HPC Café – Git: Basics, common workflow, hints and tricks

Jan Eitzinger, 09.02.2021

What is this good for?

Software development is hard!

Aspect of software engineering

Software configuration management

- Identification, control, status and auditing of configuration
- Build management
- Process management
- Environment management
- Facilitate teamwork
- Defect tracing
- …

Topic of today: Version control systems (VCS)

What are the benefits?

- **Change management** (Who changed what when?) Documentation
- **Archiving of all states** (Access to all previous versions) Backup
- **Reconstruction of previous file states** (Ability to reverse changes)
- **Coordination of access** of multiple persons Synchronisation
- **Simultanious development of variants** (Branching)
- **Manage baselines**: labels and tags (Mark relevant states)

Some history and classification

Local systems:

- Source Code Control System (**SCCS**, 1972): POSIX standard!
- Revision Control System (**RCS**, 1982): created by Walter Tichy (KIT)

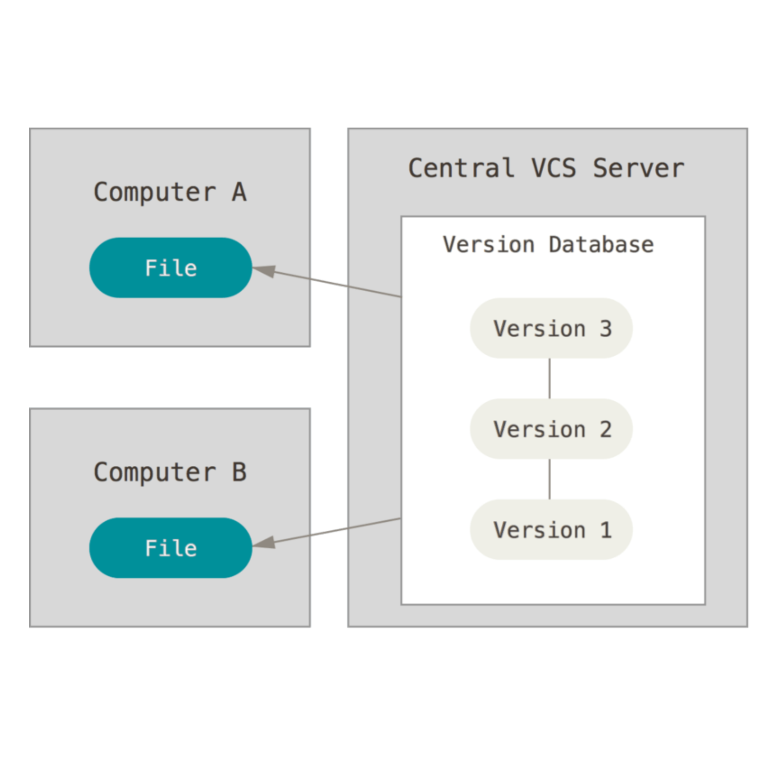
Centralized client-server:

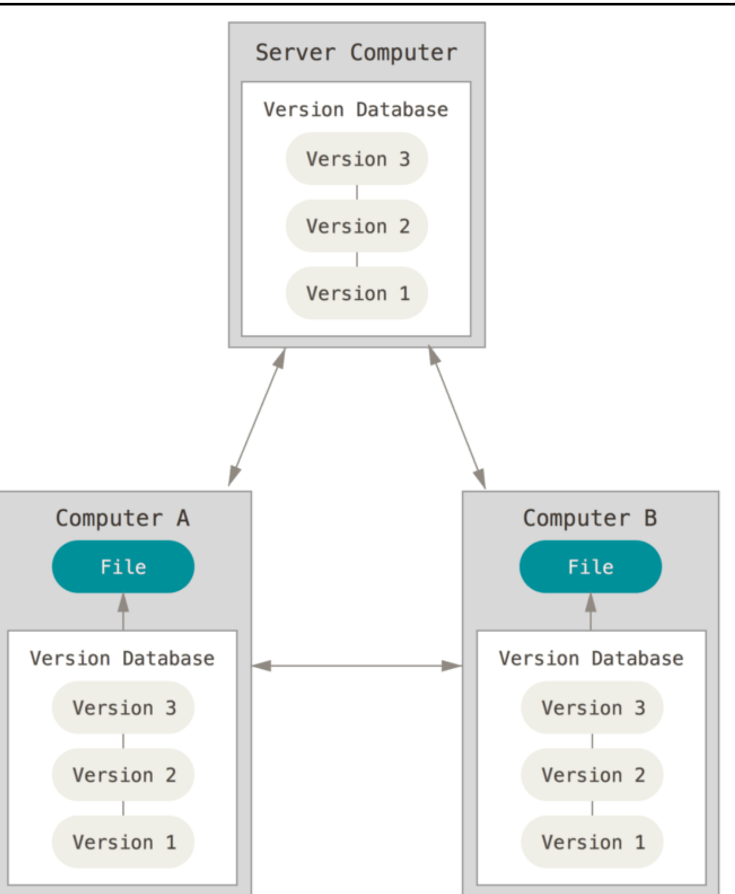
- Concurrent Versioning System (**CVS**, 1986): Frontend to RCS
- Subversion (**SVN**, 2000): CVS improved

Distributed (local + remote sync):

- Mercurial (**HG**, 2005)
- **Git** (2005): created by Linus Torvalds
- **Fossil** (2006): created by Richard Hipp (sqlite creator)

Centralized vs. Distributed VCS





Introduction to `git` (which means "unpleasant person" in British English slang)

- **De-facto standard** distributed version-control system
- Development began on 3 April 2005 by **Linus Torvalds** to create Open-Source replacement for proprietary BitKeeper
- Maintenance on 26 July 2005 handed over to Junio Hamano
- `git` is implemented as a set of **command line tools**
- Some notable features
  - Sophisticated branching support
  - Supports any workflow
  - Emphasis on data integrity (everything is checksummed)
  - Staging area allows for fine grained control about what to commit

- Website: `https://git-scm.com/`

Git Terminology - Basics

- **Repository** - The *repository* (or "repo") is where files' current and historical data are stored.
- **Clone/Fork** - Create a repository containing the revisions from another repository
- **Working copy** - The *working copy* is the local copy of files from a repository, at a specific time or revision. All work done to the files in a repository is initially done on a working copy. Also called sandbox.
- **To checkout** - Create a local working copy from the repository

Git Terminology - Synchronisation

- **To pull, push** - Copy revisions from one repository into another. *Pull* is initiated by the receiving repository, while *push* is initiated by the source.
- **Pull request** - A developer asking others to merge their "pushed" changes
- **Conflict** - A conflict occurs when different parties make changes to the same document, and the system is unable to reconcile the changes. A user must *resolve* the conflict by combining the changes, or by selecting one change in favour of the other.
- **Resolve** - The act of user intervention to address a conflict between different changes to the same document.

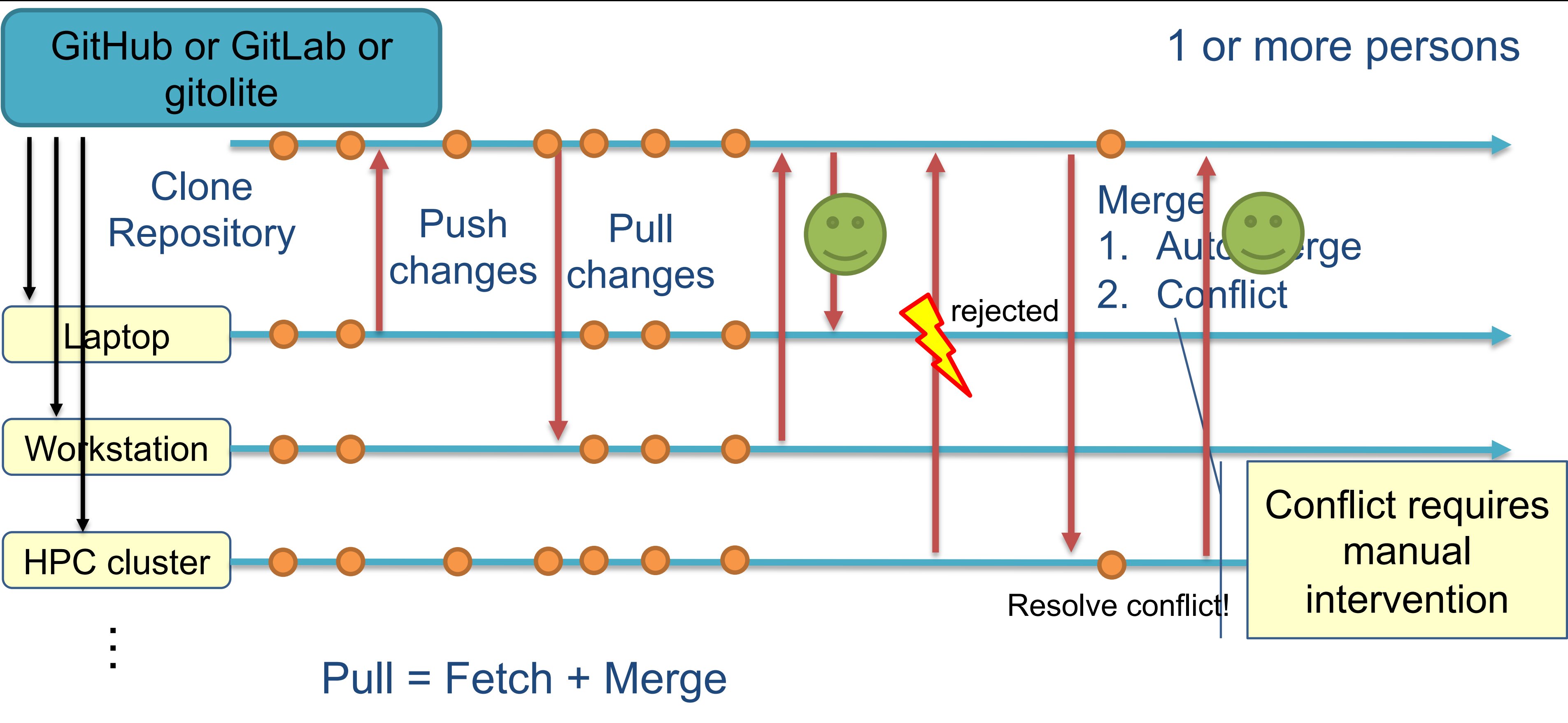
Git Terminology – Branches and commits

- **Branch** - A set of files under version control may be *branched* at a point in time; from that time forward, two copies of those files may develop at different speeds or in different ways independently of each other.
- **A Commit** - A modification that is applied to the repository
- **To commit** - Write or merge the changes made in the working copy back to the repository

Git Terminology – Revisions and Tags/Labels

- **Revision** - State at a point in time of the entire tree in the repository
- **Tag** - A *tag* or *label* refers to an important snapshot in time, consistent across many files using a user-friendly, meaningful name.
- **Trunk** or **Master** - The unique line of development that is not a branch
- **Head** – The most recent commit, either to the trunk or to a branch

Common Workflow (centralized, per Repository)



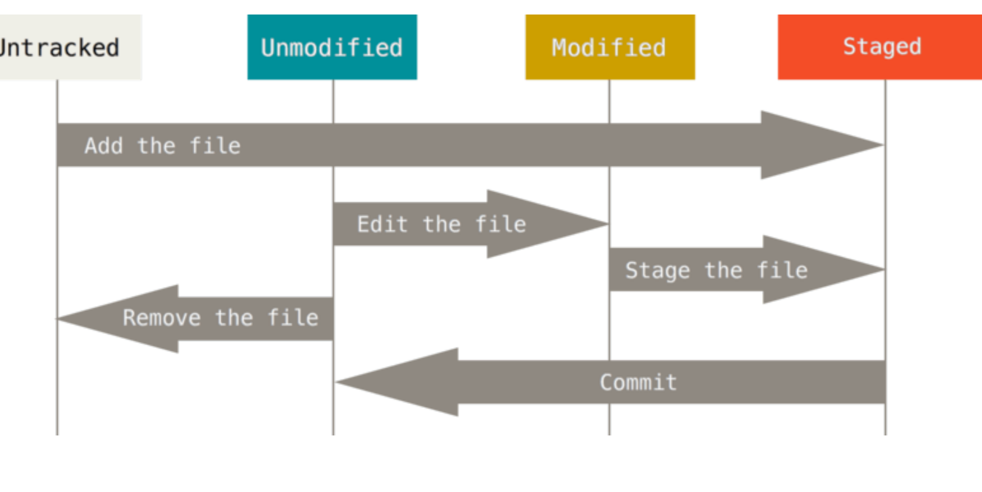
`git` – File states and workflow to apply changes

- **Tracked**: Files that were part of last snapshot (revision)
  - **Modified:** File was changed but no committed to the database yet.
  - **Staged:** Modified file is marked to go into the next commit snapshot.
  - **Committed:** Data is safely stored in the local database.
- **Untracked**: All other files

Current file version

Workflow

- **Modify files** in your working tree.
- **Stage changes** to be part of next commit
- **Commit**, save files in staging area and store snapshot permanently to the Git repository.



git – Configuration

For reference

- First time setup:

```
$ git config --global user.name "John Doe”
$ git config --global user.email johndoe@example.com
$ git config --global core.editor vim
```

Use --local to apply changes only on current repository

Or set $EDITOR environment variable

- Check settings:

```
$ git config –list
user.name=John Doe
user.email=johndoe@example.com
…
```

`git` – Getting a repository

For reference

- Create new local repository:

```
$ cd /home/user/my_project
$ git init
$ git add *.c
$ git commit -m 'Initial project version'
```

- Cloning an existing repository:

```
$ git clone git@github.com:RRZE-HPC/likwid.git
```

  - Creates a directory named likwid
  - Initializes a `.git` directory inside it and pulls down the repository data
  - Checks out a working copy of the latest version

git – Record changes

For reference

- Track new files

```
$ git add README
```

- Stage modified files

```
$ git add main.c
```

- Remove files

```
$ git rm README
```

- Move (rename) files

```
$ git mv README.md README
```

Shortcut to

```
$ mv README.md README
$ git rm README.md
$ git add README
```

- Check status of files

```
$ git status
```

- Short status

```
$ git status -s
```

- Commit staged files

```
$ git commit
```

Opens an editor

```
$ git commit –m "Fix Bla"
```

- Skip staging with

```
$ git commit –am "Fix Bla"
```

Commit all modified files

`git` – Staging area

- General advice: **Commit often!**
- Do detailed **commit messages**
- The staging area seems strange if you switch from other VCS
- But you learn to appreciate this additional step
  - Careful **review of changes**
  - **Plan** to map changes on **commits**

More on meaningful commits in a moment

You can skip staging and commit all modified files with `git commit -a`

Best Practices – Write good commit messages

The commit message should explain **What** and **Why** you did.

Capitalized, short (50 chars or less) **summary**

More **detailed explanatory text**, if necessary.

Write your commit message in the imperative: "Fix bug" and not "Fixed bug" or "Fixes bug."

Further paragraphs come after blank lines.

If you use an issue tracker, add a **reference(s)** to them at the **bottom**, like so:

Fixes: #123

Examples:

```
5ba3db6 Fix failing CompositePropertySourceTests
84564a0 Rework @PropertySource early parsing logic
e142fd1 Add tests for ImportSelector meta-data
887815f Update docbook dependency and generate epub
ac8326d Polish mockito usage
```

Simplify serialize.h's exception handling

Remove the 'state' and 'exceptmask' from serialize.h's stream implementations, as well as related methods.

As exceptmask always included 'failbit', and setstate was always called with bits = failbit, all it did was immediately raise an exception. Get rid of those variables, and replace the setstate with direct exception throwing (which also removes some dead code).

…

fail(), clear(n) and exceptions() are just never called. Delete them.

On the command line:

```
git commit -m "Subject" -m "Description..."
```

Best practices: How to review changes?

- On the command line

```
$ git diff
```

see changes in tracked unstaged files

Note: Staging is possible for parts of a file!

```
$ git diff  --staged
```

see changes in staged files

BUT: You really want to review changes in an editor or GUI!

Configure tool for git diff

For reference

- Tell git which configuration to use for diff

```
$ git config --local diff.tool meld
```

For global configuration, use --global

- Tool configuration

```
$ git config --local difftool.meld.cmd meld ”$LOCAL” ”$REMOTE”
```

- Disable y/n prompts every time you open the diff tool

```
$ git config --local difftool.prompt false
```

- Now you can get tool-based diffs with (git diff uses terminal output!)

```
$ git difftool <file(s)>
```

- If the tool is not in $PATH

```
$ git config --local difftool.meld.path /usr/bin/meld
```

Best practices: Which files to track?

Only put source files under revision control!

No intermediate build products, hidden system files, binaries, libraries …

Fine grained control which files to skip with `.gitignore` files:

```
# ignore all .a files
*.a
# but do track lib.a, even though you're ignoring .a files above
!lib.a
# only ignore the TODO file in the current directory, not subdir/TODO
/TODO
# ignore all files in any directory named build
build/
# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt
# ignore all .pdf files in the doc/ direc
doc/**/*.pdf
```

Usually one `.gitignore` in root of repo, but can be in multiple directories

GitHub maintains good list of examples:
`https://github.com/github/gitignore`

Fixing Mistakes and undoing Things

For reference

To add a file or fix the commit message of the previous commit

```
$ git commit -m 'Initial commit'
$ git add forgotten_file
$ git commit --amend
```

Replace previous commit. Only use on not yet pushed commits!

Unstaging a staged file

```
$ git reset HEAD  README.md
```

Unmodifying a modified File

```
$ git checkout -- README.md
```

Danger zone! This will delete unsaved local changes.

Better use `git stash`.

Undoing last commit(s)

```
$ git reset --hard a1e8fb5
```

Reset history to specific commit

Git 2.23 introduces new command for above actions: `git restore`

Basic Branching and Merging

For reference

- In git branches are pointers to snapshots
- The **HEAD points** to the most recent snapshot of the **current branch**

- Create new branch

```
$ git branch testing
```

create pointer to current last commit

Creates the branch but does not switch to it

- Switch to existing branch

```
$ git checkout testing
```

You cannot switch branches if there are uncommitted changes!

- Shorthand for branch+checkout

```
$ git checkout -b testing
```

Basic Branching workflow

- Typical use case

```
$ git checkout -b hotfix
# Modify files and commit changes
$ git checkout master
$ git merge hotfix
$ git branch -d hotfix
```

Doing tests and review on hotfix branch

Merge changes from hotfix into master

Delete branch

- Git tries to merge changes automatically
- If this fails a **merge conflict occurs**

```
$ git status
You have unmerged paths
Unmerged paths: index.html
```

Pro Tip: Use branching! Often!

`master` should always be a working and tested state!

Resolving merge conflicts

- Git will add standard conflict resolution markers in unmerged files

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

- Resolve conflict by editing the files, add and commit the resolved state:

```
$ git add index.html
$ git commit
```

You want to to do this in a editor or GUI with an integrated merge tool!

Configure tool for merge conflict resolution

For reference

- Tell git which configuration to use for conflict resolution

```
$ git config --local merge.tool meld
```

- Tool configuration

```
$ git config --local mergetool.meld.cmd \
     meld "LOCAL" "MERGED" "REMOTE" --output "MERGED"
```

- Disable y/n prompts every time you open the merge tool

```
$ git config --local mergetool.prompt false
```

- Now you can get tool-based conflict resolution with

```
$ git merge
$ git mergetool <file(s)> # if auto merge fails
```

- If the tool is not in $PATH

```
$ git config --local mergetool.meld.path /usr/bin/meld
```

Dealing with Remotes

For reference

- Remote repositories are **other versions** of your project
- A **remote** always **creates** an **implicit branch** that may require merging!

A clone from a URL will automatically create a remote called origin

```
$ git clone git@github.com:RRZE-HPC/likwid.git
$ cd likwid
$ git remote -v
origin git@github.com:RRZE-HPC/likwid.git (fetch)
origin git@github.com:RRZE-HPC/likwid.git (push)
```

List all remotes

Adding an additional remote

```
$ git remote add mylikwid https://github.com/jan/likwid
```

Synchronization with remotes

For reference

Get all data from a remote project that you don't have yet

```
$ git fetch <remote>
```

Creates remote tracking branches: e.g. **`origin/master`**

Merge changes from remote branch

```
$ git merge origin/master
```

Shorthand for fetch+merge

```
$ git pull
```

For this to work you need a tracking branch!

Push changes back to remote

```
$ git push <remote> <branch>
```

Explicitly create tracking branch

```
$ git checkout -b <branch> <remote>/<branch>
```

That’s complicated! Show me the simple case.

- Clone from a remote repository

```
$ git clone  <URL>
```

This will:
- clone the repository
- create a remote named origin
- set up tracked remote branches
- checkout origin/master

- Pull new changes from remote (fetch and merge)

```
$ git pull
```

You need to commit your local changes first!

- Push new revisions to remote origin

```
$ git push
```

Often requires to first pull and merge remote changes first!

Viewing the History and other Information

For reference

- List all commits in reverse chronological order

```
$ git log
```

```
$ git log --pretty=oneline
```

Compact commit list

- Show local branches

```
$ git branch
```

- Show all branches (including tracked remote branches)

```
$ git branch --all
```

- Inspect remote

```
$ git remote show <remote>
```

Creating Baselines with Tagging

For reference

Mark specific points in a repository's history with meaningful names

- List tags

```
$ git tag
```

- Tag recent commit

```
$ git tag v1.0
```

- Tag previous commit

```
$ git tag -a v1.2 9fceb02
```

Part of commit checksum

- Push tags to remote server

```
$ git push origin --tags
```

```
$ git push origin v1.5
```

`git push` does not transfer tags to remote server!

Communication protocols and Authentication

- Four protocols are available for communication between remotes
  - Local Protocol – `file:///srv/git/project.git`
  - HTTP Protocol – `https://example.com/gitproject.git`
  - SSH protocol – `[user@]server:project.git`
  - Git protocol – `git://example.com/gitproject.git`

- Most common case
  - HTTPS for read only clone
  - SSH for read/write access using ssh key authentication

Some remarks on GUIs

GUI options

- Builtin GUI options for committing (**`git-gui`**) and browsing (**`gitk`**)
- GitHub Desktop (Windows/Mac, **`https://desktop.github.com/`**)
- SourceTree (Windows/Mac, **`https://www.sourcetreeapp.com/`**)

Editor integration

- vim-fugitive plugin (**`https://github.com/tpope/vim-fugitive`**)

- Magit (**`https://magit.vc`**)

- Visual Studio Code has a powerful builtin git integration

… and Servers

Git ist commonly used as **software as a service** offering:

- **GitHub** (Microsoft)

- **GitLab** (Open Core company)

self-hosted Option

Other options:

- gitolite (central git server with fine-grained access control)
  `https://gitolite.com/`
- cgit (minimalistic web interface, can be combined with gitolite)
  `https://git.zx2c4.com/cgit/about/`

Outlook and further information

- Things we did not cover in this talk
  - Workflows and development strategies
  - Git stash and clean
  - Git rebase
  - Submodules
  - And a lot of other stuff!

Topics for next event?

- General Software engineering (including work flows)
- Build system make
- Effective Vim Editing
- Software testing and CI
- Other suggestions …

- Very good **online book** for self study:

`https://git-scm.com/book/en/v2`

- Tutorial style introduction

`https://www.atlassian.com/git/tutorials`