Table of Contents

GIT Training

(1-2 day - training)
Jochen Metzger

Projects with git

- Linux kernel project
- github
- ruby on rails
- phpmyadmin

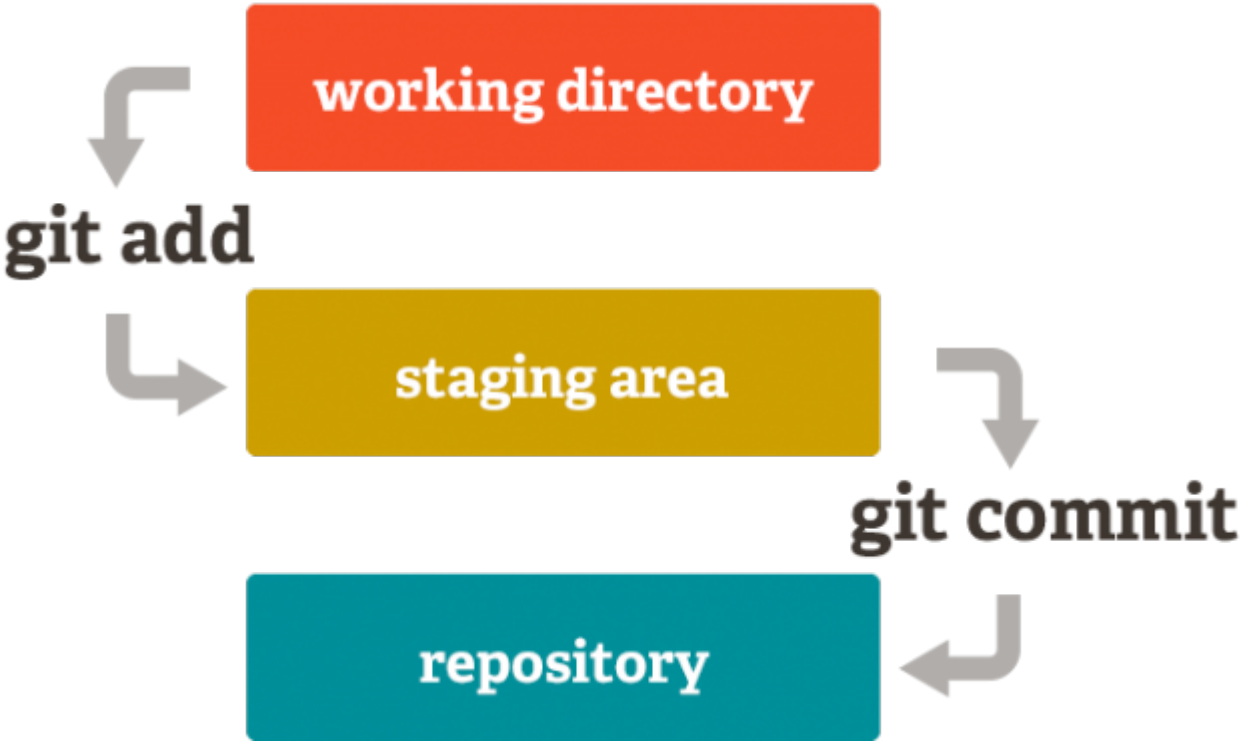
Git History

- First Release 2005
- Created By Linus Torvalds
- git means: blödmann / in engl. colloquial language

Git - how does it work ?

- git takes snapshots
- git works offline
  (git saves the complete project including all versions)
- all files are saved in objects.

Git - the flow



The starting point : the git-repository

- “git init” initializes a repository
- after that: All the intelligence and logic is within the subdirectory .git
- this means: folders/files are fully functional, also if the .git - folder is deleted

Lab 1: Create a local repository

```
# Create a folder "training" on the Desktop
# or your "home" - Directory

# On Windows:
Click the right button -> Use -> Git Bash here
# On Linux
Change to the newly created folder: cd training

# Now create the repository
git init

# What folders do see ? (the same on Windows and Linux)
ls -la
```

The helper : git status

- Show the status of files and folders in working directory

Lab 2: Files and their status

```
# create an empty file todo.txt in your directory "training"

# The command "touch" creates empty files
# Both in Windows and Linux
touch todo.txt

# Explore the changes in your directory with status
# What do you see ?
git status
```

The staging area / index :

- Here you will decide, what gets shipped (for the next “git commit”)
- For doing so, we will use: git add

The details : git add

- add all directories and files: git add . (**there is a dot after add ;o)**)
- –
- add a specific directory: git [directory]
- example: git my_directory

- –

Lab 3: Adding files to the staging area

```
# add 1 line of content to "todo.txt" with your favourite editor
# e.g. windows -> notepad
# e.g. linux -> nano

# Now add the file to the staging area
git add todo.txt

# How does the status look like now ?
git status
```

Git and its objects

- To manage its data, git uses objects
- e.g. when you add a file with git add filename
  a new object is created
- There are 4 types of objects
  - blobs
  - trees
  - commits
  - tags

SHA1 - checksums & backgrounds

- git extensively works with checksums in the background
- a checksum is a 40-char long hex-string (a unique checksum of the data)
- every object gets a checksum
  - blobs
  - trees
  - commits
  - tags

Lab 4: Find created object

```
cd .git
cd objects
ls -la
# you will find a directory name with the
# first 2 chars of the object, e.g.
# drwxr-xr-x   3 jmetzger  staff  102 26 Okt 16:28 4f

# change into that directory
# Hint: Replace name with your directory name
cd 4f
# your object
ls -la
```

```
# 51d02eb27b6bdf1741ad48ccf6f7dc3326bbd2

# now let us see the content
# (taking the 4 letters of the object is sufficient)
git cat-file -p 4f51
my first line

# and the type of object
git cat-file -t 4f51
blob
```

Git - your identity (Why ?)

- Working in a team: Who has done what ?
- Makes it easier to organize work.
- You can easier search work based on Author (=Identity)

```
 On newer versions your forced
to set your identity before you
you can publish your work to a remote server (=push)
```

Lab 5: Set up your identity Git

```
# within your project (folder training)
git config --global user.name "Jochen Metzger"
git config --global user.email "j.metzger@t3company.de"
# Checking your config:
git config --list
# Checking your config property:
git config user.email
```

The journey: there we go to the (local) repository

- the next step: get the files/directory into the repo
- with “git commit” the work is transferred (after git add or git add .)

Git commit - in detail

- git commit (commit all files from staging)
- git commit -a (in addition all files that have been deleted or known (and have changed))
- git commit / git commit -a opens an editor where i can enter the commit message
- git commit -m “my commit-message” - makes committing a oneliner

Git log

- All commits are logged in to the log
- You can access the log with
    - git log

Git log - what in detail ?

- Output the log:
  git log
- –
- a short version of logs:
  git log –oneline
- –
- all from a specific author:
  git log –author=Max

Lab 6: Commit the changes and watch the log

```
# look into the status
git status

git commit
# editor opens - now add a commit message
# & save & close the editor

# everything shoud be clean now
git status

# you will notice the same commit-id as
# in the commit message
git log
```

Git aliases

- With aliases you can create your own git commands ;o)
- Creation Syntax:
  - git config –global alias.name_of_alias “<git-command> <git-params>”
  - e.g. git config –global alias.cc “commit -a”
- Usage Syntax:
  - git cc
- You can still add params to the command on usage:
  - git cc -m “my commit”

Git log - beautified

- Why ?
  - For later usage, it will be easier to have a beautified log

- ```
  git config --global alias.lg "log --color --graph --
  pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr)
  %C(bold blue)<%an>%Creset' --abbrev-commit"
  ```
  ```

Lab 7: Setup beautified log

```
git config --global alias.lg "log --color --graph --
pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold
blue)<%an>%Creset' --abbrev-commit"

# use beautified log
git lg
```

Branches -> why ?

- important concept of git
- work on features easily
- independent from remote repository

Create branches -> 2-step-version

- git branch feature-4711
- git checkout feature-4711

Create branches -> 1-step-version

- git checkout -b feature-4711

Branch - which one is active ?

- git branch (star means active)

change to another branch

- git checkout feature-4711

Lab 8: Create a new branch + work there

```
git checkout -b feature-4711
# check which branch is active
git branch

# create a file feature-4711.txt
touch feature-4711.txt
# create a line of text into it
# with your favourite editor
# .e.g
# Line of code in 4711

# now commit the changes
git status
git add .
git commit -am "New feature-4711.txt"
```

```
git status
# do you notice, that the branch is ahead ?
git lg
```

Merge changes -> merge

- You can merge, if you want to get changes from another branch
- A typical scenario would be:
    - You are in master
    - Checkout a feature branch
    - Work on a feature
    - Checkout master
    - merge changes from feature-branch
- You merge feature from another branch into your current branch
    - with –>
    - git merge your-feature-branch

Delete branch

- You should cleanup unused branches as frequent as possible
- git branch -d feature-4711 (you should not be within the branch)

Lab 9: Merge branch (fast-forward) - from other feature

```
# we did this before
# and worked on branch
git checkout feature-4711
git checkout master
git merge feature-4711
# you will notice, that both branches are at the same commit-id
git lg
git branch -d feature-4711
```

Merge - FastForward - How come ?

- Fast-Forward just move the pointer forward
- HEAD always points to tip of the checked out branch
- HEAD is simply the entry in a file
    - Content is itsef a reference
    - Reference holds Commit-Id.

Lab 10: Exploring HEAD

```
cd .git
cat HEAD
# ref: refs/heads/master
cat refs/heads/master
# f80891db4afa604b243bd06a5779fee88c3cad53
```

```
# compare this with the last commit - id
# What do you notice ?
git lg -1
```

Replay changes in other branch + changes on current branch -> rebase

- Change into branch (z.B.feature-4711)
- git rebase master
- – how does it work ? →
    1. go back to the point where the branch was created
    2. apply change from master
    3. at the end apply changes from feature-4711
- – ATTENTION →
    1. only do rebasing in your own “local” branch (NO ! collaborative branches)
    2. never rebase, after push is done and branch is shared with others !

Back in time -> reset

- e.g. git reset –hard HEAD~1
- attention: only use it, when changes are not published (remotely) yet.
- → It is your command, IN CASE your are telling yourself, omg, what's that, what did i do here, let me undo that

Cancellation -> revert

- Take back the last change
  But: you will have an additional log entry
- `git revert`

From -> local repository -> to -> remote repository -> why ?

why ?

- data is saved outside of my system
- others can access it too (collaboration, e.g. on a feature)

Remote repository - examples

- gitlab
- github
- gitosis
- –
- under the hood: git.

Login to gitlab

- Introduction of the gui

(Windows -> git bash) create private/public key pair

1. desktop → right click → git bash
2. ssh-keygen -t rsa # set password !
3. cd ../.ssh
4. cat id_rsa.pub # that's the public key
5. # Mark key with mouse → then → CTRL + C
6. open gitlab in browser
7. (gitlab) menü → profile settings → SSH keys → paste key and click button “Add Key”
8. Test the connection with ssh git@git.server.com

Create a repo under gitlab

1. login
2. create repo (schulung)
3. introduce repo locally

Clone + Change

- Clone the empty repo
- Create a folder “code”
- Within that folder create a file “main.c” with the following content

```
#include<stdio.h>

int main() {
    printf("Hello World\n");
    return 0;
}
```

Build

- Create a “build” - folder
- Let's assume you build the software and now have the following file in the build folder

```
# just touch them
# touch main.o
# touch main.exe
```

Adding the build folder but not the files

- Add .empty in build folder
- Create .gitignore in top folder with content (above build)

```
build/*.o
build/*.exe
```

- Add build folder and main.c

  ```
  git add build
  ```

```
git add main.c
git commit -am .
```

Push changes online (to repo)

- ```
  git push
  ```

Publish the local changes remote

- git push origin master

Tagging of a current version (locally)

- git tag -a v0.0.1 -m “Commit Message”
- # Pushing the tags to the server
- git push –tags

- ```
  # Deleting pushed tags on the remote server
  # git push --delete <tagname>
  # Example
  git push --delete v0.0.1
  ```

Tags: Delete remotely deleted tags locally

- ```
  # in ~/.gitconfig
  [alias]
      ... ... ... # your existing aliases
      pt = !git tag -l | xargs git tag -d && git fetch -t
  ```

Clone an online session into a new (none existant) directory

- git bash (on the desktop - right click)
- git clone https://url schulung2

where is configuration saved ?

- on your local system the general configuration is saved in a file.
- Linux/Mac: ~/.gitconfig
- –
- example Mac: /Users/jmetzger/.gitconfig
- example Linux: /home/jmetzger/.gitconfig
- example Windows: C:\Users\<user_name>\.gitconfig
  ```

Commit - messages: what for and why should they be speaking ?

- other participants should be able to see changes based on commits
- good commit - message: reduces the time for search (for other participants) - more efficient
- search more easily and thoroughly (e.g. for later debugging)
- eventually included in the changelog and other tools

Commit - message : structure

- line 1: short! summary only(!) chars and letters
- line 1: max. 50 chars
- line 1: best practice → issue number/logical unit, then ":", then summary
  e.g.
  modulexy: Fixed problems with memory management
- line 2: EMPTY
- line 3: thorough description of commit
- line 3: max. 72 chars
- line 3: to structure: *, -, # possible
- line 3: time: presence
- line 3: do not write, what do you did, but why.

The cleanup: removal and untracking : git rm

- git rm
  –
  removes the files locally (working directory) as well as for next staging. In the next commit the file will not be contained anymore.
- git rm –cached
  –
  (if i only want to "untrack" a file (so it should be in the repo anymore), but want to keep it locally)

Branches (Tips & Tricks)

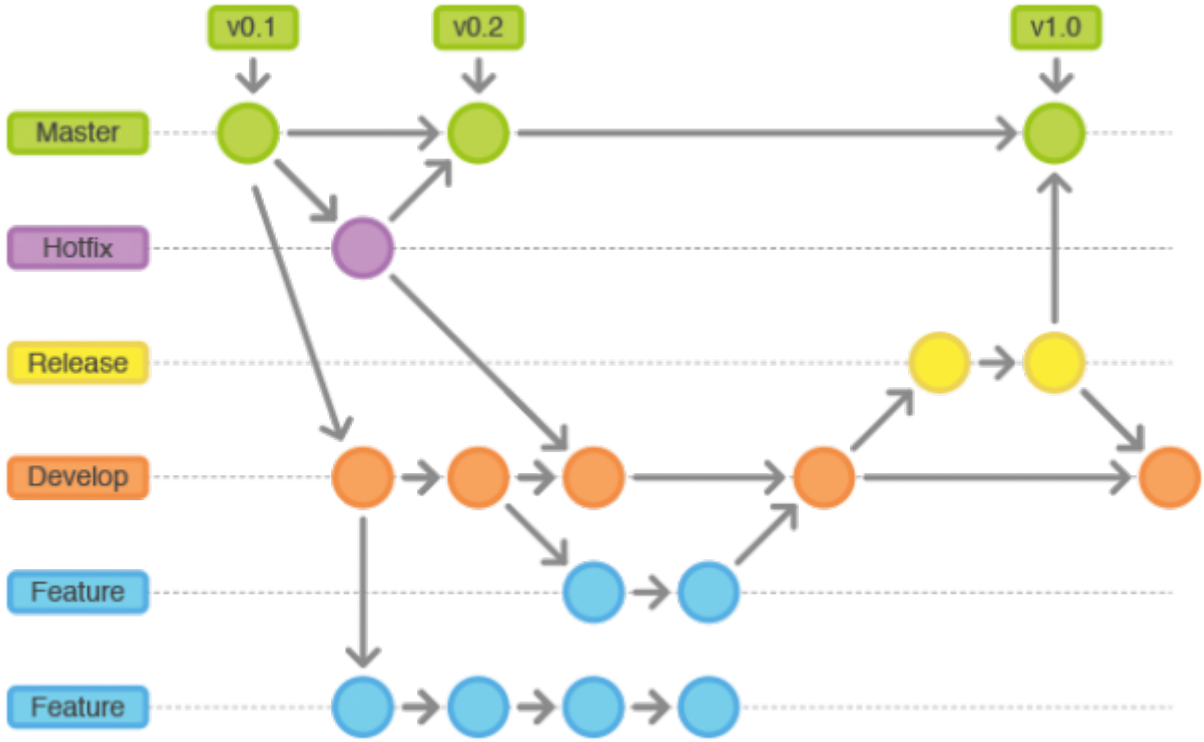
```
# Delete branch online
git push --delete origin feature-5
#
```

Troubleshooting ssh -> repo (tortoise/openssh)

It is either possible to use openssh or plink.exe. Here are the most important settings for ssh.

- git bash: echo $GIT_SSH
  it should be: C:\Program Files\Git\usr\bin\ssh.exe here
- if not: export GIT_SSH="C:\Program Files\Git\usr\bin\ssh.exe"
- in addition in tortoisegit under settings → network the should be the following:
  C:\Program Files\Git\usr\bin\ssh.exe

Workflows -> gitflow workflow



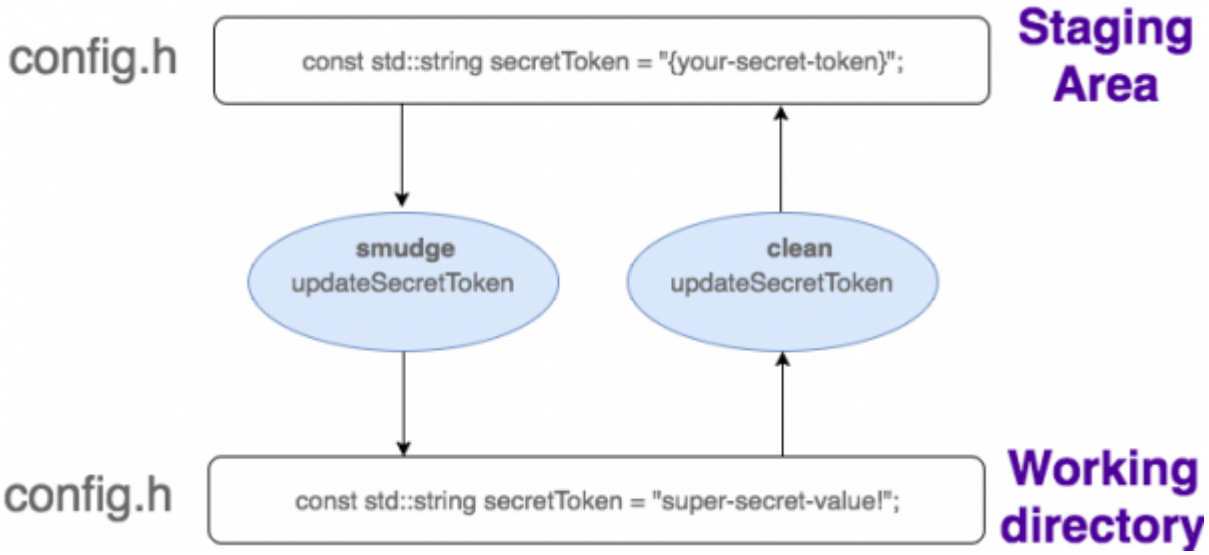
Forking (Basics)

- To fork is just another way of saying clone, PLUS
  - (bitbucket) manages the relationship between the original repository and the fork for you
  - Forking usefull when,
    - You want to do some major development work
    - you may or you may not later merge back the repository

Forking (Steps)

- Create a fork on Bitbucket/Github/Gitlab
- Clone the forked repository to your local system
- Modify the local repository
- Commit your changes
- Push changes back to the remote fork on Bitbucket
- Create a pull request from the forked repository (source)
  - back to the original (destination)

GIT - Transparent Encryption (OpenSSL) - Part 1



- checkout → smudge → decrypt (openssl)
- commit → clean → encrypt (openssl)

GIT - Transparent Encryption (OpenSSL) - Part 2

- https://lydericblog.wordpress.com/2018/06/05/git-encryption-that-works/
- https://gist.github.com/shadowhand/873637