B1B

Back in Business
Git

Thomas Zenger
mail@thomas-zenger.de
20.08.2017

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Vorwort

1.1 B1B - Was ist das?

Dieses Skript soll als Nachschlagewerk und Cheat Sheet dienen und eine Gedächtnisstütze zu einem bereits vertiefeten Thema bieten. Es soll keine Fachliteratur ersetzen und dient auch nicht zum Erlernen der Thematik, da auf ausführliche Erklärungen größtenteils verzichtet wird.

1.2 Motivation

Diese Idee zu diesem Kompendium enstand während meines Informatikstudiums. Da bereits erlernte Techniken, wie z.B. Programmier- oder Scriptsprachen, mit dem Fortschreiten des Studiums in den Hintergrund traten, zu einem späteren Zeitpunkt jedoch wieder benötigt wurden, war es unerlässlich sich diese wieder ins Gedächtnis zu rufen. Aus diesem Grund entstand dieses Werk als eine Art “erweiterte Zusammenfassung”.

1.3 Literatur und Grundlagen

Folgende Werke fanden bei der Erstellung dieses Dokuments Beachtung. An dieser Stelle soll ausdrücklich erwähnt werden, dass sich diese Arbeit nicht als Plagiat oder Kopie genannter Literatur verstanden werden soll, sondern als Lernhilfe und Zusammenfassung.

- Git Documentation https://git-scm.com/doc

2 Einführung

2.1 Versionskontrolle

Versionskontrollsysteme protokollieren Änderungen an Dateinen und ermöglichen es zu jedem Zeitpunkt auf eine ältere Version zuzugreifen.

2.2 Git

Im Gegensatz zu zentralen Versionskontrollsystemen handelt es sich bei Git um ein verteiltes Versionskontrollsystem (DVCS). Jeder Anwender clont das komplette Repository und erhält somit eine vollständige Kopie des Projektes.

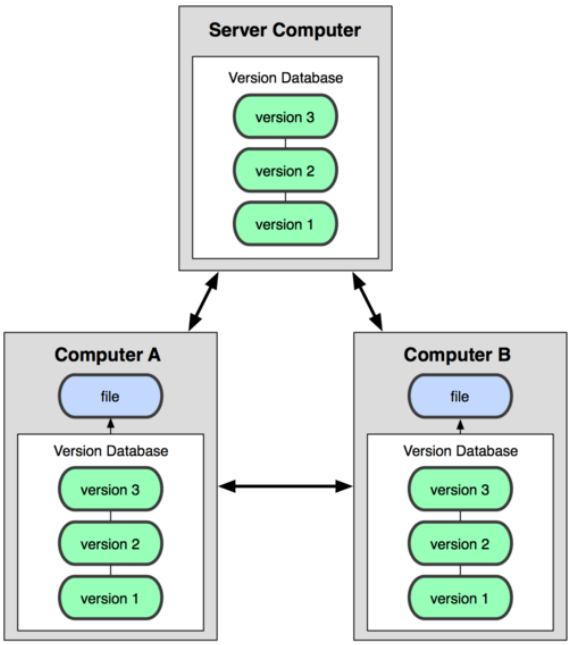

Abbildung 2.1: Verteilte Versionskontrolle

2.2.1 Arbeitsweise

Git speichert bei jedem Commit nur die geänderten Dateien. Nicht geänderte Dateien werden mit einem Verweis auf die alte Version übernommen. Es werden keine Informationen von anderen Rechnern im Netzwerk benötigt, da Git fast alle Operationen lokal ausführt und die Informationen ebenfalls in einer lokalen Datenbank hinterlegt werden. Änderungen werden zur Sicherstellung der Integrität mit Checksummen überprüft.

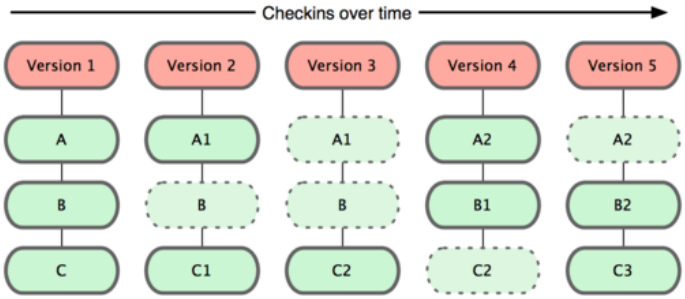

Abbildung 2.2: Git Snapshots

2.2.2 3 Zustände

Git definiert drei Hauptzustände, in denen sich eine Datei befinden kann.

- **commited**
  Daten sind in der lokalen Datenbank gesichert
- **modified**
  Datei wurde geändert, aber noch nicht commited
- **staged**
  Datei im gegenwärtigen Zustand für nächsten Commit vorgemerkt

2.3 Git Config

Git kann per Config Dateien angepasst werden. Der Befehl `git config` führt alle Optionen auf.

2.4 Manpage

Die Manpage hat den Vorteil, dass sie auch offline aufgerufen werden kann. Folgende Befehle stehen zur Auswahl:

```
1 git help <verb>
2 git <verb> --help
3 man git-<verb>
```

Listing 2.1: Git Manpage

3 Grundlagen

3.1 Git Repository anlegen

3.1.1 Existierendes Verzeichnis initialisieren

Ein bereits existierendes Verzeichnis kann als Git Repository initialisiert werden. Dabei wird ein Unterverzeichnis `.git` angelegt. Bereits im Verzeichnis erhaltene Daten werden nicht automatisch versioniert. Dies geschieht mit dem ersten Commit.

```
1 git init
```

Listing 3.1: Initialisierung

3.1.2 Existierendes Repository klonen

Um eine Kopie eines existierenden Projekts zu erstellen wird dieses geklont. Alle bereits vorhandenen Daten werden als lokale Kopie des Projekts gespeichert. Dabei wird ein Verzeichnis mit dem Namen des Projekts angelegt.

```
1 git clone [url]
2
3 #Eigener Verzeichnisname
4 git clone [url] newDir
```

Listing 3.2: Clone

3.2 Änderungen nachverfolgen

Dateien im Arbeitsverzeichnis können entweder verfolgt (tracked) oder nicht verfolgt (untracked) werden. Alle Dateien des letzten Commits befinden sich in der Versionskontrolle. Diese Dateien können unverändert (unmodified), verändert (modified) oder für den nächsten Commit vorgemerkt (staged) vorliegen. Alle anderen Dateien sind nicht versioniert (Nicht im letzten Commit und nicht in der Staging Area).

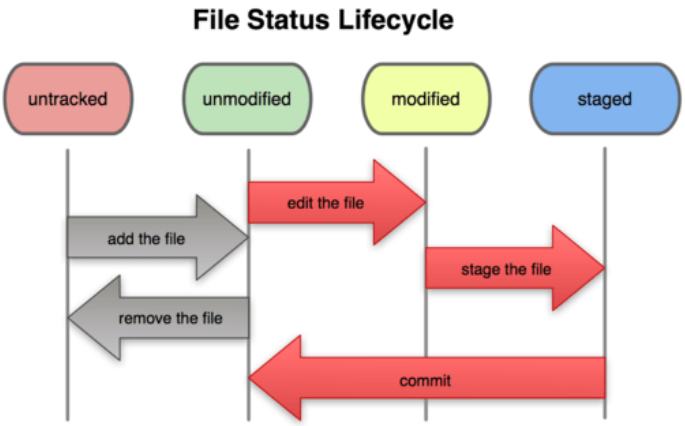

Abbildung 3.1: File Status LC

3.3 Status prüfen

```
1 git status
```

Listing 3.3: Status

3.4 Dateien hinzufügen

```
1 git add newFile
```

Listing 3.4: Add

3.5 Dateien stagen

```
1 git add existingFile
```

Listing 3.5: Stage

3.6 Dateien ignorieren

In der Datei `.gitignore` können Regeln hinterlegt werden, welche Dateien von Git ignoriert werden sollen.

3.7 Staging Area durchsuchen

Um die Staging Area und das Arbeitsverzeichnis direkt zu vergleichen kann `diff` benutzt werden.

```
1 git diff
```

Listing 3.6: Diff

3.8 Commit aus Staging Area erzeugen

```
1 #Editor
2 git commit
3
4 #Editor mit diff
5 git commit -v
6
7 #Meldung direkt angeben
8 git commit -m ''Message from Hell''
```

Listing 3.7: Commit

3.9 Staging Area überspringen

Alle Dateien, welche sich bereits unter Versionskontrolle befinden, werden im Commit aufgenommen.

```
1 git commit -a
```

Listing 3.8: Commit w/o Staging Area

3.10 Dateien entfernen

3.10.1 Staging Area

```
1 git rm --cached file.txt
```

Listing 3.9: RM Staging Area

3.10.2 Arbeitsverzeichnis

```
1 #delete file
2 rm file.txt
3 #staging area
4 git rm file.txt
5 #commit
6 git commit
```

Listing 3.10: RM

3.11 Dateien verschieben

Git verfolgt nicht explizit, ob Dateien verschoben werden.

```
1 git move file_from file_to
```

Listing 3.11: Move

3.12 Historie

Der Befehl `git log` listet alle Commits eines Projekts in umgekehrter chronologischer Reihenfolge auf. Mit Hilfe von Optionen kann bestimmt werden, welche Informationen angezeigt werden sollen.

```
1  #normal
2  git log
3
4  #display changes (diff)
5  git log -p
6
7  #diff by words not lines
8  git log -p --word-diff
9
10 #statistc
11 git log --stat
12
13 #one commit per line
14 git log --pretty=online
15
16 #branch graph
17 git log --pretty=oneline --graph
```

Listing 3.12: Log

Option	Beschreibung
`-p`	Zeigt Patch eines Commits
`--word-diff`	Vergleich Wort zu Wort
`--stat`	Statistik der geänderten Dateien und entfernten/hinzugefügten Zeilen
`--startstat`	Kurzstatistik über eingefügte/entfernte Zeilen
`--name-only`	Liste der geänderten Dateienen nach Commit Informationen
`--name-status`	Liste der Dateien mit hinzufügt/geändert/entfernt Statistik
`--abbrev-commit`	Zeigt nur erste Zeichen einer SHA-1 Checksum
`--relative-date`	Zeigt Datum in relativen Format
`--graph`	ASCII Graph der Branch- und Merch-Historie
`--pretty`	Zeigt Commits in alternativen Format (online, short, full, fuller und format [eigenes Format spezifizieren])

Tabelle 3.1: Log Options Format

3.12.1 Log Dateien Filtern

Außerdem stehen weitere Optionen zur Verfügung, die zur Filterung der Ergebnisse dienen.

Option	Beschreibung
`-(n)`	Ausgabe von n Commits
`--since, --after`	Commits nach dem angegebenen Datum
`-until, --before`	Commits vor dem angegebenen Datum
`--author`	Commits von angegebenen Author
`--committer`	Commits vcon angegebenen Committer
`--no-merges`	Commits welche keine Merges sind

Tabelle 3.2: Log Options Format

3.13 Änderungen rückgängig machen

3.13.1 letzten Commit ändern

Ändert den letzten Commit (vergessene Dateien hinzufügen, Message ändern). Direkt nach dem letzten Commit auszuführen, wenn noch keine weiteren Änderungen gemacht wurden (Staging Area wird verwendet).

```
1 #texteditor new message
2 git commit --amend
3
4 #add forgotten file
5 git commit -m 'initial commit'
6 git add forgotten_file
7 git commit --amend
```

Listing 3.13: letzten Commit ändern

3.13.2 Änderungen aus Staging Area entfernen

```
1 #list files in staging area
2 git status
3
4 #remove file from staging area
5 git reset HEAD <file>
```

Listing 3.14: Entfernen aus Staging Area

3.13.3 Änderungen an einer Datei rückgängig machen

```
1 #file status last commit
2 git checkout -- <file>
```

Listing 3.15: Änderungen an Datei rückgängig machen

3.14 Externe Repositorys

3.14.1 Remote Repositorys anzeigen

```
#display remote server
git remote

#display remote server with url
git remote -v
```

Listing 3.16: Repo anzeigen

3.14.2 Remote Repository hinzufügen

```
#add repo with shortname
git remote add [shortname] [url]
```

Listing 3.17: Repo hinzufügen

3.14.3 Änderunges aus Remote Repository herunterladen und zusammenführen

```
#download changes without merging
git fetch [remote-name]

#download changes with merging (only if branch tracked)
git pull
```

Listing 3.18: Repo zusammenführen

3.14.4 Änderunges hochladen

```
#upload changes
git push [remote-name] [branch-name]
```

Listing 3.19: Änderungen hochladen

3.14.5 Repository ansehen

```
#display remote repository
git remote show [remote-name]
```

Listing 3.20: Repository ansehen

3.14.6 Verweise auf Remote Repository

```
#rename
git remote rename name newName

#remove
git remote rm name
```

Listing 3.21: Verweise auf Remote Repo

3.15 Tags

3.15.1 Tags anzeigen

```
#display tags alphabetical
git tag

#display only versions that belong to 1.x
git tag -l 'v.1.*'
```

Listing 3.22: Tags anzeigen

3.15.2 Tags anlegen

Git bietet zwei verschiedene Typen von Tags an. einfache Tags (lightweight) und kommentierte (annotated) Tags. Ein einfacher Branch ist ein Zeiger auf einen bestimmten Commit. Kommentierte Tags werden als Objekte in Git gespeichert.

```
#lightweight tag
git tag v1.0

#annotated tag
git tag -a v1.0

#annotated tag with message
git tag -a v1.0 -m 'first release version'

#annotated GPG signed tag with message
git tag -s v1.0 -m 'first release version'

#verify signed tag
git tag -v [tagname]

#display tag
git show [tagname]
```

Listing 3.23: Tags anlegen (Teil 1)

```
1 #tag afterwards
2 git tag -a [tagname] -m [message] [hash]
3
4 #upload tag
5 git push origin [tagname]
6
7 #upload all tags
8 git push origin --tags
```

Listing 3.24: Tags anlegen (Teil 2)

3.16 Auto-Vervollständigung

Mit Hilfe eines Bash Scriptes ist es möglich eine Auto-Vervollständigung für die Bash auf Linux Systemen einzurichten. Das Script kann von folgender Quelle bezogen werden:
`https://github.com/git/git/blob/master/contrib/completion/git-completion.bash`
Im Verzeichnis `/etc/bash_completion.d/` ist nun eine Datei mit dem Namen `git-completion.bash` anzulegen. Mit `Tab` kann nun die Autovervollständigung erfolgen (2 mal drücken für Vorschläge).

3.17 Git Aliase

In Git können Aliase dazu verwendet werden um häufig verwendete Befehle abzukürzen.

```
1 git config --global alias.[name] [befehl]
```

Listing 3.25: Alias

4 Branching

4.1 Begriffserklärung

Bei einem Commit speichert Git ein Commit-Objekt ab. Dieses Objekt enthält einen Zeiger auf den Snapshot mit den Objekten der Staging Area, Autor, Metadaten und einen Zeiger auf die direkten Eltern-Commits. Projektverzeichnisse werden als tree-Objekt abgelegt und Projektdateien als Blob gespeichert. Ein Branch in Git ist ein

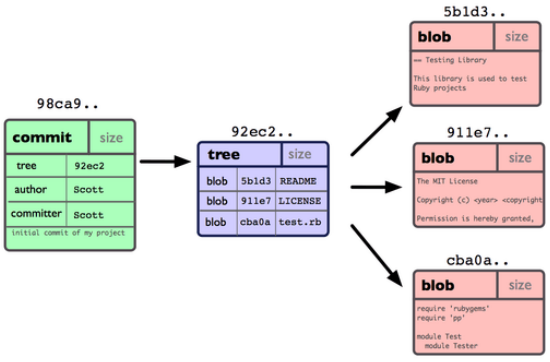

Abbildung 4.1: Commit Daten

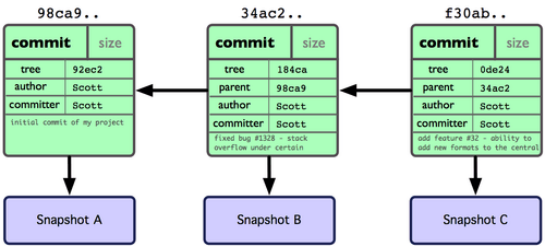

Abbildung 4.2: mehrere Commits

Zeiger auf einen dieser Commits. Der Standard Git Branch heißt `master`. Dieser wird mit dem Initial Commit erstellt. Beim Erstellen eines neuen Branches wird ein neuer Zeiger erstellt. Dieser zeigt auf den gleichen Commit, auf welchem gerade gearbeitet wird. Dies wird über den HEAD-Zeiger realisisert, welcher auf den aktullen lokalen Branch zeigt.

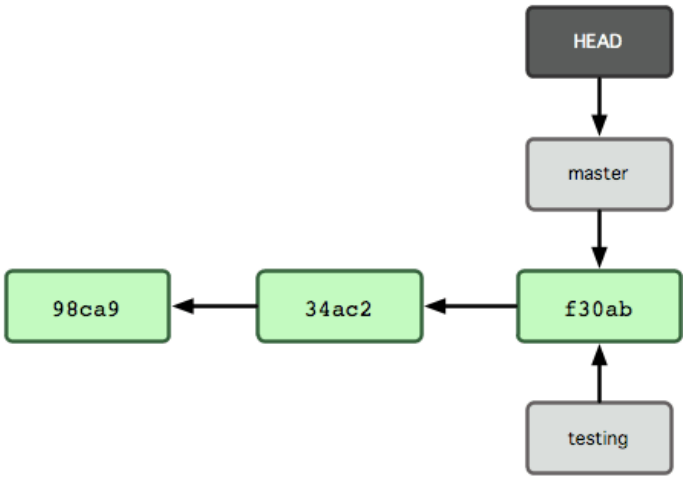

Abbildung 4.3: Branch erstellen

4.2 Branch anlegen

```
1 #create branch
2 git branch [branchname]
3
4 #create branch, activate and switch to it
5 git checkout -b [branchname]
```

Listing 4.1: Branch anlegen

4.3 Branch wechseln

```
1 git checkout [branchname]
```

Listing 4.2: Branch wechseln

4.4 Arbeiten mit Branches

Auf jeden Branch kann unabhängig zu anderen Branches gearbeitet werden. Beim Wechseln des Branches wird lediglich der HEAD Zeiger verschoben und alle Dateien im Arbeitsverzeichnis auf den Stand des letzten Commits des Branches in den gewechselt wird versetzt. Commitete Änderungen des anderen Branches bleiben natürlich erhalten. Da Branches selbst keinerlei Dateien enthalten und nur aus einer kleinen Datei bestehen (SHA-1 Prüfsumme), können diese einfach erstellt und entfernt werden.

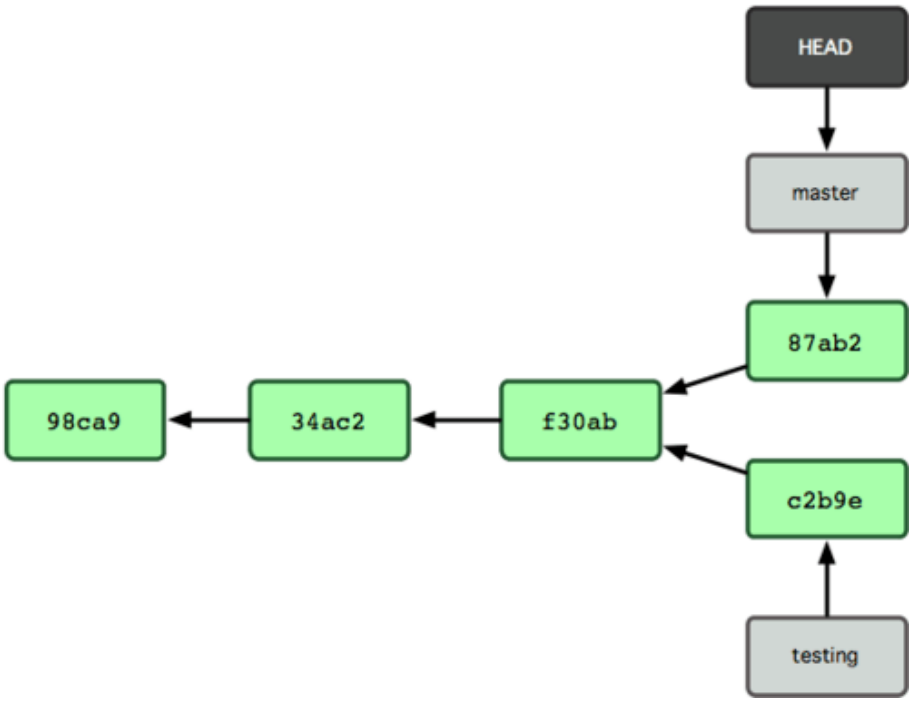

Abbildung 4.4: Verzweigte Branches

4.5 Einfaches Branching und Merging

Änderungen, Verbesserung, etc. sollten nicht im Produktivzweig erstellt werden. Hierzu ist es ratsam einen extra Branch für die Entwicklung zu erstellen. Hierzu ein einfaches Beispiel anhand einer Webentwicklung:

```
1  #Branch erstellen
2  git branch dev
3
4  #Branch wechseln
5  git checkout dev
6
7  #Entwicklung
8  vim index.html
9
10 #Commit
11 git commit -a -m 'Content added'
12
13 #Zum Master Branch wechseln und zusammenfuehren von Master und Dev
14 git checkout master
15 git merge dev
16
17 #Entwicklungsbranch entfernen
18 git branch -d dev
```

Listing 4.3: Einfaches Merging Beispiel

4.6 Grundlagen des Zusammenführens (Merge)

Vorgehen	Beschreibung
`fast forward`	Neuer Commit direkter Nachfolger von ursprünglichen Commit. Einfaches Merging. Zeiger wird einfach auf neuen Commit weiter bewegt (kein neuer Commit).
`recursive strategy`	Kein direkter Nachfolger. 3-Wege-Merge. Neuer Commit wird angelegt. Git ermittelt selbstständig die besten 3 Eltern (merge commit).

Tabelle 4.1: Merge Strategien

4.7 Merge Konflikte

Wurden an den selben Stellen in den selben Dateien unterschiedlicher Branches etwas geändert, können diese Änderungen von Git nicht sauber zusammengefügt werden. Dies führt zu einen Merge Conflict. Diese Dateien werden als `unmerged` aufgelistet und mit Markern versehen um die Konflikte manuell zu lösen. Nach dem Auflösen des Konflikts müssen die Dateien per `git add` der Staging Area hinzugefügt werden. Das Staging markiert sie für Git als gereinigt. Anschließend nochmals mit `git status` überprüfen, ob alle Konflikte aufgelöst wurden und den Merge mit `git commit` abschießen.

4.8 Branch Management

```
# list branches
git branch

#last commit per branch
git branch -v

#display only merged branches
git branch --merged

#display only unmerged branches
git branch --unmerged

#branches with unmerged changes
git branch --no-merged

#delete branch with unmerged changes
git branch -D [branchname]
```

Listing 4.4: Branch Management

4.9 Workflows

4.9.1 Langfristige Branches

Master Branch mit stabilen Code. Entwicklung in seperaten Branch (bzw. auch mehrere Branches möglich). Nur stabiler Code wird in den Master Branch gemerged.

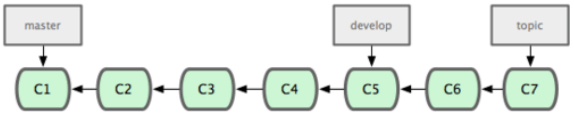

Abbildung 4.5: Langfristige Brances

4.9.2 Themen Branches

Kurzlebige Zweige für die Entwicklung spezieller Features und Funktionen. Diese Branches liegen nur lokal vor.

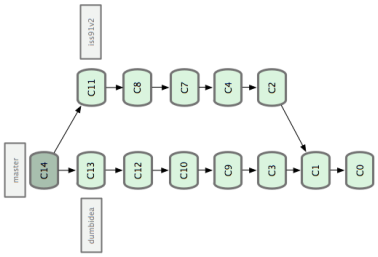

Abbildung 4.6: Themen Branches

4.10 Externe Branches

4.11 Rebasing

In Git gibt es zwei Methoden Änderungen in einen anderen Branch zu überführen. Neben dem `merge` existiert das `rebase` Kommando.

4.11.1 einfaches Rebase

Bei einem einfachen Rebase wird der Vorfahr eines Zweiges geändert. Dies dient dazu einen 3-Wege-Merge zu vermeiden, sodass ein einfacher Fast-Forward-Merge durchgeführt werden kann.

```
1  #Branch to rebase
2  git checkout dev
3
4  #Rebase Branch
5  git rebase master
6
7  #Change to Master and merge
8  git checkout master
9  git merge dev
10
11 #Rebase without Checkout
12 rebase [Basis-Branch] [Themen-Branch]
```

Listing 4.5: Einfaches Rebase

4.11.2 Rebasing auf anderen Branch

```
1  git rebase --onto master server client
```

Listing 4.6: Rebase auf anderen Branch

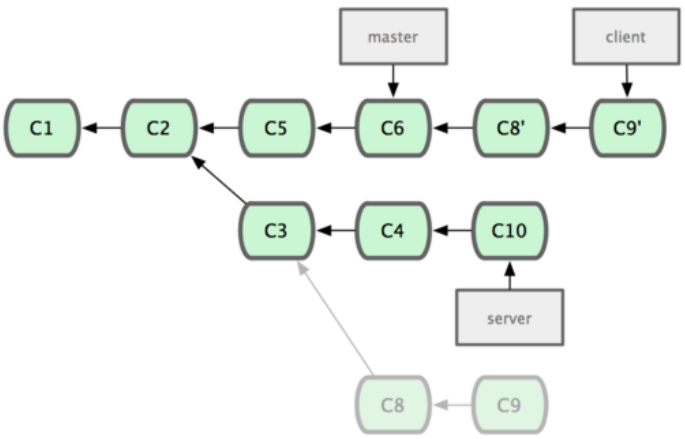

Abbildung 4.7: Themen Branches

Rebase keine Commits die in ein öffentliches Repository hochgeladen wurden

5 Server

5.1 Protokolle

5.1.1 Lokales Protokoll

5.1.2 SSH Protokoll

5.1.3 Git Protokoll

5.1.4 HTTP/S Protokoll

6 Changelog

V1.0 - 2017.08.20

- Erstveröffentlichung