace gibt es die App *project-bot*<sup>19</sup>, mit der du dein Projektboard weiter automatisieren kannst.

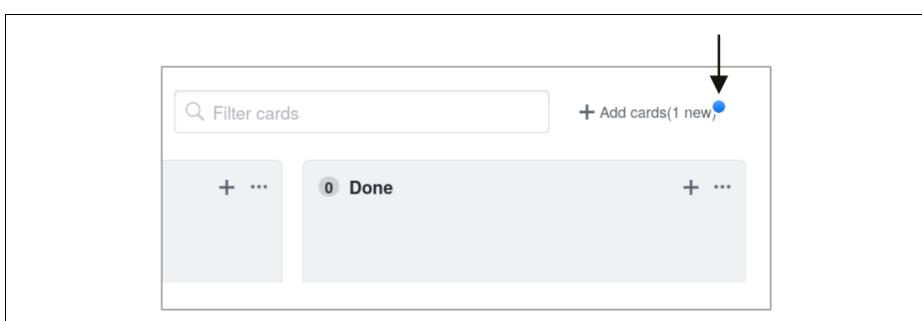


Abbildung 10-17: Der kleine blaue Punkt bei »Add cards« informiert über neue Karten, die noch gesichtet werden müssen.

19 <https://github.com/apps/project-bot>

Die Automatismen beim Schließen eines Issues funktionieren übrigens wie erwartet. Diese sind im *Preset: Done* zu finden und schieben tatsächlich automatisiert einen Issue im Projektboard in die gewünschte Spalte, sobald der Issue geschlossen wird.

So viel zu weiterführenden GitHub-Features, die hoffentlich für das eine oder andere Projekt von dir nützlich sein werden. Im letzten Kapitel möchte ich noch weitere Möglichkeiten anreißen, die eine genauere Betrachtung wert sind. Ich werde zudem einige Werkzeuge vorstellen, die das Arbeiten mit GitHub erleichtern.

## KAPITEL 11

# Nützliches und Kuriöses rund um GitHub

Dieses letzte Kapitel ist mein GitHub-Sammelsurium, in dem ich alles zusammengetragen habe, was ich rund um GitHub als hilfreich, nützlich, interessant oder witzig empfunden habe.



### Am Ende des Kapitels kennst du ...

- einige Werkzeuge, um GitHub auf der Kommandozeile zu bedienen.
- Quellen, um dich mit GitHub weiter auseinanderzusetzen.
- einige Editoren und Smartphone-Apps, die beim Arbeiten mit GitHub hilfreich sein können.
- nützliche Werkzeuge und Witziges in und um GitHub herum.
- weitere Möglichkeiten, GitHub außerhalb von Softwareprojekten zu nutzen.

## GitHub auf der Kommandozeile

Bisher haben wir uns GitHub »nur« auf der Weboberfläche angesehen. Wenn du aber vermehrt mit Git auf der Konsole arbeitest, kann es manchmal ganz schön nervig sein, ständig zwischen Konsole und Webbrower hin- und herwechseln zu müssen, nur weil du noch eben ein neues Repository anlegen möchtest. Zum Glück gibt es mehrere Wege, GitHub auch über die Konsole zu steuern, und drei davon möchte ich dir kurz vorstellen: GitHub CLI, hub und die GitHub-API.

### GitHub CLI

Anfang 2020 hat GitHub in einem Betastadium ein neues Werkzeug namens *GitHub CLI*<sup>1</sup> (*Command Line Interface*, deutsch »Kommandozeilenschnittstelle«) veröffentlicht, das einige der Konzepte von der GitHub-Weboberfläche auf die Konsole

---

<sup>1</sup> <https://cli.github.com/>

bringt, z.B. Issues anschauen, Pull-Requests anlegen oder ein Repository forken. Wenn du dieses Buch in den Händen hältst, ist die Betaphase vermutlich bereits abgeschlossen, und der Funktionsumfang hat sich eventuell deutlich verändert.



### Tipp: Tastaturkürzel

Wenn du generell lieber über die Tastatur als über die Maus navigierst, kannst du durch Eintippen von ? auf einer GitHub-Seite eine Übersicht der auf dieser Seite funktionierenden Tastaturkürzel bekommen (auch *Shortcuts* genannt).

Ein neues Repository lässt sich beispielsweise wie folgt anlegen:

```
$ gh repo create
```

Die Website bietet zudem eine Übersicht aller Befehle an,<sup>2</sup> siehe auch Abbildung 11-1.

The screenshot shows the GitHub CLI documentation page. On the left, there's a sidebar with navigation links: Getting started, Overview, Installation, Examples in use (with sub-links: Checking out, Creating, Using lists, Checking status, Viewing an item), Commands completion, issue (with sub-links: create, list, status, view), pr (with sub-links: create, checkout, status, view), and global flags (with sub-links: --help, -R, --repo OWNER/REPO, --version). The main content area has a title "GitHub CLI". Below it is a paragraph about the gh command. There are sections for "Usage" and "Commands". Under "Commands", there's a link to "View detail on any command, use gh [command] [subcommand] --help". A bulleted list follows: • gh issue [status, list, view, create] • gh pr [status, list, view, checkout, create] • gh repo [clone, create, fork, view] • gh help. The "Global flags" section contains three entries: --help (Show help for command), -R, --repo OWNER/REPO (Select another repository using the OWNER/REPO format), and --version (Show gh version).

Abbildung 11-1: Die Website GitHub CLI zeigt eine kompakte Übersicht aller möglichen Befehle.



### Wichtig zu wissen: GitHub CLI ist ein offizielles Werkzeug

GitHub CLI ist ein offizielles Werkzeug von GitHub, das bedeutet, es wird aktiv weiterentwickelt, und du hast die Möglichkeit, Support dafür zu bekommen (eventuell gegen Einwurf kleiner Münzen). Wenn du nicht bereits eines der anderen Werkzeuge einsetzt (siehe nachfolgende Abschnitte), ist es auf alle Fälle einen Blick wert.

<sup>2</sup> <https://cli.github.com/manual/>

## Hub

Hub<sup>3</sup> ist eine Open-Source-Lösung, die GitHub ebenfalls auf die Konsole bringt und das Zusammenspiel mit Git vereinfacht. Beispielsweise könntest du dir alle Issues anzeigen lassen, die dir zugewiesen sind und das Label `urgent` tragen:

```
$ hub issue --assignee YOUR_USER --labels urgent
```

Weitere Anwendungsbeispiele sind auf der Website zu finden.<sup>4</sup>

Hub ist zwar keine offizielle Erweiterung von GitHub, aber rein zufällig ist der Hauptentwickler des hub-Projekts auch bei GitHub angestellt. Das bedeutet, dass das Werkzeug nur so lange aktiv weiterentwickelt wird, solange es genügend Unterstützende gibt – wie bei so vielen Open-Source-Projekten.

## GitHub-API (für Fortgeschrittene)

GitHub bietet eine mächtige Schnittstelle (API),<sup>5</sup> die ebenfalls über die Konsole ansprechbar ist und mit deren Hilfe du auch eigene Applikationen im GitHub-Uni-versum bauen kannst.

Ein relativ simples Beispiel ist das Anlegen eines GitHub-Repositories.<sup>6</sup> Die API kannst du über die Software `cURL` ansprechen, und der Befehl zum Anlegen eines Repositories könnte wie folgt aussehen:

```
$ curl -u "USERNAME" https://api.github.com/user/repos -d '{"name": "REPONAME"}'
```

Wenn du in der Dokumentation zur API stöberst, wirst du feststellen, dass du mit dem Zugriff auf die API deutlich mehr Möglichkeiten hast als mit den anderen bei-den genannten Werkzeugen. Beispielsweise kannst du damit auch Repositories aus Templates erzeugen oder alle Topics eines Repositorys ändern.

## cURL

cURL bietet den Konsolenbefehl `curl`, mit dem du Daten von und an einen Server senden kannst. Unterstützte Protokolle sind dabei unter anderem HTTP (*Hyper-text Transfer Protocol*, ein Protokoll, um Webseiten in einen Browser zu laden), FTP (*File Transfer Protocol*, ein Protokoll zum Austausch von Daten), IMAP (*Internet Message Access Protocol*, ein Protokoll zur Bereitstellung von E-Mails) und viele weitere Protokolle, die alle auf der cURL-Website<sup>7</sup> aufgeführt sind.

3 <https://github.com/github/hub>

4 <https://hub.github.com/>

5 <https://developer.github.com/v3/>

6 Es gibt ein wunderbares Video, das den Vorgang in fünf Minuten beschreibt:  
[https://www.youtube.com/watch?v=6xmFp4\\_U9-A](https://www.youtube.com/watch?v=6xmFp4_U9-A).

7 <https://curl.haxx.se/>

Gesteuert wird das Ganze über entsprechende Parameter, die du beim Programmaufruf von cURL angeben kannst. Man kann sich beispielsweise eine Website in eine bestimmte Datei herunterladen (das -o steht für *output*):

```
$ curl -o DATEINAME http://WEBSEITE.DE
```

Genauere Informationen zu den jeweiligen Parametern findest du entweder auf der Homepage<sup>8</sup> oder über die `man`-Page auf der Konsole:

```
$ man curl
```

Der Name cURL steht übrigens für *Client für URLs* (die URL ist der *Uniform Resource Locator*, ein Identifikator für beispielsweise eine Webseite), und die Software wird auf GitHub weiterentwickelt.<sup>9</sup>

Ein Vorteil ist, dass das dafür eingesetzte Werkzeug cURL bei vielen Linux-Distributionen bereits standardmäßig installiert ist. Für Fortgeschrittene und/oder Firmen bietet die API auf jeden Fall interessante Möglichkeiten.

## Sich mit GitHub weiter auseinandersetzen

Ich hoffe, dass ich dich bis hierhin so weit »angezündet« habe, dass du Lust auf mehr GitHub hast. In diesem Abschnitt zeige ich dir, wo du weitere Lernmaterialien und Ressourcen zum Nachschlagen findest.

### GitHub Learning Lab

GitHub selbst bietet das sogenannte *GitHub Learning Lab* an, eine App, die man über den Marktplatz<sup>10</sup> installieren kann. Das Ganze ist in Module aufgeteilt, und du kannst dir nach und nach die Module, die dich interessieren, nachinstallieren<sup>11</sup> (siehe auch Abbildung 11-2 für einen kleinen Überblick über die Module).

Für ein erstes Hineinschnuppern in ein Thema ist das Lab sehr gut geeignet. Es führt einen im eigenen Repository durch die erforderlichen Schritte, wie beispielsweise das Anlegen von Issues, Pull-Requests und Branches. Auch GitHub Pages und GitHub Actions werden behandelt. Es gibt sogar Module, um Programmiersprachen oder Methoden wie *Design Thinking* zu lernen, und es kommen immer mehr hinzu.

8 <https://curl.haxx.se/docs/manpage.html>

9 <https://github.com/curl/curl>

10 <https://github.com/marketplace/github-learning-lab>

11 <https://lab.github.com/>

Du kannst, wenn du möchtest, das *GitHub Learning Lab* auch selbst unterstützen und weitere Komponenten hinzufügen.<sup>12</sup>

The screenshot displays the GitHub Learning Lab interface, which is a collection of modular learning resources. At the top, there's a section titled "Learning Paths" with a sub-section "First Day on GitHub". This path includes a brief introduction and four items: "What is GitHub?", "Introduction to GitHub", "Git Handbook", and "And 2 more...". Below this is another path titled "First Week on GitHub" by "The GitHub Training Team", which covers GitHub Pages, pull requests, and workflows. To the right is the "InnerSource: theory to practice" path, also by "The GitHub Training Team", focusing on InnerSource fundamentals and case studies. A sidebar on the right lists other learning paths like "Beccente", "Learn", and "Contributing".

**Learning Paths**

Learn something new with these curated lists of our favorite courses, videos, tutorials, and more.

**First Day on GitHub**  
Learning Path by [The GitHub Training Team](#)

Welcome to GitHub! We're so glad you're here. We know it can look overwhelming at first, so we've put together a few of our favorite courses for people logging in for the first time

- What is GitHub?
- Introduction to GitHub
- Git Handbook
- And 2 more...

**First Week on GitHub**  
Learning Path by [The GitHub Training Team](#)

After you've mastered the basics, learn some of the fun things you can do on GitHub. From GitHub Pages to building projects with your friends, this path will give you plenty of new ideas.

- Discover GitHub Pages
- Reviewing pull requests
- Securing your workflows

**InnerSource: theory to practice**  
Learning Path by [The GitHub Training Team](#)

Learn about the concept of InnerSource and put it to use in this carefully crafted learning path.

- An Introduction to InnerSource
- InnerSource fundamentals
- Case studies
- And 3 more...

**Beccente**  
Learning...  
you're here. We know it can look...  
overwhelming at first, so we've put...  
together a few of our favorite courses...  
for people logging in for the first time

**Learn GitHub with GitHub**

**GitHub Pages**  
The GitHub Training Team

Learn how to create a site or blog from your GitHub repositories with GitHub Pages.

- GitHub
- GitHub Pages

**Community starter kit**  
The GitHub Training Team

There are millions of projects on GitHub, all competing for attention from the millions of open source contributors available to help. Learn how to help your project stand out.

- GitHub
- Open Source

**Uploading your project to GitHub**  
The GitHub Training Team

You're an upload away from using a full suite of development tools and premier third-party apps on GitHub. This course helps you seamlessly upload your code to GitHub and introduces you to exciting next steps to elevate your project.

- Git
- GitHub

**Migr to Gi**  
The...  
You're...  
full su...  
premi...  
This co...  
code to...  
more a...  
• Git

**Languages and Tools**

**Introduction to HTML**  
The GitHub Training Team

If you are looking for a quick and fun introduction to the exciting world of programming, this course is for you. Learn fundamental HTML skills and build your first webpage in less than an hour.

- GitHub Pages

**Introduction to Node with Express**  
everydeveloper

Node.js gives you the ability to run JavaScript files on the server-side. Express is a library for Node.js, that allows you to make requests to different "endpoints" and get a response back.

- Node
- Express
- JavaScript
- JSON
- API

**Introduction to Python**  
everydeveloper

Go from Hello World to writing a short random quote generator using Python.

- Python

**Inter**  
every...  
Learn...  
with Py...  
• Pytho...

Abbildung 11-2: Das *GitHub Learning Lab* enthält eine Reihe von Modulen für einen guten Einstieg direkt »hands-on«.

12 <https://github.com/github/learning-lab-components>

## Weitere Ressourcen zum Recherchieren

GitHub selbst bietet noch weitere Ressourcen an, um sich tiefgehender mit einzelnen Themen zu beschäftigen, um Antworten auf drängende Fragen zu bekommen oder um Kontakt zu Gleichgesinnten aufzubauen:

- Die Hilfeseite von GitHub, auf der du bei Fragen als Erstes suchen solltest, von mir bereits mehrfach zitiert (<https://help.github.com/>).
- Kurze Anleitungen zu Themen wie GitHub Pages und Git, aber auch Anleitungen, wie deine Veröffentlichung auf GitHub zitierfähig werden kann (<https://guides.github.com/>).
- Falls du selber programmierst und tiefer einsteigen möchtest, bietet GitHub eine Extrahilfe und Anleitungen für Entwicklerinnen und Entwickler, beispielsweise wie du eine GitHub-App für den Marketplace entwickeln kannst (<https://developer.github.com/>).
- Eine gute Anlaufstelle, um Kontakt zu Gleichgesinnten aufzunehmen oder Fragen aller Art loszuwerden (und manchmal auch für deren Antworten), ist das Communityforum von GitHub. Hier schreiben durchaus auch GitHub-Mitarbeitende (<https://github.community/>).
- Bist du vielleicht Lehrer? Oder Schülerin/Studentin? Hier gibt es beispielsweise eine »GitHub Teacher Toolbox« mit vielen kostenlosen Entwicklungswerkzeugen für Menschen im Bildungswesen (<https://education.github.com/>).
- Falls dich Zahlen, Daten, Fakten rund um GitHub interessieren – etwa wie viele Entwickler\*innen GitHub nutzen oder aus welchen Ländern diese kommen –, wirst du im Bericht »The State of the Octoverse« (<https://octoverse.github.com/>) fündig.
- Möchtest du wissen, ob es neue Features auf GitHub gibt, lohnt sich ein Blick ins Changelog (<https://github.blog/changelog/>). Falls dich interessiert, woran die GitHub-Entwickler\*innen derzeit arbeiten, findest du das auf der Roadmap (<https://github.com/github/roadmap/projects/1>).

Wer an diesen Stellen nicht fündig wird, kann sich auch außerhalb des GitHub-Universums umschauen.

- Von mir bereits in Kapitel 2 empfohlen, aber in meinen Augen so essenziell, dass ich es gern wiederhole: Stackoverflow ist die Anlaufstelle schlechthin für Programmier- und Konfigurationsfragen – auch unabhängig von GitHub (<https://stackoverflow.com/questions>).
- Für Fragen, Plaudereien und Diskussionen gibt es bei Reddit eine Anlaufstelle zu GitHub (<https://www.reddit.com/r/github/>).

## Editoren und Handy-Apps

Bist du noch auf der Suche nach einem lokalen Texteditor oder einer App für das Handy? Dann stelle ich dir einige Texteditoren vor, die eine einfache Nutzung von

Git und GitHub bereitstellen. Falls dir mehr nach einer grafischen Oberfläche ist, empfehle ich dir einen Blick auf die Git-Website<sup>13</sup>. Für die mobile Nutzung stelle ich zwei Apps für GitHub kurz vor.

## Klein und schlank – Atom

Ich persönlich kann den Editor Atom<sup>14</sup> wärmstens empfehlen, den es sowohl für Linux als auch für Windows und macOS gibt. Es gibt zahlreiche Erweiterungen (Packages genannt), mit denen man den Editor weiter aufmotzen kann, unter anderem durch Funktionalitäten für Git und GitHub, aber auch To-do-Listen<sup>15</sup> oder gar Spiele<sup>16</sup> sind möglich (siehe Abbildung 11-3, links sind die Dateien aufgelistet, in der Mitte siehst du den Editor und rechts sind die Dateien aufgelistet, in der Mitte siehst du den Editor und rechts meine Packages *ToDo Show* und *Git*). Eine vollständige Auflistung aller Erweiterungen findest du unter <https://atom.io/packages/>.

Atom wird auf GitHub weiterentwickelt,<sup>17</sup> sodass du ganz einfach Fehler melden oder mitmachen kannst. Das Repository hat eine sehr ausführliche *CONTRIBUTING.md* (wir erinnern uns an Abschnitt »Contributing« auf Seite 109 in Kapitel 6), in der man gerne mal »spicken« kann.

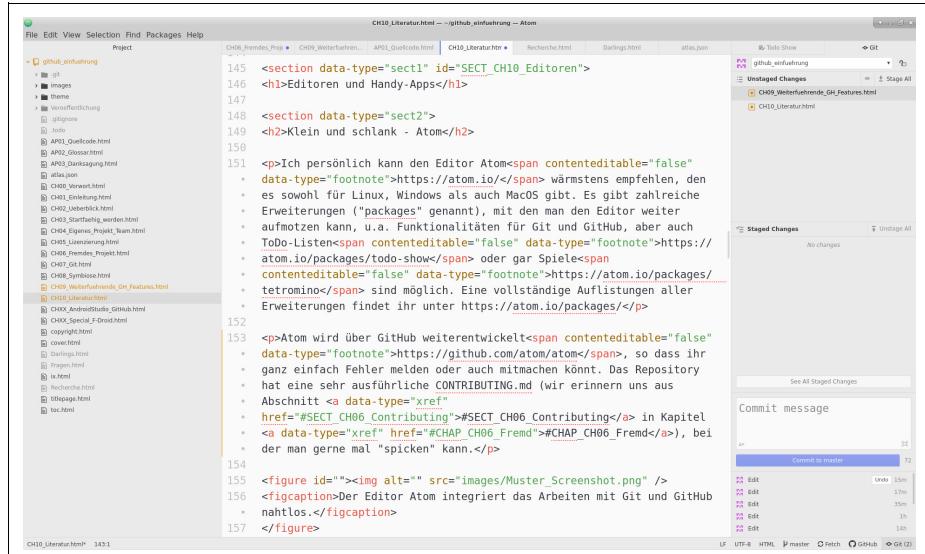


Abbildung 11-3: Der Editor Atom integriert das Arbeiten mit Git und GitHub nahtlos.

13 <https://git-scm.com/downloads/guis/>

14 <https://atom.io/>

15 <https://atom.io/packages/todo-show>

16 <https://atom.io/packages/tetromino>

17 <https://github.com/atom/atom>

## Visual Studio Code

Ein weiterer mächtiger Editor ist *Visual Studio Code*,<sup>18</sup> auch VS Code oder VSC abgekürzt, den es für Linux, Windows und macOS gibt. Neben zahlreichen Erweiterungen (hier *Extensions* genannt)<sup>19</sup> bietet der Editor noch viele Dinge mehr, wie beispielsweise die Autovervollständigung von Code, die einem das Leben leichter machen.

Visual Studio Code ist »Microsoft branded« und steht unter einer entsprechenden Lizenz, was dem Editor teils Kritik einbringt, da sich Microsoft unter anderem das Recht sichert, Nutzungsdaten zu übermitteln (sogenannte Telemetriefunktionen, dies kann aber abgeschaltet werden). Traust du nur dem, was du selbst gebaut hast, gibt es bei VS Code die Möglichkeit, eben dies zu tun. Der Quellcode ist auf GitHub veröffentlicht, und es gibt sogar eine entsprechende Bauanleitung.<sup>20</sup>

VS Code könnte aber trotz der erwähnten Kritik in Zukunft durchaus interessant werden. Ein zukünftiges Feature von GitHub<sup>21</sup> nennt sich *Codespaces* und ermöglicht den Aufbau einer kompletten Entwicklungsumgebung im virtuellen Raum von GitHub. VS Code ist dabei der Standardeditor. Ob das Feature auch für Open-Source-Repositories kostenlos zur Verfügung stehen wird, stand zum Zeitpunkt der Drucklegung dieses Buchs noch nicht fest.

## Für Website-Gestalter – Brackets

Ein weiterer interessanter Editor ist aus meiner Sicht Brackets<sup>22</sup>. Brackets gibt es für macOS, Windows und Linux (Debian/Ubuntu) und ist primär ein Codeeditor für HTML, CSS und JavaScript. Wie auch bei Atom gibt es vielfältige Erweiterungsmöglichkeiten für Git und GitHub sowie andere nützliche Helfer wie das Hinzufügen von Datei-Tabs<sup>23</sup> oder To-do-Listen<sup>24</sup>.

Eine Sache, die ich dort sehr schätzen gelernt habe, ist die Unterstützung beim Einfügen von Bildern. Bilder, die lange und/oder kryptische Namen haben, lassen sich einfach hinzufügen, da Brackets während des Tippens des Bildnamens eine Auswahlliste aller passenden Bilder anbietet, um diese dann leicht einzufügen zu können.

---

18 <https://code.visualstudio.com/>

19 <https://marketplace.visualstudio.com/VSCodium>

20 <https://github.com/microsoft/vscode>

21 Wenn du dieses Buch in den Händen hältst, könnte das Feature auch schon veröffentlicht sein.

22 <https://brackets.io/>

23 <https://github.com/dn'bard/brackets-documents-toolbar>

24 z.B. <https://github.com/mikaeljorhult/brackets-todo>

## GitHub Desktop

GitHub selbst bietet eine eigene Entwicklungsumgebung namens *GitHub Desktop*<sup>25</sup> mit – natürlich – nahtloser Integration in GitHub an. Leider gibt es GitHub Desktop nur für Windows und macOS, sodass ich mir die Software nicht näher anschauen konnte.

## GitHub auf dem Handy – GitHub Mobile

Seit März 2020 gibt es offiziell von GitHub eine App, um mobil auf die eigenen Repositories, Issues und Pull-Requests zugreifen zu können (siehe auch Abbildung 11-4).<sup>26</sup> Die App gibt es für Android (Google Play Store<sup>27</sup>) und für iPhones (Apple App Store<sup>28</sup>).

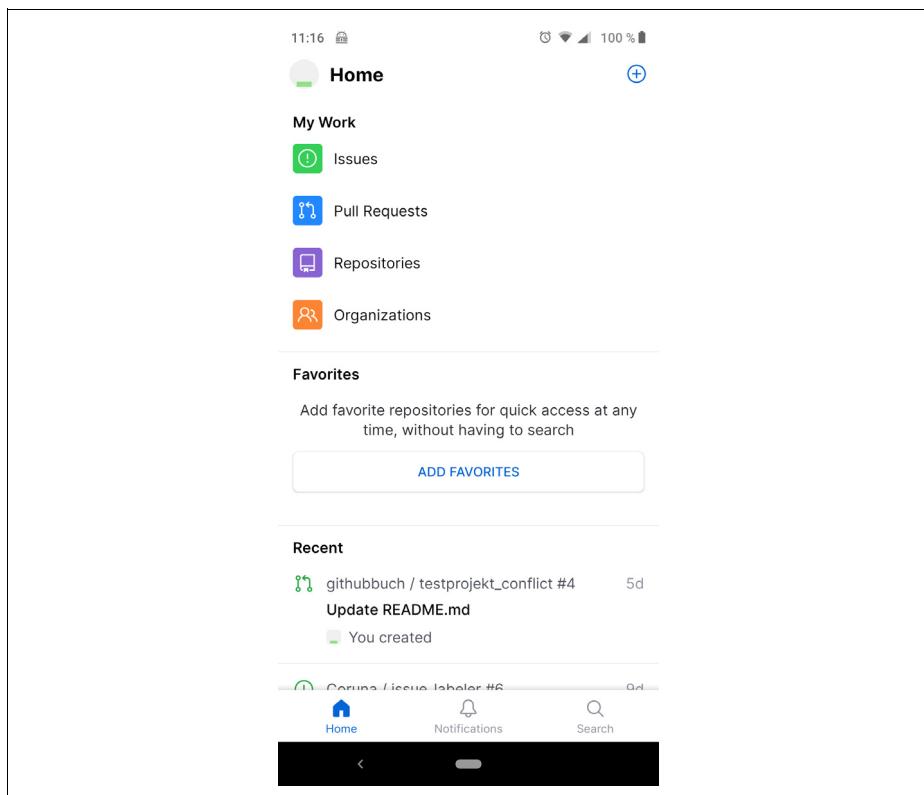


Abbildung 11-4: GitHub Mobile ist eine App für GitHub von GitHub.

25 <https://desktop.github.com/>

26 <https://github.com/mobile>

27 <https://play.google.com/store/apps/details?id=com.github.android>

28 <https://apps.apple.com/app/github/id1477376905?ls=1>

Du kannst damit unter anderem Issues und Pull-Requests lesen und kommentieren, Pull-Requests mergen und Issues mit Labels versehen.

## GitHub auf dem Handy – Octodroid

Es gibt unzählige weitere GitHub-Apps für Android- und Apple-Handys. Da ich nicht alle Apps einzeln vorstellen möchte, habe ich mir nur eine Open-Source-App herausgepickt, die ich selbst nutze. Sie heißt *Octodroid* und kann über den Play Store von Google<sup>29</sup> oder über den App-Store F-Droid<sup>30</sup> bezogen werden (siehe Abbildung 11-5).

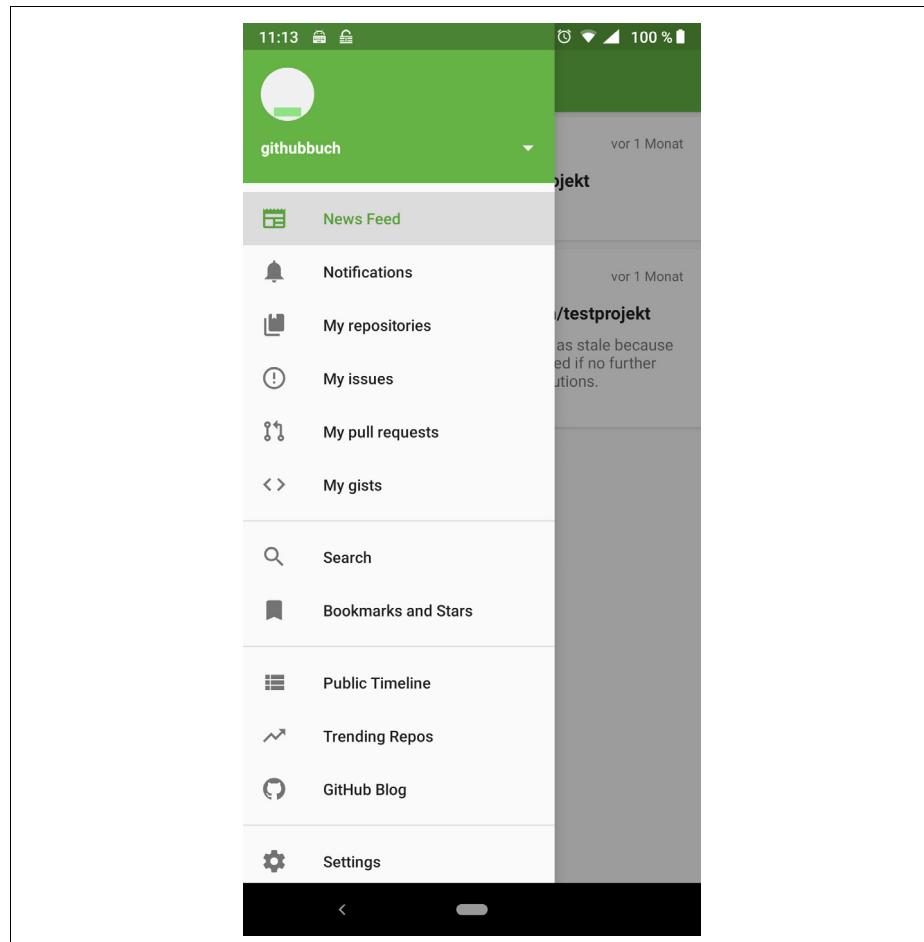


Abbildung 11-5: Octodroid ist eine Open-Source-Android-App für GitHub.

29 <https://play.google.com/store/apps/details?id=com.gh4a>

30 <https://f-droid.org/>

Octodroid hat ähnliche Funktionalitäten wie GitHub Mobile. Du kannst damit mobil einen Überblick über deine Repositories, Pull-Requests und Issues behalten und beispielsweise auch Kommentare verfassen.

## Nützliches und kleine Spielereien

Falls du dein Projekt in mehreren Sprachen anbieten und die Übersetzungen verwalten möchtest, lohnt sich ein Blick auf ein Übersetzungsmanagementtool. Ich stelle dir hier zwei solcher Tools kurz vor. Und die Welt wäre ein deutlich traurigerer Ort, wenn es nicht die eine oder andere Spielerei nach dem Motto »nicht wichtig, aber witzig« geben würde. Davon zeige ich dir ebenfalls zwei.

### Übersetzungsmanagementtools – crowdin und Weblate

Wer eine Homepage oder eine Software erstellt, möchte in der Regel, dass möglichst viele Menschen diese auch nutzen können. Das Übersetzen in unterschiedliche Sprachen ist eine gute Möglichkeit, die Reichweite zu erhöhen. Da man in der Regel selbst nicht mehr als zwei oder drei Sprachen spricht (Sprachgenies mal ausgenommen), besteht die Möglichkeit, dass dich freiwillige Übersetzer dabei unterstützen. Ein viel genutzter Weg besteht darin, unterschiedliche Sprachen über Dateinamen abzubilden, beispielsweise *datei\_DE.xml* für deutsche Texte und *datei\_ES.xml* für spanische.

Wer mehr Unterstützung und Visualisierung für Übersetzungen braucht, kann dafür entsprechende Übersetzungsmanagementtools verwenden, wie beispielsweise *Crowdin*<sup>31</sup> (siehe Abbildung 11-6) oder *Weblate*<sup>32</sup> (siehe Abbildung 11-7). Hier kann man – je nach Toolwahl – eine Übersicht darüber bekommen, welche Sprache das eigene Projekt bereits spricht oder auch zu wie viel Prozent die Menüs und Texte bereits in eine Sprache übersetzt wurden. Beide bieten zudem unterschiedliche unterstützende Werkzeuge für Übersetzungen an.

Um Crowdin oder Weblate benutzen zu können, benötigst du einen entsprechenden Account, du kannst dich aber auch mit deinem GitHub-Account anmelden. Sofern du ein Open-Source-Projekt betreibst, gibt es die Möglichkeit, auf Anfrage beide Dienste kostenlos zu nutzen. Nach meiner Recherche sind beide Dienste erst dann interessant, wenn du ein Projekt schon etwas länger »am Start« hast. Crowdin erwartet z.B., dass man seit mindestens drei Monaten an dem eigenen Projekt arbeitet und auch eine aktive Gemeinschaft an Unterstützern vorweisen kann.

---

31 <https://crowdin.com/>

32 <https://weblate.org/de/>

The screenshot shows the Crowdin interface for the Minecraft project. At the top, there are navigation links for 'Start Own Project' and 'Explore'. A search bar says 'Search Projects'. On the right, there are icons for notifications and user profiles. Below the header, the project name 'Mojang' is shown with a small icon, and a 'Join' button. The main content area is titled 'Minecraft' with tabs for 'Home', 'Activity', 'Reports', and 'Discussions'. A search bar says 'Search languages'. Under 'Translations:', there is a grid of language flags and completion percentages:

Language	Completion (%)
German	100% • 99%
Afrikaans	100% • 30%
Albanian	96% • 22%
Alligolian German	20% • 12%
Andalusian	97% • 43%
English	99% • 72%
Arabic	99% • 26%
Armenian	62% • 0%
Asturian	88% • 60%
Azerbaijani	76% • 40%

To the right, there is a large image of a grassy block from Minecraft. Below it, a 'Description' section contains a warning about transaction IDs and a note for proofreaders. At the bottom, a link points to the 'Minecraft Launcher' project.

Abbildung 11-6: Crowdin zeigt an, zu wie viel Prozent das Projekt schon welche Sprache spricht.

The screenshot shows the Weblate interface for the F-Droid project. At the top, there are navigation links for 'Hosted Weblate', 'Übersicht', 'Projekte', 'Sprachen', and 'Qualitätsprüfungen'. There is also a '+ Hinzufügen' button and a user profile icon. Below the header, there is a search bar and a filter button for 'Übersicht'.

The main content area shows a table of translation statistics for different components:

Komponente	Übersetzt	Nicht übersetzt	Nicht übersetzte Wörter	Qualitätsprüfungen	Vorschläge	Kommentare
Etar-Calendar/Strings — Deutsch Apache-2.0	✓					<a href="#">Übersetzen</a>
F-Droid/Data — Deutsch AGPL-3.0	93%	176	705	26	1	<a href="#">Übersetzen</a>
F-Droid/F-Droid — Deutsch GPL-3.0	✓			13	48	<a href="#">Übersetzen</a>
F-Droid/F-Droid metadata — Deutsch GPL-3.0	✓				3	<a href="#">Übersetzen</a>
F-Droid/F-Droid Server — Deutsch ASPL-0.3	✓			60	2	<a href="#">Übersetzen</a>

Abbildung 11-7: Weblate bietet Werkzeuge für Übersetzer.

## Zeigen, wo man steht – Badges

Abzeichen oder auch *Badges* für ein Repository haben wir im Abschnitt »Weitere Tipps, um dein Projekt attraktiv zu machen« auf Seite 111 in Kapitel 6 bereits kennengelernt. Vielleicht ist dir auch aufgefallen, dass sie sehr unterschiedliche Informationen beinhalten können – beispielsweise welche Version gerade aktuell ist, wie viele Downloads eine Software hat, wie viele Pull-Requests aktuell noch offen sind oder wie die aktuelle Lizenz lautet (siehe Abbildung 11-8, der Verhaltenskodex *Contributor Covenant Code of Conduct*, den du bereits aus Abschnitt »Code of Conduct« auf Seite 107 in Kapitel 6 kennst, hat beispielsweise auch einen eigenen Badge).



Abbildung 11-8: Viele Projekte haben sogenannte Badges in ihre README.md integriert.

So ein Badge ist mit wenig Aufwand selbst erstellt: einfach auf der Seite <https://shields.io/> nach interessanten Badges Ausschau halten – entweder über die am oberen Rand etwas unauffällig platzierte Suchfunktion oder über die Filtermöglichkeiten nach Kategorien – und den entsprechenden Badge anklicken (siehe auch Abbildung 11-9). In der Regel wirst du dann noch gefragt, für welchen Usernamen und welches Repository du den Badge erstellt haben möchtest. Danach kannst du dir einen Link kopieren, den du an geeigneter Stelle einbinden kannst, beispielsweise in der *README.md* deines Repositorys oder aber auch auf deiner Homepage. Netterweise bietet dir shields.io die Möglichkeit an, den Link in verschiedenen Varianten zu kopieren, beispielsweise als HTML oder Markdown.

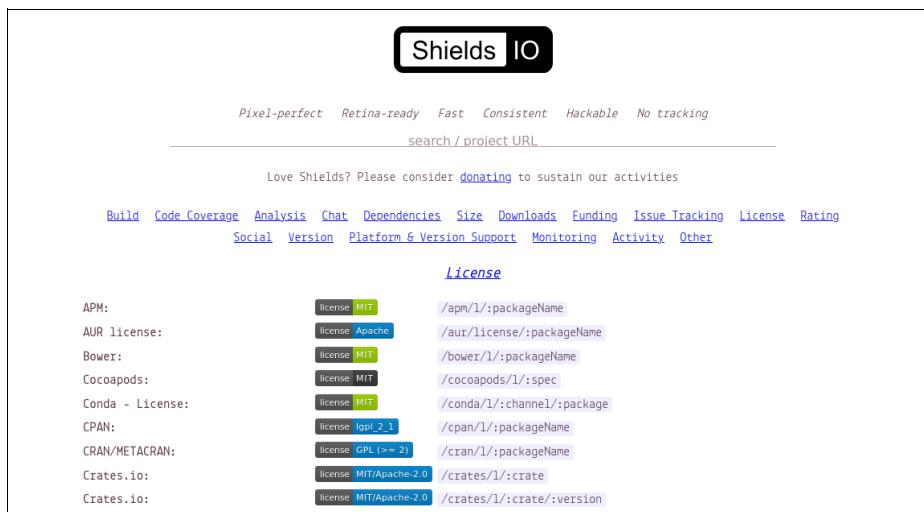


Abbildung 11-9: Auf shields.io kann man sich die Badges zusammensuchen, die man für das eigene Projekt für wichtig erachtet.

Kleiner Hinweis am Rande: Die Badges sind nicht nur für GitHub gedacht. Daher kann es sein, dass du Dinge gefragt wirst, die du gar nicht beantworten kannst. Hier solltest du einfach noch mal genau anschauen, was für einen Badge du dir ausgesucht hast. In der Regel sind Badges für GitHub auch mit *GitHub* beschrieben.

## Sag es mit einem Bild – Gitmoji

Wer ein bisschen mehr Farbe bzw. Bilder in die eigenen Commits bringen möchte, sollte sich mal *Gitmoji*<sup>33</sup> anschauen (siehe Abbildung 11-10). Die Idee ist, zu jedem Commit ein aussagekräftiges kleines Bild in die Commit-Nachricht einzufügen, anhand dessen man sofort sieht, ob der Commit für eine Übersetzung, ein neues Feature oder zum Bugfixing dient. Es gibt dafür sogar ein Kommandozeilenprogramm,<sup>34</sup> um auch direkt auf der Kommandozeile die Gitmojis einzufügen zu können.

Eine Warnung erscheint mir aber angebracht: So hübsch das Ganze auf GitHub aussieht, so nervig kann es sich eventuell in Git präsentieren, da für jedes Gitmoji dessen Kürzel (:kuerzel\_des\_gitmojis:) aufgeführt wird. Das kann dann beispielsweise so aussehen (verkürzte Darstellung):

```
* 6be4321a (15 hours ago) [E] gitmojiUser: :sparkles: add new feature X
* 6be4321b (14 hours ago) [E] gitmojiUser: :globe_with_
meridians: add spanish translation
```

Vielleicht ist das aber für dein Projekt genau das Richtige.



### Achtung bei Einsatz von Gitmoji in fremden Projekten

Bei fremden Projekten solltest du unbedingt erst anfragen, ob du Gitmoji einsetzen darfst, da du damit womöglich den Arbeitsfluss von anderen stark störst.

<a href="#">LICENSE</a>	<a href="#">Update LICENSE</a>
<a href="#">README.md</a>	<a href="#">Update build status badge</a>
<a href="#">next.config.js</a>	<a href="#">Move website to Next.js (#368)</a>
<a href="#">package.json</a>	<a href="#">Bring back copy to clipboard (#372)</a>
<a href="#">yarn.lock</a>	<a href="#">Bring back copy to clipboard (#372)</a>

Abbildung 11-10: Gitmojis bringen etwas Farbe in Commits auf GitHub – auf der Git-Kommandozeile blähen sie die Commit-Nachricht allerdings auf.

33 <https://gitmoji.carloscuesta.me/>

34 <https://github.com/carloscuesta/gitmoji-cli>

# Ideen für eigene Repositories – ohne programmieren

Ganz zu Beginn habe ich bereits geschrieben, dass man nicht programmieren können muss, um GitHub sinnvoll nutzen zu können. Da es manchmal einfacher ist, sich von anderen inspirieren zu lassen, habe ich in diesem Abschnitt ein paar Ideen gesammelt, die andere bereits hatten. Manche habe ich mit einem Beispiel-Repo verknüpft, andere nicht. Das liegt zum Teil daran, dass ich einige Repos im Laufe der Recherche zwar gefunden, diese aber blöderweise nicht notiert habe. Ich weiß, sie sind irgendwo da draußen ...

- Du möchtest Konfigurationsdateien (beispielsweise für einen Linux-Arbeitsplatz, dein Smart Home oder eingesetzte Software) irgendwo im Internet zwischenlagern, um bei Bedarf diese wieder (automatisiert) herunterladen zu können<sup>35</sup> und auch anderen zur Verfügung zu stellen.
- Du willst eine Stilanleitung (*Style Guide*) für das Schreiben von Code oder Ähnliches veröffentlichen oder gemeinschaftlich an einer arbeiten.<sup>36</sup>
- Du hast ein Buch geschrieben und möchtest zusätzliche Infos verbreiten (häufig Quellcode, es geht aber auch anders).<sup>37</sup>
- Du betreibst einen eigenen Videokanal und möchtest Dokumente oder Ähnliches mit deinen Zuschauerinnen teilen.<sup>38</sup>
- Ein Festival, Schulausflug, Skiausflug oder Ähnliches steht an, und du willst gemeinsam mit deinen Freunden eine Packliste erstellen.
- Du möchtest ein Kunstprojekt starten und schauen, wer so alles unterstützen könnte (zum Beispiel gemeinschaftlich ein Gedicht schreiben).
- Du möchtest wichtige Informationen teilen, etwa über die Ausbreitung eines Virus, so erlebt Anfang 2020 mit dem Corona-Virus.<sup>39</sup>
- Du möchtest eine Sammlung von etwas erstellen und dir von der GitHub-Community dabei helfen lassen, beispielsweise beim Sammeln von freien Tutorials und Büchern zum Thema Programmieren.<sup>40</sup>

Und noch ein Tipp zum Schluss: Da GitHub (und auch Git) besonders gut und effizient mit reinen Textdateien umgehen kann (siehe auch Abschnitt »Binärdateien mit Git verwalten« auf Seite 156 in Kapitel 7), sollten Dateien – auch in programmierfreien Repositories – lieber Markdown-Dateien als Word-Dokumente sein.

---

<sup>35</sup> So etwas wird auch Konfigurations-Repository genannt.

<sup>36</sup> Siehe beispielsweise Google: <https://github.com/google/styleguide>.

<sup>37</sup> Der Autor Simon Monk beispielsweise verbreitet darüber auch Bauanleitungen für Elektroniksachen, siehe <https://github.com/simonmonk>.

<sup>38</sup> Siehe beispielsweise <https://github.com/iamshaunjp/oauth-playlist> – das Bemerkenswerte an diesem Repo ist, dass der Projekteigner Branches nutzt, um die einzelnen Lektionen abzubilden.

<sup>39</sup> <https://github.com/CSSEGISandData/COVID-19>

<sup>40</sup> <https://github.com/EbookFoundation/free-programming-books>

Damit sind wir jetzt am Ende unserer gemeinsamen Reise angelangt. Ich hoffe, ich konnte dir einen guten ersten Einstieg in die wunderbare Welt der Free- und Open-Source-Software unter GitHub bieten. Es würde mich freuen, wenn ich dich mit diesem Buch dabei unterstützen könnte, auf GitHub zukünftig als Contributor, Maintainer oder gar Projekteignerin tätig zu werden.

## ANHANG A

# Gängige Git-Befehle zum Nachschlagen

Wörter in durchgehenden GROSSBUCHSTABEN sind durch eigene Werte zu ersetzen.

### Einrichten

Name und E-Mail für *alle* Git-Repositories festlegen, die später in der Git-Historie erscheinen:

```
$ git config --global user.name "VORNAME NACHNAME"  
$ git config --global user.email "USER@MAIL.DE"
```

Name und E-Mail für *ein* Git-Repository festlegen, in dem man sich aktuell befindet:

```
$ git config --local user.name "VORNAME NACHNAME"  
$ git config --local user.email "USER@MAIL.DE"
```

Kommandozeile einfärben:

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

Standardeditor festlegen am Beispiel *vim*:

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

Konfigurationsparameter auslesen am Beispiel der Mailadresse:

```
$ git config --local user.email
```

Die aktuell installierte Git-Version ermitteln:

```
$ git --version
```

### Git LFS einrichten

Git LFS installieren:

```
$ git lfs install
```

Fügt eine Datei oder ein Verzeichnis DATEI\_VERZEICHNIS der Bearbeitungsliste von Git LFS hinzu:

```
$ git lfs track DATEI_VERZEICHNIS
```

Anzeigen, welche Dateien und Verzeichnisse von Git LFS bearbeitet werden:

```
$ git lfs track
```

### Neues Repository

Ein existierendes Verzeichnis als Git-Repository festlegen:

```
$ git init
```

Ein Repository unter der Webadresse WEBSITE lokal kopieren und als Git-Repository festlegen:

```
$ git clone WEBSITE
```

### Git Basics – Staging/Commit

DATEI in den Staging-Bereich überführen:	\$ git add DATEI
Den Staging-Bereich mit der Commit-Nachricht NACHRICHT in das Repository committen:	\$ git commit -m "NACHRICHT"
Alle geänderten Dateien in den Staging-Bereich überführen und mit NACHRICHT committen:	\$ git commit -a -m "NACHRICHT"
Informationen zum aktuellen Zustand des Git-Repositorys, beispielsweise zu unversionierten Dateien:	\$ git status
Anzeigen, was verändert wurde, aber noch nicht im Staging-Bereich ist:	\$ git diff
Anzeigen, was im Staging-Bereich ist, aber noch nicht committet wurde:	\$ git diff --staged

### Branches

Listet alle Branches auf, der aktuelle Branch wird mit einem * markiert:	\$ git branch
Erzeugt einen Branch NEUER_BRANCH:	\$ git branch NEUER_BRANCH
Erzeugt einen Branch NEUER_BRANCH abzweigend vom Branch ALTER_BRANCH:	\$ git branch NEUER_BRANCH ALTER_BRANCH
Wechselt auf den Branch BRANCH:	\$ git checkout BRANCH
Erzeugt den Branch NEUER_BRANCH und wechselt auf ihn:	\$ git checkout -b NEUER_BRANCH

### Remote-Repositories / Merge

Verknüpft ein fernes Repository auf der WEBSEITE mit dem Kürzel ALIAS:	\$ git remote add ALIAS WEBSEITE
Zeigt Informationen zu allen fernen Repositories an:	\$ git remote -v
Zeigt Informationen zum fernen Repository REMOTE_REPO an (Webadresse oder Alias):	\$ git remote show REMOTE_REPO
Bringt die Änderungen auf dem Branch LOKALER_BRANCH auf das ferne Repository REMOTE_REPO (Webadresse oder Alias) oder erzeugt einen Branch LOKALER_BRANCH auf dem fernen Repository, falls noch nicht existent:	\$ git push REMOTE_REPO LOKALER_BRANCH
Bringt die Änderungen auf dem Branch LOKALER_BRANCH auf das ferne Repository REMOTE_REPO (Webadresse oder Alias) und verknüpft den lokalen mit dem Remote-Branch; erzeugt einen Branch LOKALER_BRANCH auf dem fernen Repository, falls noch nicht existent:	\$ git push -u REMOTE_REPO LOKALER_BRANCH
Holt alle Änderungen vom fernen Repository REMOTE_REPO:	\$ git fetch REMOTE_REPO
Überführt Änderungen auf dem Branch BRANCH in den aktuellen Branch:	\$ git merge BRANCH
Abrufen und Zusammenführen aller Commits aus dem Remote Tracking Branch:	\$ git pull

## ANHANG B

# Quellcode

Quellcode für eine eigene Action, die automatisch neue Issues labelt (siehe in Kapitel 9 den Abschnitt »Eine eigene Action erstellen (für Fortgeschrittene)« ab Seite 212).

## Datei .github/workflows/issuelabeler.yml

```
name: labeling_workflow

on:
  issues:
    types: [opened, edited, reopened]

jobs:
  label_issue:
    runs-on: ubuntu-latest

  steps:
    - uses: actions/checkout@v1

    - name: Print Hello world
      run: echo Hello world!

    - name: label-step
      uses: ./github/actions/issuelabeler
      with:
        repo-token: ${{secrets.GITHUB_TOKEN}}
```

## Datei .github/actions/issuelabeler/action.yml

```
name: "Issue labeler"
description: "automatically labels any new issue with the label 'new'"

runs:
  using: "node12"
  main: "index.js"
```

## Datei .github/actions/issuelabeler/index.js

```
const core = require("@actions/core");
const github = require("@actions/github");

async function run() {
  try {

    const [owner, repo] = process.env.GITHUB_REPOSITORY.split("/");
    const token = core.getInput("repo-token");
    const issue = github.context.payload.issue.number;
    const octokit = new github.GitHub(token);

    const label = ["new"];

    // Label issue
    const response = await octokit.issues.addLabels({
      owner,
      repo,
      issue_number: issue,
      labels: label
    });

    } catch (error) {
      core.setFailed(error.message);
    }
  }

run();
```

## ANHANG C

# Glossar (oder: Was bedeutet noch mal ...?)

## A

Eine **Action** ist ein in JavaScript oder innerhalb eines Docker-Containers geschriebenes Programm, das ein bis mehrere Aktivitäten durchführt, z.B. »alle Issues labelt«. Actions kann man selber schreiben oder über den GitHub Marketplace beziehen.

**API** steht für *Application Programming Interface*, zu Deutsch »Schnittstelle für die Programmierung von Anwendungen«. Damit ist eine Schnittstelle gemeint, die beispielsweise einen Kartendienst bereitstellt, sodass eine andere Applikation darauf zugreifen und die Kartendaten verarbeiten kann.

**App** ist das Kurzwort für *Applikation*. Man könnte auch Anwendung oder Software dazu sagen. Mit App sind meist Applikationen auf dem Smartphone oder dem Tablet gemeint, aber auch Applikationen auf dem Rechner werden mittlerweile immer häufiger so genannt.

## B

Mit **Best Practice**, zu Deutsch etwa »beste Praktik«, sind Praktiken gemeint, die sich bei (vielen) anderen bewährt haben. Anstatt das Rad neu zu erfinden, schaut man sich an, wie andere ein bestimmtes Thema bearbeitet haben, z.B. das Lösen eines bestimmten Problems.

**Branch**, zu Deutsch »Zweig«. Wird eingesetzt, um neuartige oder auch experimentelle Weiterentwicklungen in einer isolierten Umgebung auszuprobieren. Diese bezeichnet man dann zumeist als »Feature-Branch« oder auch »Topic-Branch«. Branches für das Beheben von Fehlern nennt man »Bugfixing Branches«.

**Bug**, wörtlich übersetzt »Käfer«, beschreibt einen Fehler in einer Software.

**Bugfix** ist in der Regel ein Stück Softwarecode, der einen Fehler reparieren soll.

**Bugfixing Branch**, siehe Branch.

## C

**CI/CD** steht entweder für *Continuous Integration/Continuous Delivery* oder für *Continuous Integration/Continuous Deployment* (siehe jeweils dort).

**CLI** steht für *Command Line Interface* und könnte mit »Kommandozeilenschnittstelle« übersetzt werden. Es bedeutet, dass die Kommandozeile für die Interaktion mit den Anwender\*innen genutzt wird. Dem steht das *Graphical User Interface* (GUI) gegenüber, bei dem eine grafische Benutzeroberfläche zum Einsatz kommt.

**Continuous Delivery** bedeutet, dass Codeänderungen kontinuierlich in produktivnahen Umgebungen bereitgestellt werden. Man könnte jederzeit die neueste Version der Software ausliefern. Durch die kontinuierliche Bereitstellung der Software werden Reibungspunkte minimiert, die mit den Bereitstellungs- oder Freigabeprozessen verbunden sind.

**Continuous Deployment** bedeutet einen Schritt mehr im Vergleich zu Continuous Delivery: Codeänderungen werden nicht mehr in produktivnahen Umgebungen, sondern direkt in der produktiven Umgebung bereitgestellt.

**Commit**, zu Deutsch »Übergabe« oder »etwas übergeben«. In GitHub kann es als Synonym zu »Speichern« verstanden werden, generell bedeutet es, etwas in ein Repository zu übergeben.

**Continuous Integration** bedeutet, dass Entwickler\*innen ihren Code möglichst früh und oft in ein gemeinsames Repository einbringen. Durch automatisierte Tests können dadurch frühzeitig Integrationsfehler entdeckt werden.

**Copyleft** beschreibt das Konzept, bei dem alle Freiheiten einer Software »weitervererbt« werden. Diese Freiheitsgarantien können darin bestehen, die Software zu verwenden, zu verbreiten, zu verändern und den Quellcode zu verstehen.

## E

**Eingabeaufforderung**, siehe Kommandozeile.

## F

**Feature-Branch**, siehe Branch.

**Fork**, zu Deutsch »Abspaltung« oder »Aufgabelung«. Wird bei GitHub verwendet, um fremde Projekte in den eigenen Account zu kopieren. Der Begriff wird auch genutzt, um aufzuzeigen, dass eine Software eine Abspaltung einer anderen ist, beispielsweise »Nextcloud ist ein Fork von ownCloud«.

## I

**Issue**, zu Deutsch »Thema«, »Aspekt« oder auch »Problem«. Wird bei GitHub und auch sonst im Rahmen der Softwareentwicklung häufig verwendet, um Fehler, Anmerkungen oder auch Wünsche zu einem Projekt zu äußern. Ein Issue kann prinzipiell von jeder beliebigen Person erstellt werden, sie benötigt lediglich einen Account auf GitHub.

## J

Ein **Job** ist Teil eines Workflows. Jobs laufen auf definierten Betriebssystemen und werden parallel abgearbeitet.

## K

**Kanban** ist eine Methode zur Prozesssteuerung, bei der Aufgaben bzw. Prozessschritte nach dem Pull-Prinzip angegangen werden.

Die **Kommandozeile** ist ein Programm, mit dem man textbasierte Befehle ausführen kann, die ausschließlich über die Tastatur eingegeben werden. Wird auch Terminal, Konsole oder Eingabeaufforderung genannt.

**Konsole**, siehe Kommandozeile.

## L

**Logdateien**, häufig auch Logfiles oder schlicht Logs, zu Deutsch »Protokolldateien«, genannt, werden unter anderem beim Programmieren oder beim Hochfahren eines Rechners erstellt. Dort wird protokolliert, was der Rechner so gemacht hat (»Suche Tastatur«) und wo beispielsweise Fehler aufgetreten sind (»Keine Tastatur gefunden«). Solche Dateien werden häufig genutzt, um Problemen auf den Grund zu gehen.

## M

**Master-Branch**, siehe auch Branch, ist der »Hauptzweig« eines jeden Projekts unter GitHub (und Git). Man könnte ihn auch als das »Hauptverzeichnis« des Projekts bezeichnen.

**Mention**, zu Deutsch »Erwähnung«, bedeutet die Erwähnung einer Person in einem Issue oder Pull-Request, indem man »@NAME\_DER\_PERSON« eingibt. Der so markierte Text wird zum einen gesondert hervorgehoben, zum anderen wird die betreffende Person informiert.

**Merge**, zu Deutsch »Verschmelzung« oder »verschmelzen«. Verschmolzen werden in der Regel Weiterentwicklungen, die auf Branches stattgefunden haben, auf den master-Branch. Nach einem Merge kann in der Regel der Branch gelöscht werden.

**Merge Commit** ist ein Commit, der zwei Branches miteinander verschmilzt (mergt).

## P

**Pull-Request**, zu Deutsch grob übersetzt mit »Anfrage zur Übernahme von Quellcodeänderungen in einen anderen Entwicklungszweig in einem Repository«.

**Pull-Request-Merge**, eine besondere Form des Merges.

**PR**, Abkürzung für *Pull-Request*.

## R

**Refactoring** bezeichnet in der Softwareentwicklung das Restrukturieren von Quellcode, um Komplexität zu reduzieren (z.B. Unnötiges entfernen, Komplexes vereinfachen, Zusammengehöriges zusammenführen). Der Code wird dadurch besser lesbar, leichter wartbar und erweiterbar. Das Restrukturieren kann manuell oder auch automatisiert erfolgen.

**Release** beschreibt in der Regel die Version einer Software, die zur Veröffentlichung gedacht ist.

Ein **Remote Tracking Branch** ist ein lokaler Branch, der eine direkte Beziehung zu einem Branch auf einem Remote-Repository hat. Dadurch weiß Git, wie das lokale mit dem Remote-Repository verknüpft ist, sodass beispielsweise `git pull` oder `git push` ohne weitere Parameter nutzbar ist.

**Repository**, zu Deutsch »Aufbewahrungsort«, ist ein Verzeichnis oder allgemeiner ein »Ort«, an dem Dateien zu einem Projekt abgelegt werden. Dateien können alles Mögliche sein: Quellcode, Bilder, Textdokumente. Ein Repository kann lokal sein (auf dem eigenen Dateisystem) oder im Internet, z.B. bei GitHub.

**Repo**, Kurzwort für Repository.

Ein **Runner** ist eine von GitHub gehostete virtuelle Maschine mit einem festgelegten Betriebssystem (Windows, Linux, macOS). Es ist auch möglich, selbst einen Runner zu hosten (*Self-hosted Runner*).

## S

**Semantische Versionierung** ist der Versuch, die Versionierung von Software zu vereinheitlichen, indem genau festlegt wird, welche Zahl einer Version welche Bedeutung hat.

**Single Sign-on (SSO)** beschreibt einen Dienst, der einen automatisch bei anderen Diensten anmeldet. Man benötigt nur noch die Zugangsdaten zum SSO-Dienst.

Ein **Step** ist ein Schritt innerhalb eines Jobs. Schritte können eigene oder fremde Actions sein oder auch Kommandos des entsprechenden Job-Betriebssystems.

## T

**Terminal**, siehe Kommandozeile.

**TLDR** steht für *Too Long, Didn't Read* – zu Deutsch etwa »der Text war mir zu lang, ich habe ihn daher nicht gelesen«. Dies soll darauf hinweisen, dass Menschen sich doch bitte kurz fassen mögen. Trifft man häufig auch in Form einer Zusammenfassung bei Präsentationen oder Ähnlichem an. In einem Arbeitskontext würde man das als *Management Summary* (zu Deutsch »Managementzusammenfassung«) oder *Key Takeaways* (zu Deutsch etwa »die wichtigsten Dinge, die du [von diesem Vortrag] mitnehmen solltest«) bezeichnen.

**Topic-Branch**, siehe Branch.

## V

Mit **Versionsverwaltung** ist in der Regel eine Software gemeint, die Entwickler\*innen mit entsprechenden Werkzeugen dabei unterstützt, Dokumente zu versionieren. Sie ermöglicht damit das gleichzeitige Arbeiten an Dokumenten und verhindert beispielsweise, dass man sich Änderungen gegenseitig überschreibt, nur weil man denselben Dateinamen gewählt hat. Git ist ein Beispiel für eine Versionsverwaltungssoftware.

## W

Ein **Workflow** ist ein benutzerdefinierter automatisierter Prozess, der mehrere Actions und/oder Kommandos ausführen kann.



---

# Index

## Symbole

- .bashrc 144
- .gitattributes 159
- .gitconfig 143
- .github/actions/ 216
- .github/workflows 216
- .gitignore 153
  - Vorlage 154
- @mention 33
- 2FA 23
- 2-Faktor-Authentifizierung 23

## A

- Ablageort für Dateien mit Spezialfunktion 66
- Action 212
  - ACTIONS\_RUNNER\_DEBUG 223
  - ACTIONS\_STEP\_DEBUG 223
- geforktes Repository 222
- Metadatendatei 216
- Template 216
- ACTIONS\_RUNNER\_DEBUG 223
- ACTIONS\_STEP\_DEBUG 223
- Alias 168
- ANSI-Escape-Sequenz 145
- Apache 2.0 License 88
- API
  - GitHub 247
  - Lizenzen 94
- Arbeitsverzeichnis 147
- Asignee 33

## B

- BFG Repo-Cleaner 159
- Binärdatei 157

- Bitbucket 5
- Branch 44
  - Branchverwaltung 51
  - Namenskonvention 46
  - Network Graph 60
  - Protected Branch 67
  - Remote Tracking 176, 262
- Bremer Lizenz für freie Softwarebibliotheken 90

## C

- cache 194
- CircleCI 57
- Closed Source 89
- Code of Conduct 107
- CODE\_OF\_CONDUCT 107
- CODEOWNERS 65
- Collaborator 12, 26
- Commit 28
- Commit-ID 37
- Commit-Nachrichten 28
- commits ahead 52
- commits behind 52
- Commit-Statistik 30
- Community Health Files 105, 125
- Continuous Delivery 56, 197
- Continuous Deployment 56, 197
- Continuous Integration 56, 197
  - CircleCI 57
  - Travis CI 57
- Contributing 109
- CONTRIBUTING.md 109
- Contributor 12
- Copyleft 89
- Creative Commons 91

credential.helper 195  
curl 247

## D

Danger Zone 39  
Dateirechte 140  
datenschutzfreundliche Suchmaschinen 16  
Definition of done 238  
Description 105  
detached HEAD 177  
Deutsche Freie Software Lizenz 90  
Docker 215

## E

echo 166  
Eingabeaufforderung 139  
Einzeldatei herunterladen 18

## F

FabLab 124  
F-Droid 104  
Feature-Branch 47  
Feature-Request 15, 31  
Filter  
    created:2020-02-18 23  
    is:issue 15  
    is:open 15  
    operation:create 23  
FLOSS 2  
Fork 128  
    aktuell halten (GitHub) 133  
FOSS 2  
Free Software Foundation 89

## G

gh repo 246  
Git  
    Hooks 158  
    Large File Storage 156  
    LFS 156  
    mergetool 190  
git  
    add 152, 262  
    branch 155, 262  
    checkout 155, 262  
    clone 174, 261  
    commit 152, 262  
    config 142, 195, 261  
    diff 262

fetch 169, 262  
init 150, 261  
lfs 158, 261  
merge 169, 262  
pull 169, 262  
push 170, 262  
remote 167, 262  
rm 193  
status 151, 262  
GitHub CLI 245  
GitHub Flavored Markdown 26  
GitHub Flow 44  
GitHub Learning Lab 248  
GitHub Pages 225  
    Projekt-Website 227  
    Remote Themes 233  
    Standardlayoutdatei 231  
    Startseite 229  
    Theme Chooser 227  
    User-Website 227  
GITHUB\_TOKEN 221  
GitLab 5  
git-prompt.sh 144  
GNU Affero General Public License 88  
GNU Free Documentation License 90  
GNU General Public License 88  
Good first issue 122

## H

Hackspace 124  
Hash-Wert 37  
HEAD 177  
Home-Verzeichnis 144  
Hooks 158  
HTTPS 194  
hub issue 247

## I

Identicon 21  
Identifikationsnummer 24  
Interaction Limits 83  
Issue 15, 31  
    automatisch schließen 38  
    verlinken 33  
ISSUE\_TEMPLATE.md 79  
IT-Meetup 124  
IT-Stammtisch 124

## J

JavaScript 215  
Jekyll 228  
Job 213  
JSLint 87

## K

Kanban 237  
KeePass2Android 21  
KeePassXC 21  
Kommandozeile 139  
Konfliktmarker 188  
Konsole 139

## L

Label 33, 121  
License 110  
Lizenz  
    Apache 2.0 License 88  
    Bremer Lizenz für freie Softwarebibliotheken 90  
    Creative Commons 91  
    Deutsche Freie Software Lizenz 90  
    GNU Affero General Public License 88  
    GNU Free Documentation License 90  
    GNU General Public License 88  
    JSLint-Lizenz 87  
    MIT License 88  
Locking Conversations 80  
ls 140

## M

Mailadressen, mehrere 71  
Main Branch 44  
man 140  
Markdown 14, 26  
master-Branch 44  
Merge 59, 138  
Merge Commit 45  
Merge-Konflikt 54  
Metrik  
    commits ahead 52  
    commits behind 52  
MFA 23  
MIT License 88  
Mona Lisa Octocat 11  
Multi-Faktor-Authentifizierung 23

## N

Network Graph 60  
Node Package Manager 218  
Node.js 217  
node\_modules 218  
npm 218

## O

Octocat 11  
Open Source, Freiheiten 2

## P

package.json 218  
Passwortgenerator 21  
Passwortmüdigkeit 7  
Passwortsafe 21  
Permissive License 88  
Personal Access Token 170  
private Repository 26  
Projects 236  
Projektboard  
    note 240  
proprietäre Software 89  
Protected Branch 67  
    Allow deletions 74  
    Branch pattern name 68  
    Dismiss stale pull request approvals  
        when new commits are pushed 72  
    Include administrators 70, 209  
    Require pull request reviews before  
        merging 69  
    Require review from Code Owners 73  
public Repository 26  
Pull-Request 53, 128  
    den ersten Pull-Request herausfinden 54  
    fremde Pull-Requests anpassen 64  
    Weiterarbeiten bei offenem Pull-Request 59  
PULL\_REQUEST\_TEMPLATE.md 79  
Pulse 124

## R

README.md 14, 106  
Refactoring 30  
Remote Tracking Branch 176, 262  
Repository XIII  
Required Reviews 68

- Review 61
  - Request changes 72
  - Require pull request reviews before merging 69
  - Require review from Code Owners 73
  - Required Reviews 68
- Ruby 233
- Ruby Gems 233
- runner 220
  
- S**
  - secrets.GITHUB\_TOKEN 221
  - semantische Versionierung 136
  - SHA-1 37
  - Shell 144
  - Single Sign-on 7
  - SLOC 30
  - Softwarepatente 93
  - Source Lines of Code 30
  - SSH 194
  - SSO 7
  - Stackoverflow 16
  - Staging-Bereich 148
  - Standard-Branch 44
  - Step 211, 213
  
- T**
  - TASL 97
  - Template
    - .gitignore 154
    - Action 216
    - Bug report 76
    - Code of Conduct 108
  
- Feature request 76
- Issue 110
- ISSUE\_TEMPLATE.md 79
- Projektboard 240
- Pull-Request 110
- PULL\_REQUEST\_TEMPLATE.md 79
- Repository 235
- Vorausgesetzt – Wenn – Dann (– Und) 238
  
- Terminal 139
- Topics 102
- Travis CI 57
- Trending 120
  
- U**
  - UI 143
  - Unlock conversation 83
  - Urheberrecht 86
  - User Interface 143
  
- V**
  - Verzeichnis auf GitHub anlegen 50
  
- W**
  - Wiki 15
  - Wildcard 68
  - Workflow 212
    - Zeitplan 214
  
- Y**
  - YAML 207
    - Front Matter 232

## Über die Autorin

Anke Lederer ist Fachinformatikerin Systemintegration, Diplom-Informatikerin und zertifizierte Project Management Professional (PMP®). Wenn sie nicht gerade überlegt, wie man ein Holz-Projekt mit dem Arduino oder Raspberry Pi noch besser machen könnte (LEDs gehen immer!), schreibt sie ehrenamtlich Artikel zu IT-spezifischen Themen für die Online-Zeitung Infotechnica ([www.infotechnica.de](http://www.infotechnica.de)).

## Kolophon

Das Tier auf dem Cover von »GitHub – Eine praktische Einführung« ist ein Ozelot (*Leopardus pardalis*). Dieses zu den Pardelkatzen gehörende Säugetier lebt in Mittel- und Südamerika und bevorzugt bewaldetes oder buschiges Gelände, wo es sich verstecken und auf die Lauer legen kann.

Ozelots werden 55 bis 100 Zentimeter groß und sind damit die größten Vertreter dieser Kleinkatzen-Gattung.

Das weiche Fell hat eine orange-bräunliche Grundfärbung, die zum Bauch hin immer heller wird. Auffällige Kreise und Rosetten in Schwarz, die in Streifen angeordnet sind und innerhalb der schwarzen Umrandungen braune oder orange Farbflächen mit schwarzen Punkten bilden, bieten im Unterholz der Bäume eine perfekte Tarnung. Zum Kopf und zu den Gliedmaßen hin werden aus den Kreisen Punkte, die zum Gesicht hin in Streifen auslaufen. Die recht großen Ohren haben auf der Hinterseite jeweils einen weißen Fleck.

Ozelots sind Einzelgänger und kommen nur zur Paarung zusammen. Das Weibchen sucht sich einen geschützten Ort in einer Baumspalte oder Höhle und bringt dort ein bis zwei Junge zur Welt, die ca. drei Monate lang gesäugt werden. Zwei bis drei Jahre durchstreifen die Tiere gemeinsam das Revier der Mutter, werden aber mit der Geschlechtsreife daraus vertrieben. Ozelots jagen bevorzugt kleine Wirbeltiere, verschmähen aber auch Reptilien, Vögel und Fische nicht. Sie können wie alle Katzen sehr gut klettern, jagen aber vor allem am Boden.

Ozelots wurden vor allem in den 60er- und 70er-Jahren stark bejagt, da das Fell – verarbeitet zu Pelzmänteln – vor allem in den USA, später auch in Europa heiß begehrt war. Seit den 90er-Jahren besteht ein völliges Handelsverbot, und die Tierart wurde in das Washingtoner Artenschutzübereinkommen mit aufgenommen. Die Tiere sind in den meisten Ländern ihres Vorkommens geschützt, was sie aber nicht vor illegaler Wilderei bewahrt.

