GIT

Beginner

Poll

Who here…

- uses Linux, Windows, OS X?
- doesn’t use a shell regularly?
- knows CVS, SVN, Mercurial, Bazaar, ...?
- has not installed Git yet?
- has no github account?
- does not have his public key on github yet?

Installation

http://git-scm.com

Questions

- Ask questions right away!

Why should I use Git?

.bak

.new, .new_new, .new_jetzt_aber_wirklich,
...

Undo

Distributed

Basic configuration

```
git config --global user.name "John Doe"
git config --global user.email mail@host.com
git config --global push.default simple
```

Username und Email wird an jeden commit in git angeheftet.

Anhand der Email wird auch zum Beispiel github die commits zuordnen

Initializing the repository

```
git init test-project
cd test-project

git status
```

git init funktioniert auch mit existierenden Verzeichnissen problemlos

git status ist einer der wichtigsten Befehle: jeder 3. bis 6. git Befehl auf der Konsole ist im Allgemeinen ein git status.

Im folgenden werden wir nach fast jedem Befehl git status benutzen, um zu sehen, wie sich das repository durch unsere Befehle verändert

Adding a file

```
touch README
git add README
git commit -m 'my first commit'

git log
```

touch erzeugt eine leere Datei
git add fügt sie zum index hinzu und staged sie (vorbereitung zur verpackung)
git commit verpackt sie in einen commit

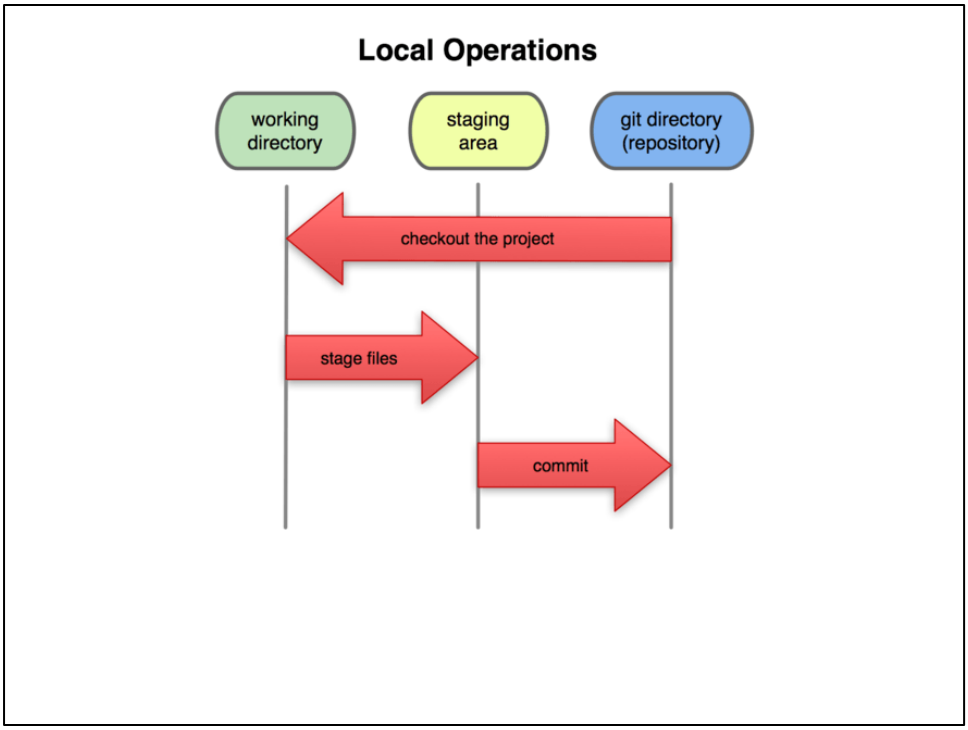
Making changes

```
echo 'This is a test repo.' > README

git diff
git add README
git commit -m 'first real diff'
git log -p
```

jetzt können wir den ersten diff begutachten.
in grün mit einem + sieht man die hinzugefügten zeilen
gelöschte Zeilen sind rot mit einem - am anfang.

git log -p zeigt nicht nur die commit messages an, sondern auch den diff zu jedem commit.
Es gibt zu jedem git befehl sehr viele optionen, mit denen man die ausgabe ändern kann, oder den bereich auf dem das kommando ausgeführt wird ändern kann.
um etwas darüber herauszufinden kann man z.B. git help log eintippen.



Man kann dinge in die staging area bekommen, indem man sie mit git add hinzufügt
entfernen geht mit git reset

die staging area ist dazu da, um zusammenhängende kleine commits machen zu können
die staging are ist quasi wie ein offenes paket, in das man sachen legen kann
mit git commit ‘schnürt’ man dann das paket zusammen und klebt ein etikett darauf.

Creating a branch

```
git branch feature/readme
git checkout feature/readme
echo 'Please only use at workshop' >> README

git commit -am 'made another change'
git log -p
```

man kann das git branch und git checkout auch zu einem Befehl kombinieren:
git checkout -b new-branch

>> fügt ans ende der Datei an

git commit -a ist wieder eine abkürzung, dabei wird git add auf alle modifizierten und gelöschten Dateien angewendet. Neue dateien werden dabei NICHT hinzugefügt!

Merging a branch

```
git diff master
git checkout master
git merge feature/readme
```

statt master kann man bei git diff alles mögliche benutzen was einen commit identifiziert (commit-sha, HEAD, HEAD~2, HEAD^2, branch-name, …)

merge mergt den angegebenen branch in den aktuellen. Das sollte man nur tun wenn der aktuelle branch keine uncommitteten änderungen hat

Forking

Press the “fork” button on github

```
copy ssh clone url
git clone <url>
```

change, commit, push
click the green arrows button
create a pull-request

Optionale Folie, wenn noch Zeit ist.

Tagging a commit

```
git tag <version/name>
git tag
git log --decorate
git push --tags
```

- Never delete a remote tag to change it!
  - A tag is only ever downloaded once. If it changes remote, it does not change locally

GIT

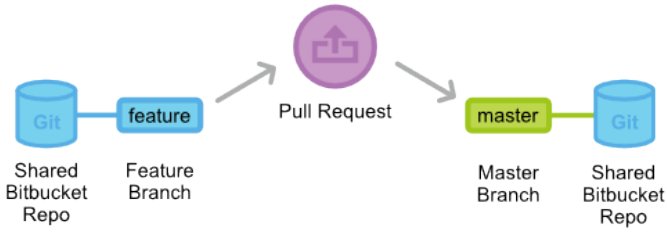
Advanced

Creating a Git server

```
git init --bare
```

Git Workflow

- Based on features:



Am Flipchart aufzeichnen:

Master - Features - Pull-Request - [Release-Branch - Hotfix] Release-branch und hotfix sind nur wichtig wenn man nicht continuous deployment macht

Git Workflow

- Create feature-branches:
  - `feature/…, bugfix/…,`
- Use tags to mark *stable* releases
- There are other workflows

Extending Git

Git recognizes commands like “`git-mycommand`”.

```
/usr/bin/git-feature:
   #!/bin/sh
   git checkout -b feature/$1
   git push -u origin feature/$1

git feature new_cool_feature
```

Git-Devbliss

```
github.com/devbliss/git-devbliss

sudo pip3.4 install git-devbliss
```

Demo!

git devbliss feature test-feature
editieren irgendwas sinnvolles (oder auch nicht :p)
git devbliss finish

pull-request in github öffnen

git devbliss status

Troubleshooting

What if something went wrong?

Undoing commits

- `git reset --hard origin/master` # undo a merge
- `git reset HEAD~1`
  - last commit is “undone”
  - content is “not staged for commit”
- `git reset --soft HEAD~1`
  - last commit is “undone”
  - content is staged.
- `git reset --hard HEAD~1`
  - force your HEAD to HEAD~1
  - the old HEAD is still out there, but without reference

Find out who wrote that line

- `git blame Vagrantfile`
  - what is the oldest revision that already had that line
- `git blame --reverse <start>..<end>`
  - git blame --reverse HEAD~20..HEAD
  - what is the highest revision that the lines from <start> are still in?
- `git blame -M`
  - detect moved lines
- `git blame -C[ -C[ -C]]`
  - detect lines copied from other files in same commit
  - “ in same commit and in creating commit
  - “ in all commits

Binary search the first bad commit

- binary search all commits for the first bad
- if HEAD is defunct and tag 1.0.0 is good:
  - `git bisect start`
  - `git bisect bad` # no hash /ref => HEAD
  - `git bisect good 1.0.0`
  - git will checkout a commit between good and bad
  - run tests
  - `git bisect good` / `git bisect bad`
  - repeat
- `git bisect reset` # to stop bisect mode
- `git bisect run` # auto bisect!

falls neu:
git init test-repo
cd test-repo
touch README
git add README
git commit -m 'most important file'

git tag last_working_tag
for i in {1..100}; do git commit --allow-empty -am 'some unimportant commit '$i; done
git rm README && git commit -am 'removed README'
for i in {101..400}; do git commit --allow-empty -am 'some unimportant commit '$i; done
git bisect start HEAD last_working_tag

ls
git bisect bad/good

git bisect run test -f README
git bisect reset

nächste Folie!

git revert <bad-sha>
test -f file1

Revert a commit

- `git revert <hash>`
- creates a commit with negative diff to cancel out the reverted commit
- the old commit stays!
  - important for reverting a merge

Secret / Binary in Repo

Why is this bad?

- Binary:
  - Every time a binary changes, copy in history
  - history has all versions of all binaries
  - git get's real slow real fast (jenkins, local)

- Secret & Binary:
  - Deleting is useless (still in history!)

Secret / Binary in Repo - Fix it!

- `git filter-branch --prune-empty --index-filter 'git rm -rf --cached --ignore-unmatch <file.name>' --tag-name-filter cat -- --all`
  - `-- --all`: work this command on all revisions
  - `--index-filter`: run filter on every commit's index
    - `git rm -rf --cached --ignore-unmatch <filename>`: delete the file in the index, no error if not found
  - `--prune-empty`: delete commits that are now empty
  - `--tag-name-filter`: run filter on every tag
    - cat: use same name => update tag - DANGER!

Secret / Binary in Repo - Gotchas

- Tell everyone to merge all branches
- No new branches anymore!
- Pull master
- `git filter-branch --prune-empty --index-filter 'git rm -rf --cached --ignore-unmatch <file.name>' --tag-name-filter cat -- --all`
- `git push --force` (!)
- `git reset --hard origin/master` for everyone

Secret / Binary in Repo - Avoid it!

- if anybody forgets to reset --hard and instead does a git pull:
  - all old comits are still there
  - all the non-binary commits are duplicated

=> DO NOT PUSH BINARIES TO REPO!

Accidental Merge - Revert Merge

- on master
  - `git log --graph`
  - `*` commit <merge-sha>

    |\ Merge: cadc526 835716e
  - Check which of the commits is the good one (should be the left). left is 1, right is 2.
  - `git revert -m 1 <merge-sha> && git push`

Accidental Merge - Revert Merge

- on feature-branch:
  - `git merge master`
  - `git revert HEAD && git push` # revert the revert
  - Do not forget to revert the revert! (else feature defunct)

Only possible if there is a merge commit!
github does '`git merge --no-ff`' for pull-requests

Accidental Fastforward Merge

If there is no merge commit:

- reset master:
  - `git reset --hard <last-non-bad> && git push --force`
  - `git cherry-pick <all-other-good>`
  - `git fetch && git reset --hard origin/master` # all others
- revert and revert-revert
  - `git revert <all-bad>` # on master
  - `git revert <all-revert-commits>` # on feature-branch

Manage submodules

git submodule - Setup

- Very easy to add a submodule:
  - `git submodule add git@github.com:user/repo`
  - `git commit -am ‘added submodule’`
  - `git push`

- But what did actually happen?

git submodule - where is the URL?

- submodule URL is under git version control:
  - <project_root>/.gitmodules
    [submodule "<submodule_path>"]
    url = <submodule_url>
- There is a local copy in .git/config that is used for the actual commands!

git submodule - where is the hash?

- submodule commits are saved directly within each commit:
  - `git ls-tree HEAD` # for submodules in project root
  - `git ls-tree HEAD: <folder_with_submodules>`
- Every commit can be checked out at any time later, and the submodules are compatible (as they were when that commit was pinned)

git submodule - get it step by step

- `git pull`:
  - get .gitmodules (under version control!)
  - get HEAD (which includes submodule commits)
- `git submodule init`
  - copy entries from .gitmodules to .git/config
  - OR: .git/modules/<path-to-submodule>/config
- `git submodule update`
  - clone/update submodules to pinned commit

git submodule - get it faster

- `git submodule update --init`
  - combines init and update => no local modification
- `git submodule update --init --recursive`
  - as above, but does this recursively for all submodules
- `git clone --recursive`
  - Only on first clone, but will clone all submodules along

git submodule - status

- `git submodule [status [--recursive]]`
  - hash for every submodule
  - + => hash does not match with upstream
  - - => not initialized
- `git submodule summary`
  - commit headlines for changes between current and remote revision
  - only for direct submodules, no recusive!

git submodule - foreach

- `git submodule foreach [--recursive] <cmd>`
  - runs command on all submodules
  - stops on first exitcode > 0

Appliances:

- `git submodule summary` # for all submodules
- `git pull origin master` # update all submodules
  - This is only ok for bleeding edge development!

git submodule - change

- change pinned version
  - ‘`git add`’ the folder, commit and push.
  - all others have to pull and ‘`git submodule update`’
  - `git commit -am` can mess things up!
- change upstream URL
  - edit .gitmodules
  - `git submodule sync && git submodule update`
  - tell everybody to pull + sync + update.
- sync vs init:
  - init only creates an entry (will not update)
  - sync only updates an entry (wil not create)

git submodule - remove

- `git submodule delete`? - No!
  - `git submodule deinit` => deletes entry in local .git/config (or do it yourself)
  - `git rm <path-to-submodule>` => deletes folder
  - Entry in .gitmodules has to be deleted manually