Bild: Thorsten Hübner

Auf den grünen Zweig gekommen

Praxistaugliche Branching-Konzepte für Git

Die Versionsverwaltung Git erlaubt die Zusammenarbeit von mehreren Entwicklern am selben Projekt. Um Chaos zu vermeiden, braucht man Absprachen, wie man mit Branches umgeht. Diverse Automationen erleichtern dabei die Arbeit.

Von Manuel Ottlik

Viele kleine Git-Projekte funktionieren sehr lange ohne einen zweiten Branch. Gibt es nur einen Entwickler, weiß der am besten, an welcher Baustelle er gerade arbeitet und welche Dateien betroffen sind. Konflikte mit anderen Änderungen sind alleine unwahrscheinlich. Spätestens aber, wenn das Projekt wächst und zwei Entwickler gleichzeitig dieselbe Datei im `master`-Branch, dem Stammpfad von Git, bearbeitet haben, klemmt es beim Push zum Git-Server. Branches können aus der Bredouille helfen: Sie bilden eine Abzweigung vom Stamm und können zu einem späteren Zeitpunkt wieder zusammengeführt werden. In der Zwischenzeit sammelt der Branch Commits, was die Arbeit auf dem `master`-Branch nicht behindert. Ein neuer Branch ist also eine Kopie des aktuellen Stands, an der man in Ruhe arbeiten und experimentieren darf.

Ein Branch für eine neue Funktion kann durchaus monatelang offen bleiben. Andere Kollegen arbeiten derweil an ganz anderen Ecken des Codes. Mit solchen Feature-Branches wird man einer goldenen Regel bei der Arbeit mit Git gerecht: „Don't push to master". Die Idee dahinter: Im `master` liegt immer funktionsfähiger Code, der in diesem Zustand kompiliert, verpackt und zum Kunden oder auf den Server ausgeliefert werden darf. Experimente und Zwischenstufen haben im `master`-Branch nichts verloren.

Aber für welche Änderungen eröffnet man einen neuen Branch und wie organisiert man die Verzweigungen? Git selbst macht keine Vorgaben, wie man Branches

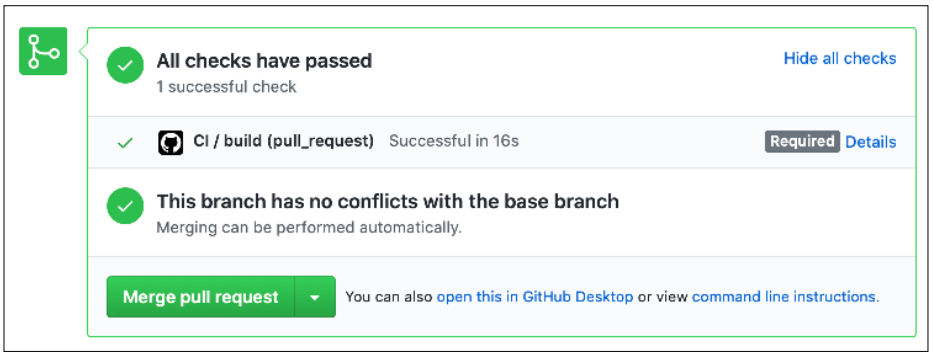
nutzen soll. Man kann sie nach Belieben öffnen, schließen und benennen. Wenn man nicht rechtzeitig einen Plan entwickelt und mit allen beteiligten Entwicklern bespricht, hat man schnell Wildwuchs und wird früher oder später versehentlich den falschen Code ins fertige Produkt schieben.

Git Flow

Wer auf der Suche nach Struktur nach bewährten Branching-Konzepten sucht, stößt schnell auf Git Flow. Dieses Konzept existiert bereits seit über zehn Jahren, wird gern in euphorischen Blog-Posts beschrieben und sieht gleich fünf verschiedene Arten von Branches vor, um die Versionsverwaltung einer Anwendung zu organisieren. Wenn Sie aber nicht zufällig die nächste Mondfahrt planen, reicht vermutlich auch ein schlankeres Konzept.

Git Flow befolgt die goldene Regel zum Schutz des `master`-Branch, allerdings auch noch einige andere – die eine mehr, die andere weniger golden. Neben dem `master`-Branch, der auch in diesem Konzept für die Auslieferung verantwortlich ist, sieht es den `develop`-Branch vor, in dem sich sämtliche Entwicklung abspielt. Wenn Sie eine neue Funktion in die Software einbauen möchten, würden Sie vom `develop`-Branch einen neuen `feature`-Branch abzweigen – die dritte Kategorie. Innerhalb dieses Branches entwickeln Sie dann Ihre neue Funktion, bis sie auslieferungsreif ist. Um die Funktion in die fertige Software zu übernehmen, hat Git Flow die vierte Art von Branch parat: den `release`-Branch. Wenn Sie beispielsweise nur vierteljährlich eine neue Version veröffentlichen, würden hier alle neuen Funktionen der letzten drei Monate gesammelt und getestet. Ist das abgeschlossen, wird der `release`-Branch in den `master`-Branch überführt.

Dann folgt die fünfte und letzte Kategorie: der `hotfix`-Branch. Wenn Sie einen Fehler in Ihrer ausgelieferten Software finden, der schnell repariert werden muss, wird vom `master`-Branch abgezweigt, der Bug beseitigt und wieder auf den `master` gemergt. Zwischendurch müssen alle Änderungen, die in `release`- oder `hotfix`-Branches gemacht werden, auch in den `develop`-Branch gemergt werden, damit Probleme nicht später unerwartet wieder auftauchen. All das erfordert viel Disziplin und am besten ein Teammitglied, das sich vorwiegend um die Ordnung im Repository kümmert. Für kleine und mittlere Projekte ist das utopisch.



Sorgfältig eingerichtete automatische Tests geben ein gutes Gefühl, wenn man Code übernimmt.

Eingestaubt

Sie haben den Überblick verloren? Das passiert vielen, die mit Git Flow experimentiert haben. Die vielen Abzweigungen basieren auf Annahmen, die für die meiste Software heutzutage schlicht nicht mehr gelten. Statt Software auf DVDs zu brennen und an Kunden zu schicken, haben viele Entwickler auf CI/CD-Pipelines (Continuous Integration/Continuous Delivery) umgestellt, die Code automatisch bauen und fertige Programme in kurzen Zyklen über das Internet zum Kunden transportieren oder automatisch auf einem Server installieren.

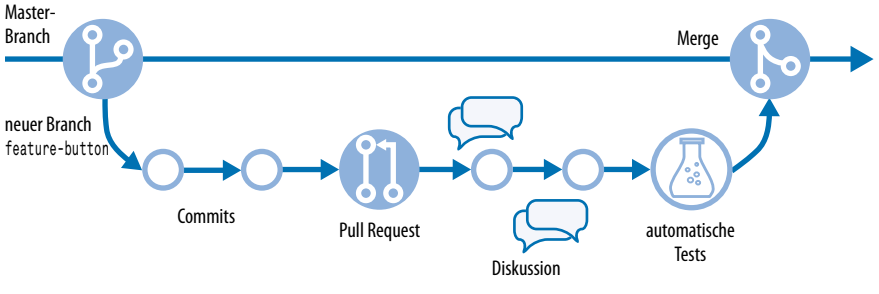
Für Entwickler bedeutet das: Statt Änderungen an der Software bis zum nächsten großen Release zu sammeln, gemeinsam zu testen und zu festgelegten Terminen zu veröffentlichen, werden kleine Änderungen immer sofort getestet und so schnell wie möglich veröffentlicht. Somit werden `release`-Branches schon einmal größtenteils überflüssig. Wenn jede Änderung in kleinen Häppchen direkt in Produktion geht, werden auch Hotfixes überflüssig – letztendlich sind das nur beschleunigte `feature`-Branches. Wenn immer nur die aktuellste Version einer Software betrieben wird, müssen die Fehlerbehebungen auch nicht von veralteten Versionen abzweigen, sondern können immer auf der neuesten Version der Software basieren – „Forward Fixing“ statt „Rolling Back“ nennt sich dieses Prinzip. Diese Hotfixes werden bei Git Flow vom `master`-Branch abgezweigt und dann in den `develop`-Branch integriert. Der wird aber genauso überflüssig, wenn alle umliegenden Branches wegfallen. Übrig bleiben also der `master`-Branch und `feature`-Branches. Klingt gleich viel attraktiver, oder?

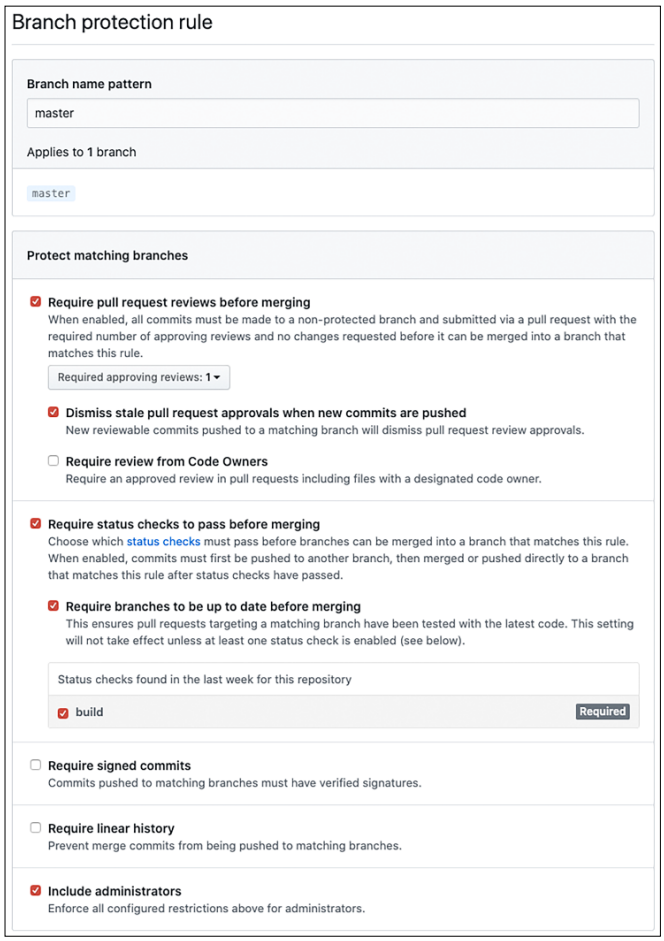
GitHub Flow

Auch die Entwickler der größten Git-Plattformen GitHub und GitLab haben erkannt, dass Git Flow für die meisten Projekte eine Nummer zu groß ist. Sie haben

GitHub Flow

Um eine neue Funktion zu entwickeln, eröffnet der Entwickler einen neuen Branch. Ist die Funktion fertig, erstellt er einen Pull Request und diskutiert ihn mit Kollegen. Automatische Tests stellen sicher, dass der Code funktioniert, bevor er im Master-Branch landet.





Wer sein Projekt mit GitHub Flow organisiert, muss sich und seinen Kollegen untersagen, direkt in den master-Branch zu schreiben. Mit Actions wird Code automatisch geprüft.

deshalb jeweils ihre eigenen Konzepte „GitHub Flow“ beziehungsweise „GitLab Flow“ veröffentlicht, die letzendlich sehr ähnliche Konzepte beschreiben: Ausgehend vom `master`, der immer eine auslieferungsfähige Version enthält, werden `feature`-Branches abgezweigt, die entweder Funktionalität hinzufügen oder Fehler beheben können. Wie Sie den jeweiligen Branch nennen, ist dabei völlig egal und bestenfalls eine interne Absprache – Branches für neue Funktionen können mit `feature-` beginnen, Bugfixes mit `fix-`.

In einem solchen Branch arbeiten Sie so lange an der Neuerung, bis Sie der Meinung sind, dass Sie fertig sind. Dann reichen Sie einen Pull-Request auf GitHub (oder einen Merge-Request auf GitLab) ein. Dieser enthält im Idealfall einen aussagekräftigen Titel und beschreibt kurz, warum Sie diese Änderung vorschlagen. Wenn Sie Continuous-Integration-Werkzeuge wie GitHub Actions benutzen, sollen an dieser Stelle auch alle automatischen Tests durchlaufen, um zu beweisen, dass Ihr Code den Anforderungen entspricht, oder im Falle eines Fehlers einen Merge verhindern. Somit stellen Sie automatisiert sicher, dass nur lauffähiger Programmcode in den `master` integriert wird. Eines Ihrer Teammitglieder kann Ihre Änderungen am Code dann begutachten und entscheiden, ob Korrekturen nötig sind.

Ist der Kollege einverstanden, kann er Ihren Feature-Branch per Knopfdruck in `master` integrieren. Um Releases zu erzeugen, können Sie die Tags von Git benutzen, um bestimmte Punkte in der Versionshistorie zu kennzeichnen. Sowohl GitHub als auch GitLab bieten darüber hinaus zusätzliche Funktionen für Releases an, die Metadaten zu den Git Tags speichern und die Software zum aktuellen Zeitpunkt als Zip-Archiv bereitstellen.

GitHub Flow einrichten

Um ein Repository auf GitHub nach dem vorgestellten Branching-Modell einzurichten, sind nur wenige Schritte erforderlich. Erstellen Sie zum Experimentieren ein neues Repository, indem Sie oben links neben Ihrem Profilbild auf das Plus-Symbol klicken. Wählen Sie Namen und Beschreibung und lassen Sie auch direkt eine README.md erstellen – diese Datei können Sie herbeinehmen, um die Arbeitsweise nachzuvollziehen. Damit `feature`-Branches vor einem Merge auf Qualität geprüft werden, brauchen Sie eine GitHub Action [1]. Klicken Sie dafür oben auf den Reiter „Actions“ und dann in der Kachel „Simple workflow“ auf „Set up this workflow“. Es öffnet sich ein Editor für die Yaml-Datei, in der Actions definiert werden.

Die letzten beiden Absätze der Datei, die GitHub bereits vorbereitet hat, sollten Sie löschen – sie demonstrieren nur, wie man einen Befehl ausführt. Ergänzen Sie stattdessen die folgenden Zeilen:

```
- name: Markdown Linting Action
  uses: avto-dev/markdown-lint@v1.1.0
  with:
    args: '*.md'
```

Damit haben Sie einen Markdown-Linter vorbereitet, also einen statischen Code-Prüfer, der die README.md und alle anderen Markdown-Dateien auf guten Stil für Markdown prüfen wird. In einem Projekt mit Programmcode würden Sie an dieser Stelle Linter und Skripte für Unit Tests für Ihre Programmiersprache platzieren. Speichern Sie die Einstellungen an der Action mit der Schaltfläche „Start Commit“ oben rechts und commiten Sie die Änderung – letztmalig direkt auf den `master`-Branch.

Damit die Qualitätskontrolle greift und Sie zukünftig nach GitHub Flow arbeiten können, müssen Sie sich und Ihren Kollegen nun das Pushen auf den `master`-Branch verbieten. Öffnen Sie dafür die „Settings“, wählen dort den Reiter „Branches“ und legen Sie dort eine neue „Branch protection rule“ an. Geben als Pattern „master“ an.

Den ersten Haken dürfen Sie erst setzen, wenn Sie mindestens einen Kollegen zum Mitwirkenden am Projekt ernannt haben: Um zu verhindern, dass andere ohne die Zustimmung eines anderen Änderungen am `master` vornehmen kann, setzen Sie beim ersten Punkt „Require pull request reviews before merging“ einen Haken. Jetzt muss immer ein anderer Nutzer die Änderungen begutachten und zustimmen, bevor sie in den `master` integriert werden können. In diesem Fall sollten Sie außerdem den ersten Unterpunkt anhaken, damit die Zustimmung sofort ungültig wird, wenn nachträglich noch Commits hinzugefügt werden.

Solange Sie alleine am Repository arbeiten, müssen Sie die beiden Haken aber wieder entfernen – sonst sperren Sie sich selbst aus. Um die soeben eingerichtete GitHub Action als verpflichtende Prüfung zu aktivieren, wählen Sie auch den zweiten Haken sowie dessen Unterpunkte

an. Sie sollten dann eine Liste von Status-Checks sehen – wählen Sie die gerade eingerichtete Aktion mit dem Namen „build“ aus.

Jetzt muss die Linter-Action für Markdown fehlerfrei laufen, bevor man einem Pull-Request überhaupt zustimmen kann. Abschließend sollten Sie noch „Include administrators“ anwählen, damit es auch für Sie als Inhaber keine Extrawürste gibt. Wenn Ihre Regel aussieht wie im Bild auf Seite 144, können Sie den Dialog speichern.

Bei einem Blick auf die bisher durchgelaufenen Actions sehen Sie ein paar fehlgeschlagene Durchläufe. Das liegt daran, dass der Linter gerne eine freie Zeile vor und nach einer Markdown-Überschrift hätte, die angelegte Readme-Datei verstößt gegen diese Regel. Dieses Problem muss beseitigt werden, ein `fix`-Branch muss her.

Wie das flowt

Ein größeres Projekt würden Sie jetzt per Git auschecken, für dieses Beispiel reicht auch der Online-Editor von GitHub: Öffnen Sie die README.md und fügen Sie die leeren Zeilen um die Überschrift hinzu, die der Linter verlangt. Wenn Sie die Branch-Protection-Rule korrekt konfiguriert haben, sehen Sie unten auf der Seite, dass Sie keine Commits auf den `master`-Branch mehr machen dürfen. Stattdessen werden Sie aufgefordert, einen neuen Branch direkt in der Weboberfläche zu erstellen: Geben Sie ihm einen sprechenden Namen, zum Beispiel `fix-readme`. GitHub leitet Sie danach direkt zur Erstellung eines Pull-Request weiter. Sie können aber auch mehrere Commits in dem neuen Branch sammeln und erst dann einen Pull-Request erstellen, wenn Sie das Feature fertig entwickelt haben.

Für den kleinen Schönheitsfehler im Markdown soll diese eine Änderung aber reichen: Erstellen Sie den Pull-Request und schauen Sie zu, wie die Action automatisch losläuft, um Ihre Änderung zu prüfen. Der „Status Check“ sollte jetzt durchlaufen und grüne Haken anzeigen. Wenn Sie die erforderlichen Reviews durch Kollegen deaktiviert haben, können Sie nun direkt den Merge anstoßen. Sie können das Spielchen sogar noch ein bisschen weiter treiben: Wenn Sie vor der Änderung ein Issue in GitHub mit dem Problem erstellt haben, können Sie dieses im Pull-Request auf der rechten Seite des Formulars verknüpfen – dann wird das Issue automatisch geschlossen, sobald der Pull-Request genehmigt wurde.

Fazit

Branching-Konzepte brauchen Disziplin – und sie sollen dem Entwickler die Arbeit erleichtern. Das Prinzip sollte also lauten: so kompliziert wie nötig, aber so einfach wie möglich. Wählen Sie ein zu kompliziertes Modell, besteht die Gefahr, dass Sie es am Ende selbst nicht einhalten. Denn wenn Sie sich die Hälfte der Zeit mit Branches rumschlagen, können Sie die Zeit nicht zum Entwickeln nutzen. Zu einfach sollte er auch nicht sein, sonst kommen sich die Entwickler mit ihren Änderungen in die Quere.

GitHub Flow wählt hier einen guten Mittelweg zwischen einfachem „Push to Master“und dem komplexen Git-Flow. Der Mittelweg ist sogar so gut, dass auch große Open-Source-Projekte damit klarkommen. Im Zusammenspiel mit einer CI-Lösung wie GitHub Actions reduziert man zuverlässig die Gefahr, Fehler in die fertige Software zu schieben. *(jam@ct.de)* c't

Literatur

[1] Merlin Schumacher, Und Actions!, Erste Schritte mit GitHubs CI/CD-Werkzeug Actions, c't 25/2019, S. 164