Vorlesung Linux-Praktikum

3. Versionskontrolle mit `git` Teil 2

Dirk Frettlöh

Technische Fakultät
Universität Bielefeld

Git II - Überblick

`git clone`	ein repo auf den eigenen Rechner holen
`git pull, fetch, push`	Änderungen holen / sichern
`git rm, revert, reset`	Änderungen rückgangig machen

Branches:

`HEAD`	Zeiger auf aktuellen *branch*
`git branch`	neuen *branch* erzeugen
`git checkout`	HEAD auf anderen *branch* setzen
`git checkout -b`	kurz für `branch` und `checkout`
`git merge`	*branch*es zusammenfügen

Erinnerung:

- Versionskontrolle
- git - Prinzipien, lokales Arbeiten
  - git init
  - git add; git commit
  - git rm
  - git log, git status

Wie Leute git auch benutzen

xkcd.com

...aber wir nicht.

Git - Verteiltes Arbeiten

Siehe `man giteveryday`:

A developer working as a participant in a group project needs to learn how to communicate with others, and uses these commands in addition to [the ones above]

- git-clone from the upstream to prime your local repository.
- git-pull and git-fetch from origin to keep up-to-date with the upstream.
- git-push to shared repository
- git-format-patch to prepare e-mail submission
- git-send-email to send your e-mail submission
- git-request-pull to create a summary of changes for your upstream to pull

The latest ballyhoo

Anmerkung: Die meisten Menschen benutzen `git` nicht in der Kommandozeile, sondern in einer graphischen Oberfläche (GUI) bzw Entwicklungsumgebung (IDE).

Außerdem gibt es web-basierte git-Umgebungen (bzw *hosting services*).

Davon gibt es viele, und was die populärste ist, ändert sich alle paar Jahre.

~~sourceforge (1999)~~ ~~github (2008)~~ gitlab (2014)

Und bald was anderes: gitea (20??)

The latest ballyhoo

Anmerkung: Die meisten Menschen benutzen `git` nicht in der Kommandozeile, sondern in einer graphischen Oberfläche (GUI) bzw Entwicklungsumgebung (IDE).

Außerdem gibt es web-basierte git-Umgebungen (bzw *hosting services*).

Davon gibt es viele, und was die populärste ist, ändert sich alle paar Jahre.

~~sourceforge (1999)~~ ~~github (2008)~~ gitlab (2014)

Und bald was anderes: gitea (20??)

Daher zeigen wir hier alles auf der Kommandozeile.

Die Prinzipien sehen wir so auch (evtl. sogar klarer)

Git - Verteiltes Arbeiten

Globales Repositorium klonen

Um mit anderen zusammenzuarbeiten gibt es wieder extrem viele Möglichkeiten. Hier aber nur eine häufige.

Eine weitere häufige Möglichkeit ist, dass es einen git-Manager gibt.

Dann reicht es für den Normalnutzer ebenso aus, das Folgende zu wissen.

Ein zentrales ("globales") Repositorium.

Git - Verteiltes Arbeiten

Globales Repositorium klonen

Um mit anderen zusammenzuarbeiten gibt es wieder extrem viele Möglichkeiten. Hier aber nur eine häufige.

Eine weitere häufige Möglichkeit ist, dass es einen git-Manager gibt.

Dann reicht es für den Normalnutzer ebenso aus, das Folgende zu wissen.

Ein zentrales ("globales") Repositorium.

Um die Daten von dort auf den eigenen Rechner zu bekommen (nur einmal am Anfang der Projekts):

`$ git clone https://github.com/pfad/projekt`
(per https) **oder**

`$ git clone juser@files.techfak.de:/pfad/projekt` (per scp) **oder**

`$ git clone /vol/lehre/pfad/projekt` (lokal)

Danach hat man ein neues Verzeichnis. Meist der Name am Ende der Pfade oben (hier also `projekt`)

Git - Verteiltes Arbeiten

Globale Repositorien

Das Repo, das wir klonten, heißt *origin*.

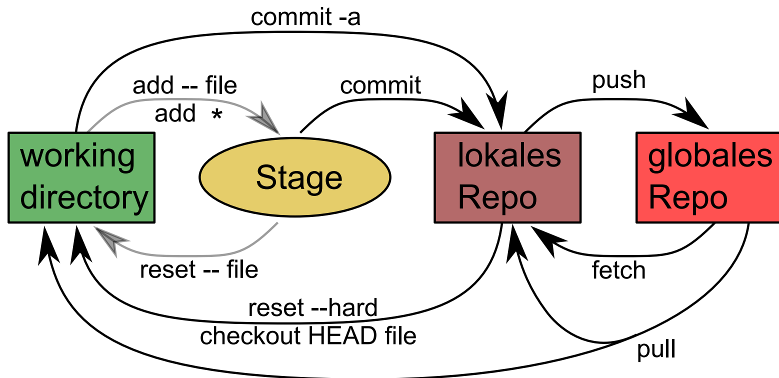
```
$ git remote
origin
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

(Es kann auch mit mehreren globalen Repos gearbeitet werden, das zeigen wir hier nicht.)

Git - Verteiltes Arbeiten

Schema

Erweiterung des Schemas zum lokalen Arbeiten:



Git - Verteiltes Arbeiten

fetch und pull

Zwei Möglichkeiten, die aktuelle Version von origin zu holen:

```
git fetch
```

```
git pull
```

Git - Verteiltes Arbeiten

fetch und pull

Zwei Möglichkeiten, die aktuelle Version von origin zu holen:

`git fetch`

`git pull`

Unterschied:

`fetch` holt die Dateien, aber fügt sie nicht unbedingt mit meinen lokalen Dateien zusammen: parallele Versionen (*branch*es) möglich.

`pull` holt die Dateien und fügt (*merge*t) sie mit meinen lokalen Dateien zusammen.
(Das ist auch das Ergebnis von clone: lokale Version = globale Version, bzw master = origin)

Git - Verteiltes Arbeiten

push und pull

Umgekehrt fügt `push` die lokalen Änderungen dem globalen Repo hinzu (falls ich Schreibberechtigung habe)

```
$ git push
```

(kurz für `git push origin master`, in komplexeren Situationen `git push remote branch`)

Git - Verteiltes Arbeiten

push und pull

Umgekehrt fügt `push` die lokalen Änderungen dem globalen Repo hinzu (falls ich Schreibberechtigung habe)

```
$ git push
```

(kurz für `git push origin master`, in komplexeren Situationen `git push remote branch`)

Ein einfacher Arbeitsablauf wäre also
(ein *branch*, master = origin)

```
1. git clone dfrettloeh@files.techfak.de:projekt
2. Dateien bearbeiten
3. git add Datei ; git commit -m 'kommentar'
4. git pull
5. git push
6. GOTO 2.
```

Git - Verteiltes Arbeiten

push und pull

...wenn alles gutgeht. Was ist, wenn zwei Leute A und B gleichzeitig Dateien ändern?

Git - Verteiltes Arbeiten

push und pull

Fall 1: A ändert Datei eins.dat, B ändert Datei zwei.dat

A pusht zuerst, das wird durchgeführt.

Jetzt ist im globalen Repo eins.dat neu

B pusht danach, das wird nicht durchgeführt.

Git - Verteiltes Arbeiten

push und pull

Fall 1: A ändert Datei eins.dat, B ändert Datei zwei.dat

A pusht zuerst, das wird durchgeführt.

Jetzt ist im globalen Repo eins.dat neu

B pusht danach, das wird nicht durchgeführt.

Also handelt B wie oben: erst `git pull`:

Dann repariert git das: pull ist holen (*fetch*) und zusammenfügen (*merge*)

Kein Problem: eins.dat neu wird geholt, zwei.dat wird durch Bs neue Version ersetzt. Jetzt sind im wd und im lokalen Repo eins.dat neu und zwei.dat neu

Nun kann B `git push` ausführen. (Zeigen)

Jetzt ist auch im globalen Repo zwei.dat neu

git - Verteiltes Arbeiten

push und pull

Fall 2: A ändert Datei eins.dat in Zeile 1, B ändert Datei eins.dat in Zeile 3

A: `git pull ; git push`, das wird durchgeführt.

B: `git pull`:

git - Verteiltes Arbeiten

push und pull

Fall 2: A ändert Datei eins.dat in Zeile 1, B ändert Datei eins.dat in Zeile 3

A: `git pull ; git push`, das wird durchgeführt.

B: `git pull`:

Git repariert auch das: “Merge branch master of [remote]”

(Zeigen)

Git - Verteiltes Arbeiten

push und pull

Fall 3: A ändert Datei eins.dat in Zeile 1, B ändert auch Datei eins.dat in Zeile 1

A pusht zuerst, das wird durchgeführt.

B pullt zuerst mal.

Git - Verteiltes Arbeiten

push und pull

Fall 3: A ändert Datei eins.dat in Zeile 1, B ändert auch Datei eins.dat in Zeile 1

A pusht zuerst, das wird durchgeführt.

B pullt zuerst mal.

```
CONFLICT (content):  Merge conflict in eins.dat
```

Nun eins.dat von Hand reparieren. Inhalt von eins.dat nun:

```
<<<<<<<<<<<<< HEAD
2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 erste Textdatei
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
======
3 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 erste Textdatei
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
>>>>>>>>>>>>>> 7df79e9d896778e8ccd587943cb68bd75
```

Git - Verteiltes Arbeiten

Versionen

Von Hand reparieren, dann

```
git add eins.dat; git commit; git push
```

Git - Verteiltes Arbeiten

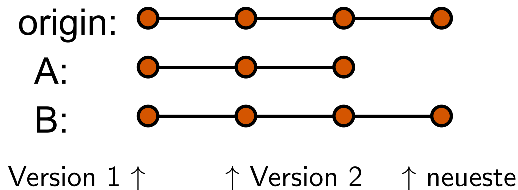
Versionen

Von Hand reparieren, dann

```
git add eins.dat; git commit; git push
```

Bislang: immer nur ein *branch*: keine parallelen Versionen, nur nacheinander. (Von links nach rechts: von alt nach neu).

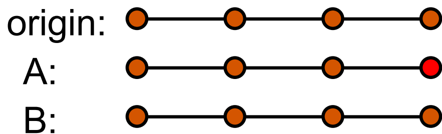
Ein möglicher Verlauf:



Git - Verteiltes Arbeiten

push und pull

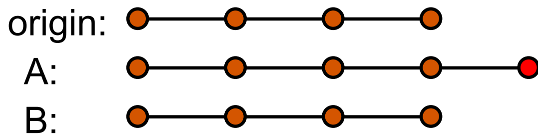
A pullt:



Git - Verteiltes Arbeiten

push und pull

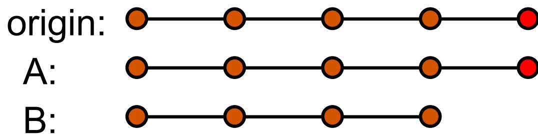
A commitet was neues:



Git - Verteiltes Arbeiten

push und pull

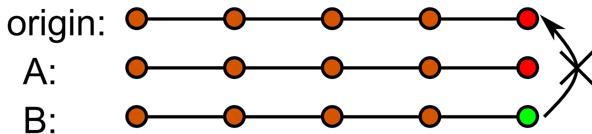
A pusht:



Git - Verteiltes Arbeiten

push und pull

B macht was neues und will pushen:

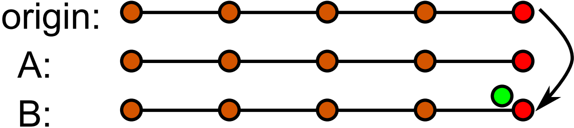


Missrät, da B nicht auf dem letzten Stand ist.

Git - Verteiltes Arbeiten

push und pull

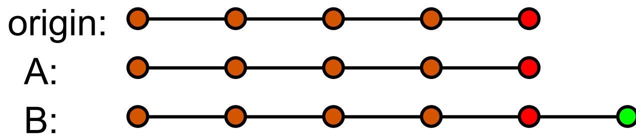
B pullt:



Git - Verteiltes Arbeiten

push und pull

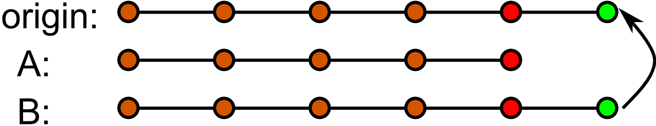
B merget (oder macht was neues):



Git - Verteiltes Arbeiten

push und pull

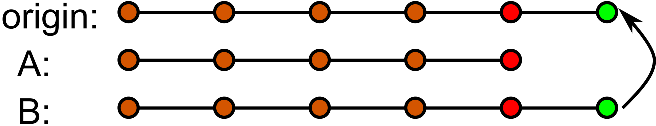
B pusht:



Git - Verteiltes Arbeiten

push und pull

B pusht:



usw.

Git - Verteiltes Arbeiten

push und pull

- git-clone from upstream to prime your local repository. ✓
- git-pull and git-fetch from origin to keep up-to-date with the upstream. ✓
- git-push to shared repository ✓
- git-format-patch to prepare e-mail submission
- git-send-email to send your e-mail submission
- git-request-pull to create a summary of changes for your upstream to pull

Recall: `fetch` ist `pull` ohne `merge`

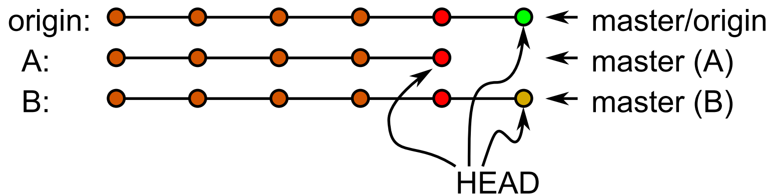
Fifty shades of....

Rückgängig machen

Git - Dinge rückgängig machen

HEAD und master

Bezeichnungen:



Die aktuellste Version ist HEAD.

Git - Dinge rückgängig machen

HEAD und master

Die früheren Versionen (*commits*) haben als Namen *hashes*
(das sind zufällig aussehende, eindeutig zuzuordnende Zeichenketten)

```
$ git log
commit 8d2f775a1d18bbdd02951e25cfa575b0f8aebf43
Author: Dirk Frettlöh <dfrettloeh@techfak.de>
Date:   Mon Jan 16 16:21:55 2021 +0100

    eins.dat repariert

commit 4c1f5e121a6f1c8b7af74013c983f4325aa69a25
Merge: 1a61068 df96eac
Author: Dirk Frettlöh <dfrettloeh@techfak.de>
Date:   Mon Jan 16 16:20:01 2021 +0100

    drei.dat ist nun korrekt

commit 1a6106815ec314f07253f5ba08fc3e7bc554f15e
Author: Dirk Frettlöh <dfrettloeh@techfak.de>
Date:   Mon Jan 16 16:18:30 2021 +0100

    drei.dat repariert
```

Git - Dinge rückgängig machen

commits und hashes

Oben ist also 8d2f775a1d18bbdd02951e25cfa575b0f8aebf43 dasselbe wie HEAD (letzter commit).

Wenn ich im `wd` eine Datei lösche:

Git - Dinge rückgängig machen

commits und hashes

Oben ist also 8d2f775a1d18bbdd02951e25cfa575b0f8aebf43 dasselbe wie HEAD (letzter commit).

Wenn ich im `wd` eine Datei lösche:

`git reset --hard` (setzt wd auf aktuelle Version im lokalen (!) repo)

ist dasselbe wie

```
git reset --hard HEAD
```

ist dasselbe wie

```
git reset --hard 8d2f775a1d18bbdd02951e25cfa575...
```

Git - Dinge rückgängig machen

commits und hashes

```
git reset --hard 8d2f775a1d18bbdd02951e25cfa575...
```

ist natürlich etwas zu lang. Es geht auch

```
git reset --hard 8d2f77
```

Regel: die ersten n Zeichen ($n \geq 4$), so dass es eindeutig ist.

Das alles setzt auf letzten commit im lokalen repo zurück.

Git - Dinge rückgängig machen

commits und hashes

```
git reset --hard 8d2f775a1d18bbdd02951e25cfa575...
```

ist natürlich etwas zu lang. Es geht auch

```
git reset --hard 8d2f77
```

Regel: die ersten n Zeichen ($n \geq 4$), so dass es eindeutig ist.

Das alles setzt auf letzten commit im lokalen repo zurück.

```
git reset --hard origin/master
```

Das setzt auf letzten commit im globalen repo zurück.

Git - Dinge rückgängig machen

Weitere undos

Es gibt noch viel mehr Möglichkeiten, Dinge rückgangig zu machen. Eine kleine Auswahl:

Situation: Ich habe gerade etwas gepusht und gemerkt, dass es Mist war. Der commit hash war `a1b2c3...`

```
$ git revert a1b2c3
$ git push
```

Git - Dinge rückgängig machen

Weitere undos

Situation: Ich habe gerade etwas commited, noch nicht gepusht, und gemerkt, dass mein Kommentar ("eins.dat geixft") Mist war.

```
$ git commit --amend -m ''eins.dat gefixt''
```

(ersetzt meinen letzten commit durch das aktuell gestagete - hier: nix gestaged, also: ersetzt nur den Kommentar)

Git - Dinge rückgängig machen

Weitere undos

Situation: Ich habe gerade etwas commited, noch nicht gepusht, und gemerkt, dass mein Kommentar ("eins.dat geixft") Mist war.

```
$ git commit --amend -m ''eins.dat gefixt''
```

(ersetzt meinen letzten commit durch das aktuell gestagete - hier: nix gestaged, also: ersetzt nur den Kommentar)

Situation: Ich habe gerade versehentlich eins.dat aus meinem wd gelöscht. Mist.

```
$ git checkout -- eins.dat
```

(Siehe oben: holt eins.dat aus dem lokalen git)

Git - Dinge rückgängig machen

Weitere undos

Situation: Meine letzten drei commits waren alle Mist. Ich möchte zurück auf den Zustand vorher.

`$ git reset` *hash*

(setzt zurück auf commit *hash*. Dateien im wd sind immer noch Mist.)

Git - Dinge rückgängig machen

Weitere undos

Situation: Meine letzten drei commits waren alle Mist. Ich möchte zurück auf den Zustand vorher.

`$ git reset` *hash*

(setzt zurück auf commit *hash*. Dateien im wd sind immer noch Mist.)

Oder: `$ git reset --hard` *hash*

(setzt zurück auf commit *hash*. Dateien im wd sind auch wieder wie damals.)

Git - Dinge rückgängig machen

Weitere undos

Situation: (siehe oben) Ich habe eine Datei `mist.txt` commitet, die ich gar nicht im repo haben möchte.

```
git rm mist.txt
git commit -m "remove mist.txt"
```

(git rm nimmt sie aus dem repo und aus dem wd raus, aber sie ist noch gestaget.)

Git - Dinge rückgängig machen

Weitere undos

Situation: (siehe oben) Ich habe eine Datei `mist.txt` commitet, die ich gar nicht im repo haben möchte.

```
git rm mist.txt
git commit -m "remove mist.txt"
```

(git rm nimmt sie aus dem repo und aus dem wd raus, aber sie ist noch gestaget.)

Situation: Ich habe eine Datei `mist.txt` gestaged, aber noch nicht commitet, die ich gar nicht im repo haben möchte.

```
git rm --cached mist.txt
```

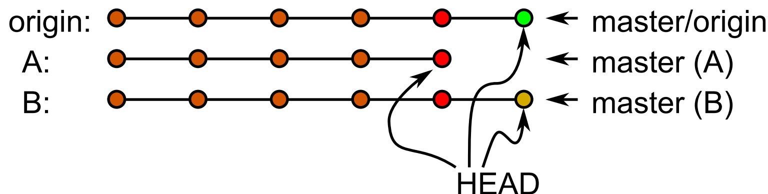
(git rm –cached nimmt sie nur aus dem Stage-Bereich ("Index") raus)

Branches
(Verzweigungen)

Git - Branches

HEAD und branch

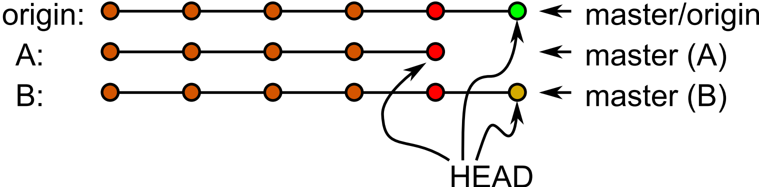
Bisher keine Verzweigungen (*branches*). Bezeichnungen:



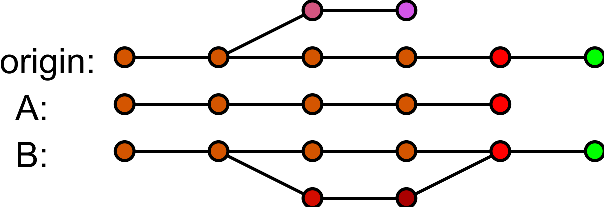
Git - Branches

HEAD und branch

Bisher keine Verzweigungen (*branches*). Bezeichnungen:



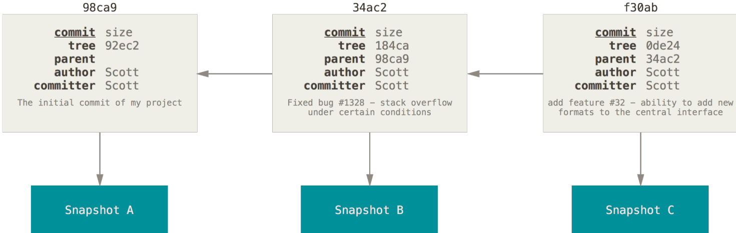
Wir wollen nun auch Versionen verzweigen können:



Git - Branches

HEAD und branch

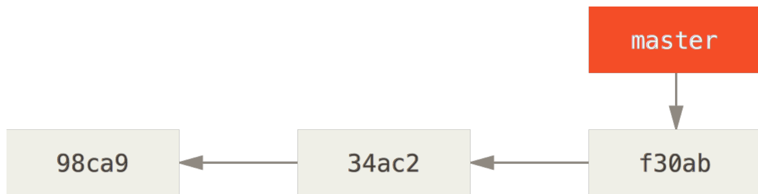
Genaueres Bild des lokalen master *branchs*:



Git - Branches

HEAD und branch

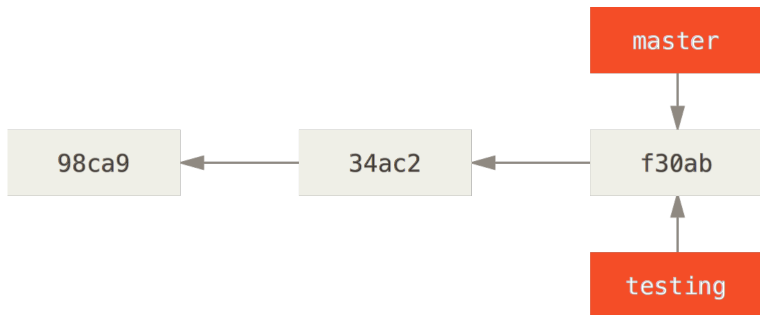
Gröberes Bild des lokalen master *branch*s:



Git - Branches

HEAD und branch

Neuen *branch* erzeugen: `$ git branch testing`

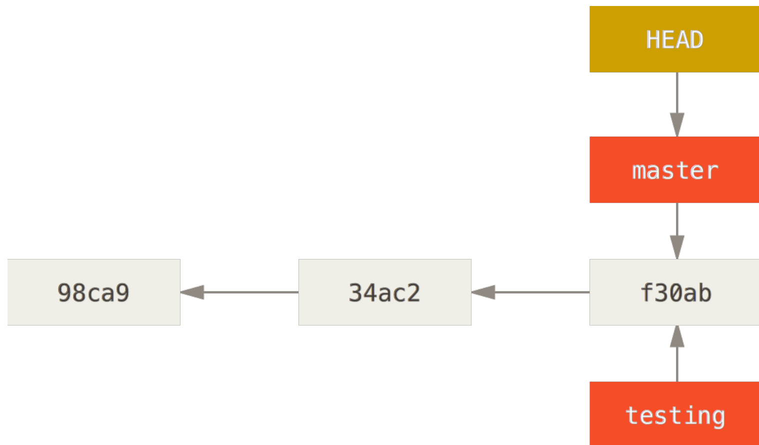


Es wird also keine Kopie aller Snapshots angelegt, sondern nur ein neuer Zeiger erzeugt.

Git - Branches

HEAD und branch

Wohin zeigt HEAD?

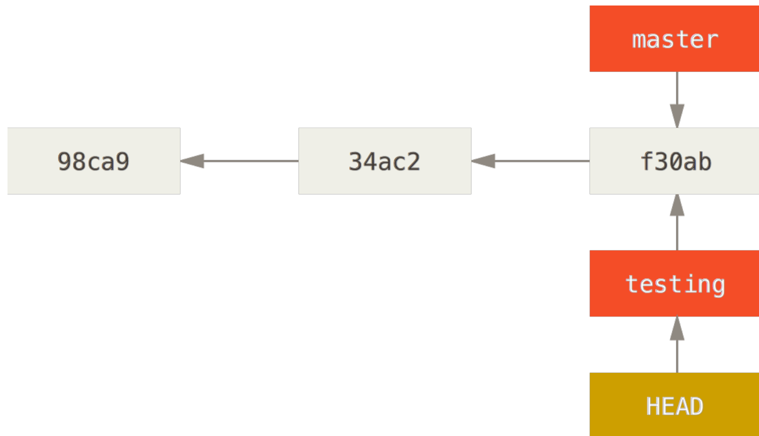


Wir “befinden” uns noch im *branch* master.

Git - Branches

HEAD und branch

```
$ git checkout testing
```



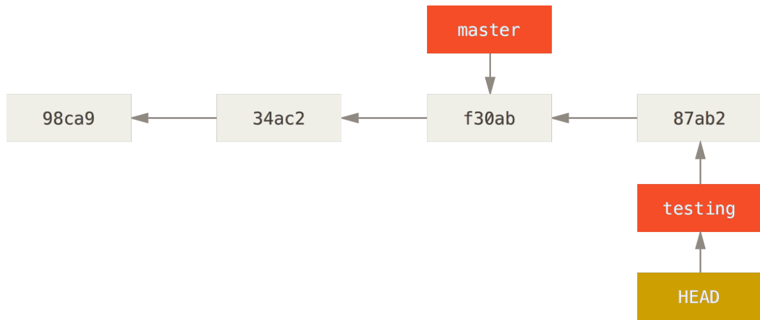
Jetzt “befinden” wir uns im *branch* testing.

Git - Branches

HEAD und branch

```
$ emacs test.rb
$ git commit -a -m 'test.rb geändert'
$ git push --set-upstream origin testing
```

(dem remote-repo mitteilen: wir arbeiten aktuell in testing)

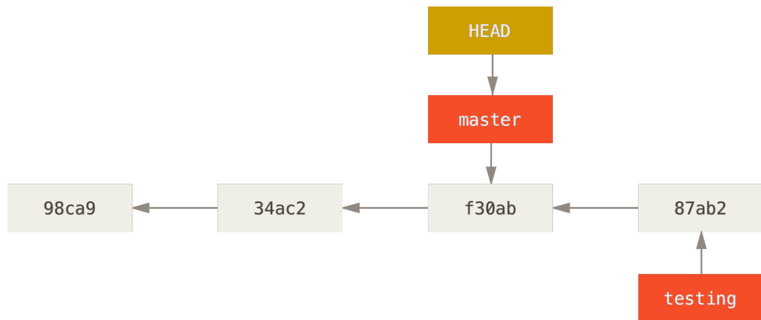


... und arbeiten im *branch* testing.

Git - Branches

HEAD und branch

```
$ git checkout master
```

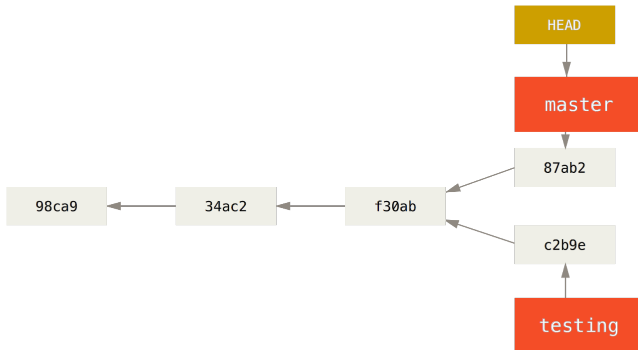


Wir wechseln zurück zu master.

Git - Branches

HEAD und branch

```
$ emacs test.rb
$ git commit -a -m 'test.rb verbessert'
```



... und arbeiten nun im *branch* master. Usw.

Git - Branches

HEAD und branch

Eine primitive Anzeige bietet git log:

```
$ git log --oneline --decorate --graph --all
```

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) made other changes
| * 87ab2 (testing) made a change
|/
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #1328 - stack overflow under certain cond
* 98ca9 initial commit of my project
```

Git - Branches

merge branches

Verschiedene *branch*es können gemerget werden:

`$ git checkout master` (in *branch* master wechseln)
`$ git merge testing` (testing in master einfügen)

Jetzt stimmen master und testing überein, also kann testing auch wieder gelöscht werden:

```
$ git branch -d testing
```

Git - Branches

merge branches

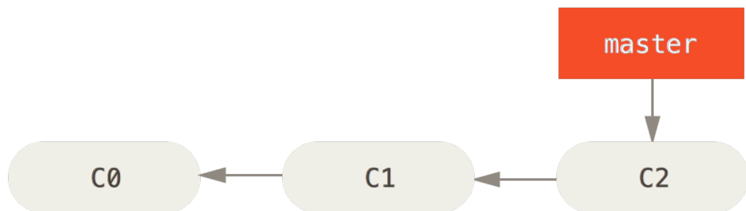
Ein Beispielszenario (vgl. Pro Git, Kapitel 3):

- A arbeitet an einer Webseite.
- Erzeugt dazu einen neuen *branch* `iss53` und arbeitet darin
- Es kommt ein Anruf: dringend etwas an der eigentlichen Webseite reparieren
- A wechselt zurück zu `master` und erzeugt da einen neuen *branch* `hotfix`
- Repariert in `hotfix` den Fehler, testet, merget `hotfix` in `master`
- Arbeitet weiter in `iss53`

Git - Branches

merge branches

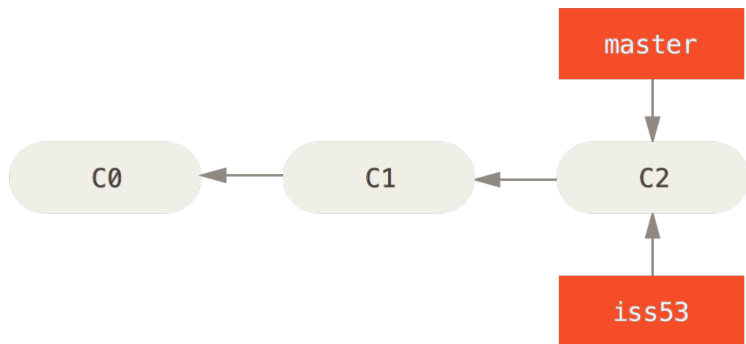
Zustand vorher:



Git - Branches

merge branches

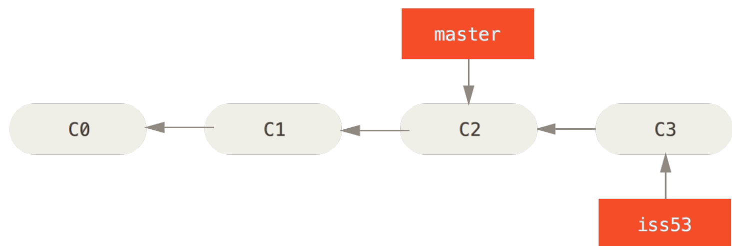
`$ git checkout -b iss53` (kurz für `git branch` und `git checkout`)



Git - Branches

merge branches

```
$ emacs index.html
$ git commit -a -m 'added [issue 53]'
```



Git - Branches

merge branches

A arbeitet in iss53. Nun kommt der Anruf.

`$ git checkout master` (in master wechseln)

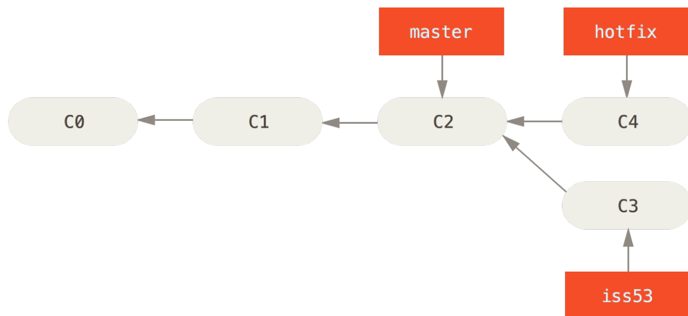
Beachte: Das wd sieht nun so aus wie master!

A erzeugt einen neuen *branch*, behebt dort den Fehler:

Git - Branches

merge branches

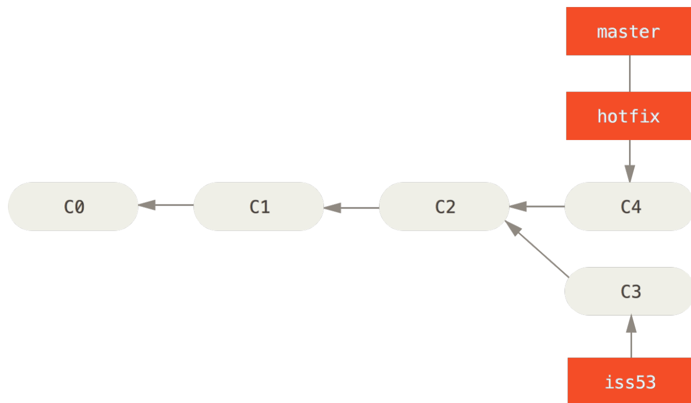
```
$ git checkout -b hotfix
$ emacs index.html
$ git commit -a -m 'index.html fixed'
```



Git - Branches

merge branches

```
$ git checkout master
$ git merge hotfix
```
(hotfix in master einfügen)



(Kein Konflikt, “fast-forward” nach hotfix)

Git - Branches

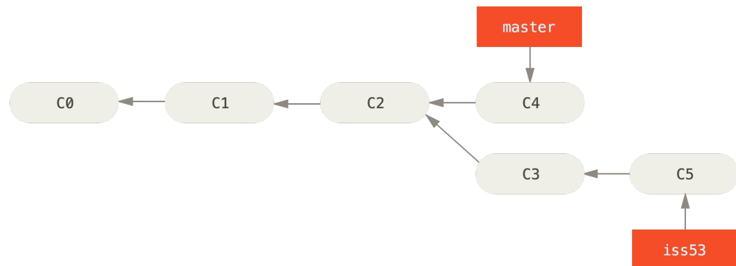
merge branches

```
$ git branch -d hotfix
```
(hotfix löschen)

```
$ git checkout iss53
```
(nach iss53 wechseln)

```
$ emacs index.html
```
(weiterarbeiten)

```
$ git commit -a -m 'finished [issue 53]'
```

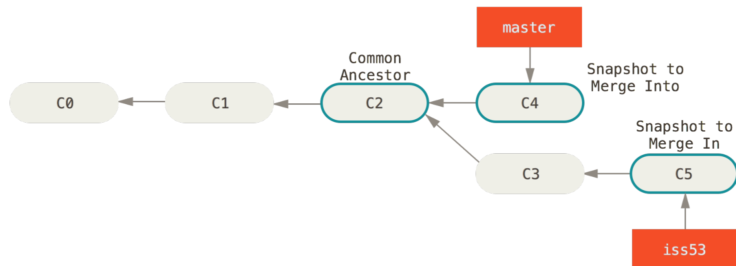


Git - Branches

merge branches

`$ git checkout master` (nach master wechseln)

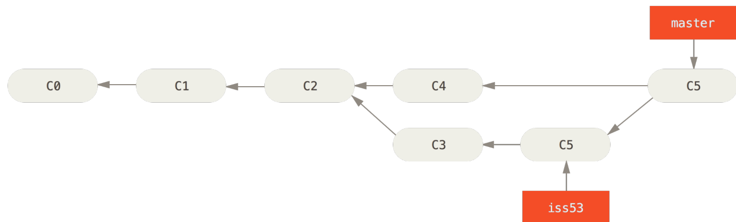
`$ git merge iss53` (iss53 in master einfügen)



Git - Branches

merge branches

master und iss53 sind zwei verschiedene Zweige, “fast-forward” geht hier nicht.
Also mergen, Konflikte lösen, siehe oben.



Git II - Überblick

`git clone`	ein repo auf den eigenen Rechner holen
`git pull, fetch, push`	Änderungen holen / sichern
`git rm, revert, reset`	Änderungen rückgangig machen

Branches:

`HEAD`	Zeiger auf aktuellen *branch*
`git branch`	neuen *branch* erzeugen
`git checkout`	HEAD auf anderen *branch* setzen
`git checkout -b`	kurz für `branch` und `checkout`
`git merge`	*branch*es zusammenfügen

Ende der heutigen Vorlesung

Sie probieren das alles gleich in den Tutorien aus.

Viel Spaß!