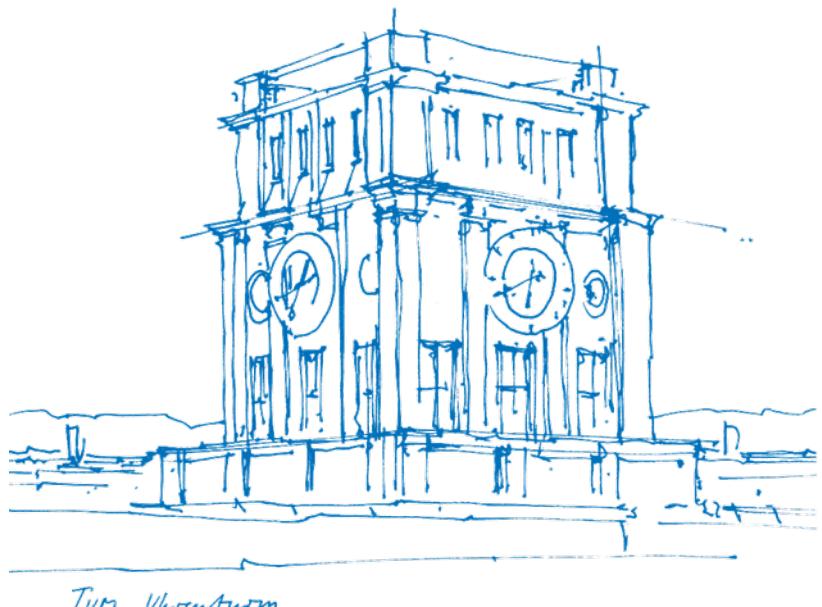


Open Source Lab - Git Basics

Fabian Sauter, Christian Menges
Technical University of Munich
Department of Informatics
Chair of Connected Mobility
Garching, 20th October 2021



Credits

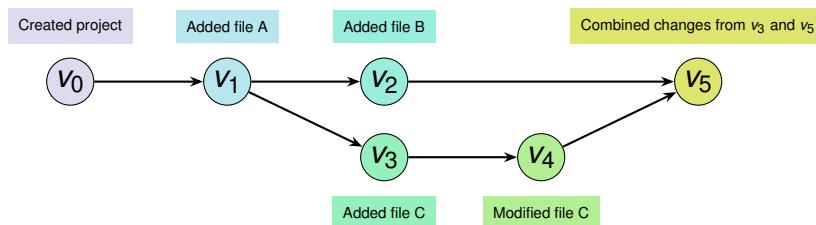
These slides are based on the awesome materials from:

Moritz Sichert (sichert@in.tum.de) }
Michael Freitag (freitigm@in.tum.de) } Systems Programming in C++ (Practical Course)

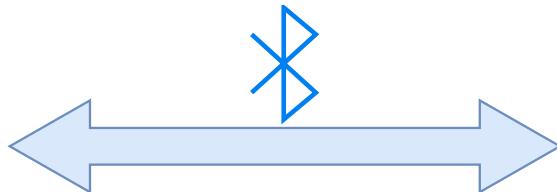
Pro Git book: <https://git-scm.com/book>

Version Control Systems (VCS)

- Code projects evolve gradually
- Incremental changes, also called *versions*, should be tracked to allow:
 - Documentation of the project history
 - Selective inspection/modification of specific versions
 - Efficient collaboration when working in a team
- A **Version Control System (VCS)** manages versions, usually represent them in a directed acyclic graph



A more concrete example...



A more concrete example...

```
int mo6312r(int i, int i2, int i3, int i4) {
    byte[] bArr = arr1;
    int i5 = i2 >> 4;
    return mo6308n(mo6308n((bArr[mo6308n(((arr2[mo6308n(((bArr[mo6308n((i + i2) + i3) % 16] + i4) + mo6308n
        (i5)) - i2) - i3) % 16] + i3) + i2) - i4) - mo6308n(i5)) % 16] - i2) - i3) % 16);
}
```

A more concrete example...

```
int mo6312r(int i, int i2, int i3, int i4) {
    byte[] bArr = arr1;
    int i5 = i2 >> 4;
    return mo6308n(mo6308n((bArr[mo6308n(((arr2[mo6308n(((bArr[mo6308n((i + i2) + i3) % 16] + i4) + mo6308n
        (i5)) - i2) - i3) % 16] + i3) + i2) - i4) - mo6308n(i5)) % 16] - i2) - i3) % 16);
}
```

```
int mo6308n(int i) {
    while (i > 255) {
        i += InDevComp.SOURCE_ANY; // -256
    }
    while (i < 0) {
        i += 256;
    }
    return i;
}
```

- Looks like a fancy mod256 for Java.
- So let's replace it...

A more concrete example...

```
int mo6312r(int i, int i2, int i3, int i4) {
    byte[] bArr = arr1;
    int i5 = i2 >> 4;
    return mod256(mod256((bArr[mod256(((arr2[mod256(((bArr[mod256((i + i2) + i3) % 16] + i4) + mod256(i5))
        - i2) - i3) % 16] + i3) + i2) - i4) - mod256(i5)) % 16] - i2) - i3) % 16);
}
```

A more concrete example...

```
int mo6312r(int i, int i2, int i3, int i4) {
    byte[] bArr = arr1;
    int i5 = i2 >> 4;
    return mod256(mod256(bArr[mod256(((arr2[mod256(((bArr[mod256(i + i2 + i3) % 16] + i4) + mod256(i5) - i2)
        - i3) % 16] + i3) + i2) - i4) - mod256(i5)) % 16] - i2 - i3) % 16);
}
```

- Let's continue simplifying and remove parentheses...

A more concrete example...

```
uint8_t shuffle(int dataNibble, int nibbleCount, int keyLeftNibbel, int keyRightNibbel) {  
    uint8_t i5 = mod256(nibbleCount >> 4);  
    uint8_t tmp1 = numbers1[mod256(dataNibble + nibbleCount + keyLeftNibbel) % 16];  
    uint8_t tmp2 = numbers2[mod256(tmp1 - keyRightNibbel + i5 - nibbleCount - keyLeftNibbel) % 16];  
    uint8_t tmp3 = numbers1[mod256(tmp2 + keyLeftNibbel + nibbleCount - keyRightNibbel - i5) % 16];  
    return mod256(tmp3 - nibbleCount - keyLeftNibbel) % 16;  
}
```

- And now split it up and rename everything...

A more concrete example...

```
uint8_t shuffle(int dataNibble, int nibbleCount, int keyLeftNibbel, int keyRightNibbel) {  
    uint8_t i5 = mod256(nibbleCount >> 4);  
    uint8_t tmp1 = numbers1[mod256(dataNibble + nibbleCount + keyLeftNibbel) % 16];  
    uint8_t tmp2 = numbers2[mod256(tmp1 - keyRightNibbel + i5 - nibbleCount - keyLeftNibbel) % 16];  
    uint8_t tmp3 = numbers1[mod256(tmp2 + keyLeftNibbel + nibbleCount - keyRightNibbel - i5) % 16];  
    return mod256(tmp3 - nibbleCount - keyLeftNibbel) % 16;  
}
```

- And now split it up and rename everything...

And then we notice, our tests do not produce the same results as the Java code any more.

What do we do now?

Reset and start over?

Spend hours debugging?

An other example



Hi, kann jemand von den Tutoren mein Bash-Hausaufgaben-Repository löschen/neu starten, damit ich die Hausaufgaben wiederholen kann?
Ich habe etwas falsch gemacht und jetzt fehlen die Dateien `setup.sh` und `script.sh` komplett
(gelöscht)

An other example



Hi, kann jemand von den Tutoren mein Bash-Hausaufgaben-Repository löschen/neu starten, damit ich die Hausaufgaben wiederholen kann?
Ich habe etwas falsch gemacht und jetzt fehlen die Dateien `setup.sh` und `script.sh` komplett
(gelöscht)

He deleted his code and asks us whether we could reset it for him, so he could start over again.
But since he uses git, he can go back to any version of his code by him self.

Git

- Many VCS exist, Git is a very popular one: Used by projects like Linux, GCC, LLVM, etc.
- Git in particular has the following advantages compared to other version control systems (VCS):
 - Open source (GPLv2.1)
 - Decentralized, i.e. no server required
 - Efficient management of *branches* and *tags*
- All Git commands are documented with man-pages (e.g. type `man git-commit` to see documentation for the command `git commit`)
- Pro Git book: <https://git-scm.com/book>
- Git Reference Manual: <https://git-scm.com/docs>

Git History

- **Initiator** Linus Torvalds
 - **Goals** speed, "simple" design, fully distributed, able to handle large projects, ...
 - **git is British slang for** "pig headed, think they are always correct, argumentative"
 - **Quoting Linus** "I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'Git'."¹
 - **1991-2002** Changes were passed as patches and archived files.
 - **2002** The Linux kernel project began using a proprietary DVCS called BitKeeper.
 - **April 2005** The Relationship between the commercial company behind BitKeeper and the Linux Kernel community broke down.
 - SourcePuller was created by reverse engineering the BitKeeper protocols.
 - Free use licence got withdrawn.
- ⇒ Linus Torvalds started working on an alternative, called **git**.
- **7. April 2005** git is self-hosted²

¹https://git.wiki.kernel.org/index.php/GitFaq#Why_the_.27Git.27_name.3F

²<https://marc.info/?l=git&m=117254154130732>

Git Concepts

Repository: A collection of Git objects (*commits* and *trees*) and references (*branches* and *tags*).

Branch: A named reference to a *commit*. Every repository usually has at least the `master` (`main`) branch and contains several more branches, like `fix-xyz` or `feature-abc`.

Tag: A named reference to a *commit*. In contrast to a branch a tag is usually set once and not changed. A branch regularly gets new commits.

Commit: A snapshot of a *tree*. Identified by a SHA1 hash. Each commit can have multiple parent commits. The commits form a directed acyclic graph.

Tree: A collection of files (not directories!) with their path and other metadata. This means that Git does *not* track empty directories.

Creating a Git Repository

Created a new directory (home) for our repository and change into it.

```
mkdir myRepo && cd myRepo
```

Initialize a new Git repository.

```
git init
```

Sets the name that will be used when creating a commit.*

```
git config --global user.name Firstname Lastname
```

Sets the e-mail address that will be used when creating a commit.*

```
git config --global user.email first.last@in.tum.de
```

Shows the current status and information for this repository.

```
git status
```

*Required only for the first time you create a Repository.

Creating a Git Repository

```
$ git config --global user.name Firstname Lastname
$ git config --global user.email first.last@in.tum.de
$ mkdir myRepo && cd myRepo
$ git init
Initialized empty Git repository in /tmp/myRepo/.git/
$ git status
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
$ ls -la # Show the contents of the directory
total 0
drwxrwxr-x. 3 user user 60 Sep 7 13:16 .
drwxrwxrwt. 31 root root 780 Sep 7 13:35 ..
drwxrwxr-x. 7 user user 200 Sep 7 13:16 .git
```

Cloning an existing Git Repository

Usually we do not want to start a new repository, instead we want to [contribute](#) to an existing one.

Creating a local copy (cloning) some remote repository.

```
git clone <remote>
```

Example cloning the cpr repository from GitHub.

```
git clone https://github.com/libcpr/cpr.git
Cloning into 'cpr'...
remote: Enumerating objects: 4969, done.
remote: Counting objects: 100% (557/557), done.
remote: Compressing objects: 100% (309/309), done.
remote: Total 4969 (delta 324), reused 356 (delta 224), pack-reused 4412
Receiving objects: 100% (4969/4969), 1004.03 KiB | 3.16 MiB/s, done.
Resolving deltas: 100% (3300/3300), done.
```

Branches

A branch is a [named reference](#) to a specific commit.

Gives you a list of all (local) branches and which is currently active.

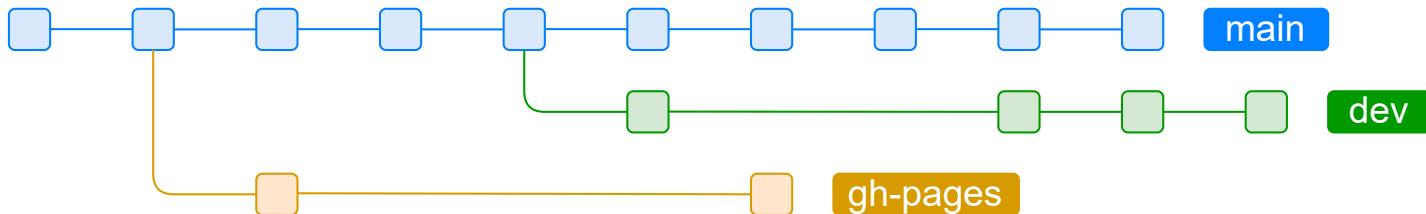
```
git branch
```

Create a new branch from the current commit.

```
git branch <name>
```

Switch to another branch, i.e. change all files in the working directory so that they are equal to the tree of the other branch.

```
git switch <name>
```



Tags

A named reference to a commit. In contrast to a branch, a tag is usually [set once](#) and not changed. A branch regularly gets new commits.

Gives you a list of all (local) tags.

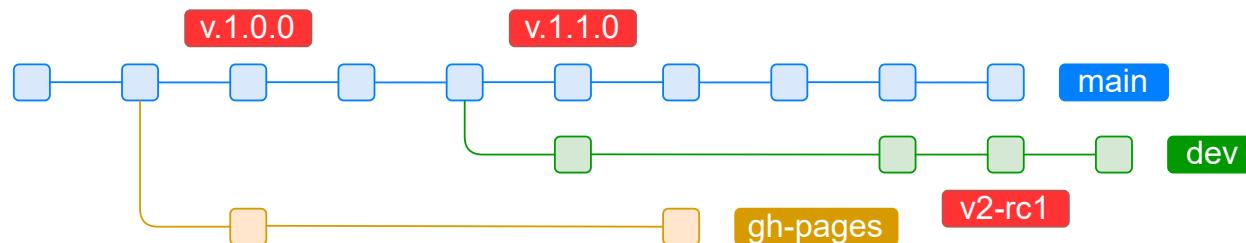
git tag

Create a new tag from the current commit. With the “-s” option you can sign it using a PGP key.

```
git tag [-s] <name>
```

Checkout the given tag. Be aware to make changes you need to create a branch first!

```
git checkout <tag>
```



Commits

A snapshot of a *tree*. Identified by a SHA1 hash. Each commit can have multiple parent commits. The commits form a directed acyclic graph.

Stages all changes inside the given file and starts tracking it in case it is not already being tracked.

```
git add <file>
```

With this you can bundle all your staged changes (git add) to one commit with a commit message.

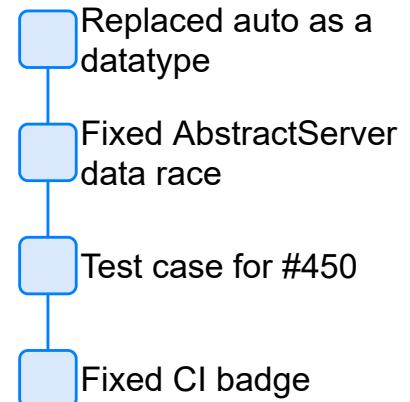
The "-S" option allows you to sign commits using a PGP key.

```
git commit [-S] -m "Some message"
```

This enables signing commits by default for all your repositories.

```
git config --global commit.gpgsign true
```

main



Commit Messages

A few [guidelines](#) for creating commits:

Dos

- Commit early and often.
- Split up your work into atomic commits.
- Make commit messages meaningful.
- Subject should be less than 50 characters.
- Do not end the subject line with a ":".
- Separate the subject and body by a blank line.

Commit Messages

A few [guidelines](#) for creating commits:

Dos

- Commit early and often.
- Split up your work into atomic commits.
- Make commit messages meaningful.
- Subject should be less than 50 characters.
- Do not end the subject line with a ":".
- Separate the subject and body by a blank line.

Don'ts

	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSOKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT
MESSAGES GET LESS AND LESS INFORMATIVE.

Figure: "Git Commit" by xkcd

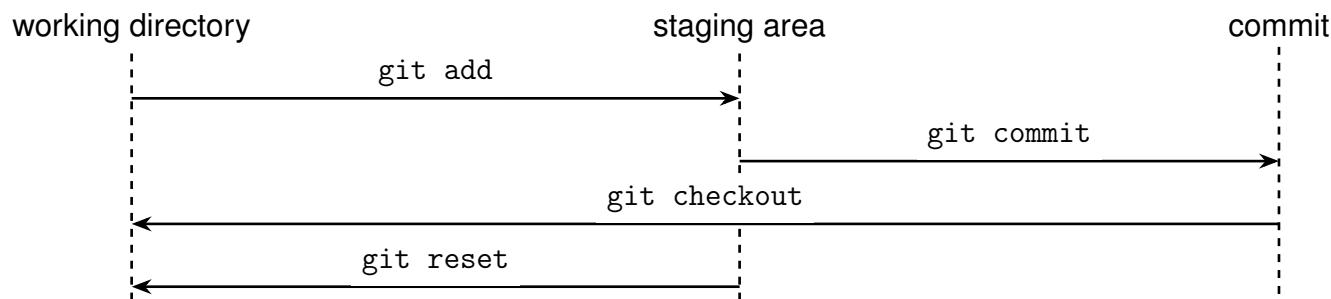
Funny, but not recommended: <http://whatthecommit.com>

Git Working Directory and Staging Area

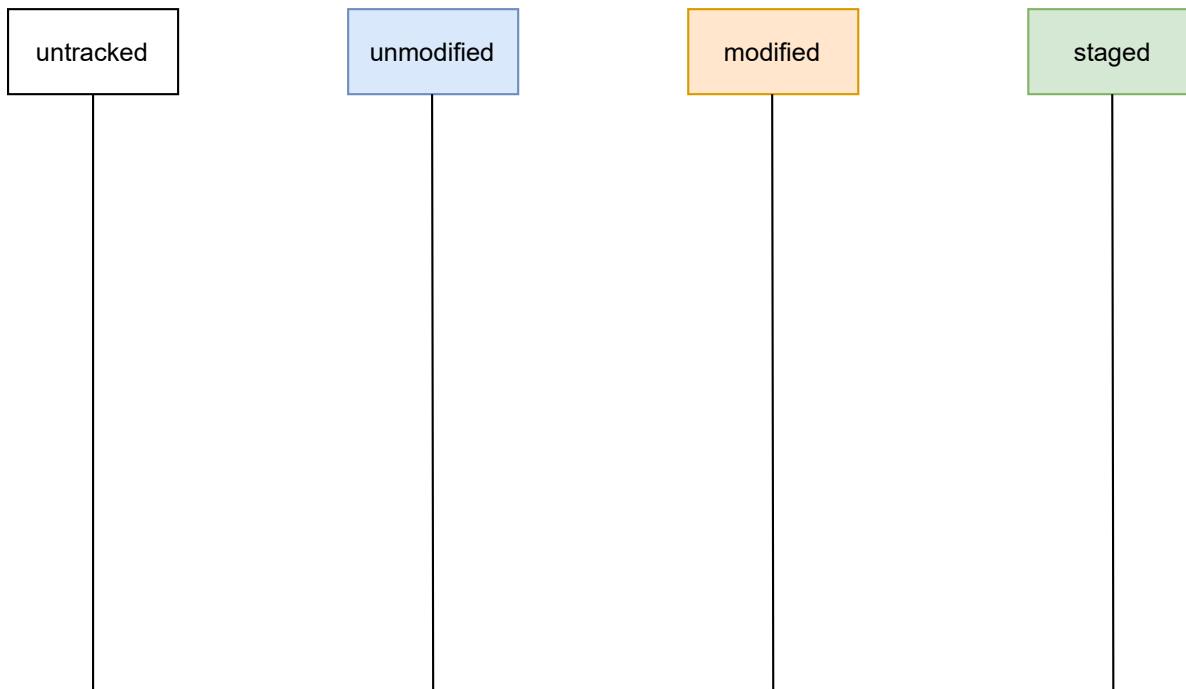
When working with a Git repository, changes can live in any of the following places:

- In the working directory (when you edit a file)
- In the staging area (when you use `git add`)
- In a commit (after a `git commit`)

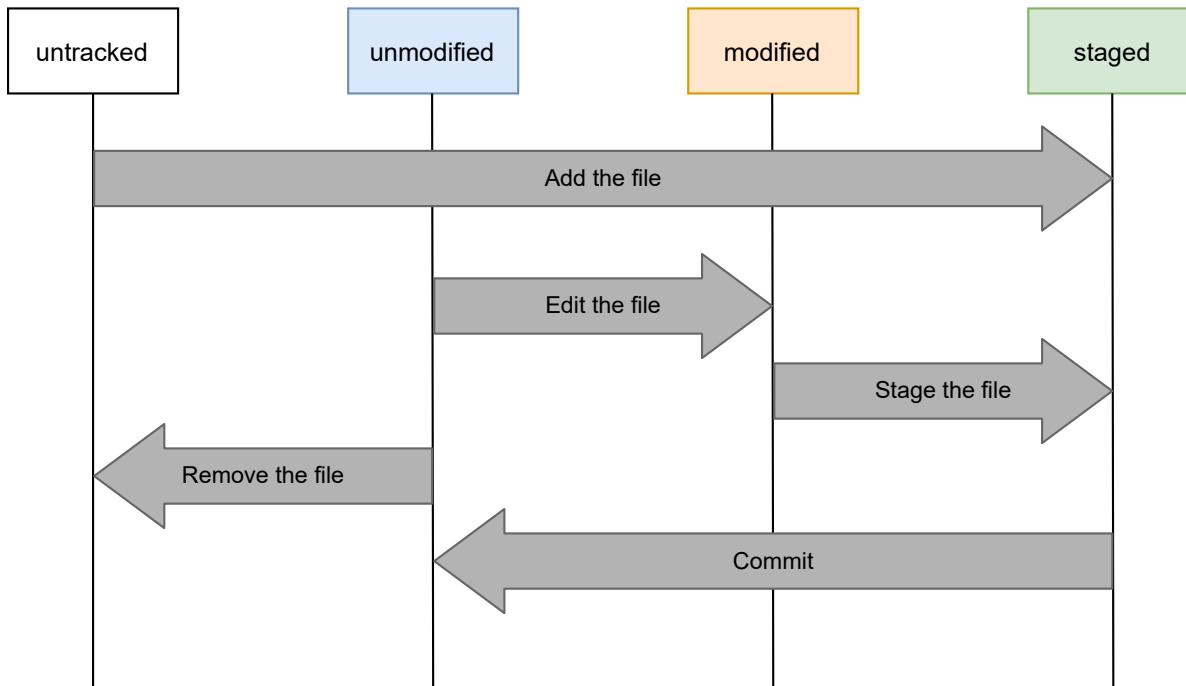
Once a change is in a commit and it is referenced by at least one branch or tag you can always restore (go back to) it, even if you remove the file.



Livecycle of a File



Livecycle of a File



History

Allows you to inspect the git [commit history](#).

--online condenses every commit into one line.

--graph shows the git history as an ASCII graph

```
git log [--oneline] [--graph]
```

Example

```
$ git log --oneline --graph
* bc2a09a (HEAD -> main) Merge branch 'feature-print'
|\ 
| * ad80c4c (feature-print) Fixed print helper new line
| * 7fce54f Added print helper framework
* | 17fdfab Added reader framework
|/
* 3783ea8 Added main call
```

Pushing and Pulling

When working in a team, it is required to [synchronize changes](#) between the individual team members.

⇒ For this a remote repository is being used where everybody pushes its changes (commits) to.

Usually called "origin".

Upload the current branch to a remote repository.

 "-f" force overrides the remote branch. Required in case you changed the git history (deleting commits/rebasing/...). Be extremely careful with this option!

```
git push [-f]
```

Retrieve the latest metadata from `origin` and check if there are changes available.

```
git fetch
```

Fetches changes from `origin` and merges/rebases them with your local changes.

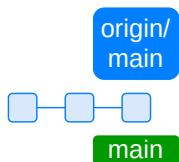
```
git pull [--rebase]
```

Enables rebasing instead of merging by default.

```
git config --global pull.rebase true
```

Push and Pull in Action

Local

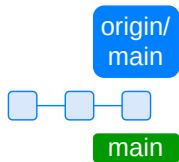


Remote

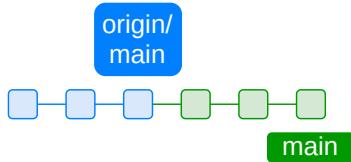


Push and Pull in Action

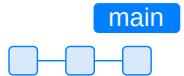
Local



You create three new commits.

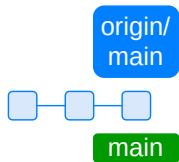


Remote

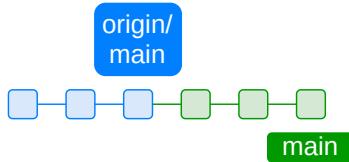


Push and Pull in Action

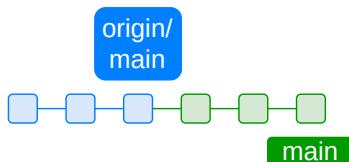
Local



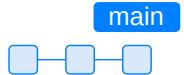
You create three new commits.



In the meantime, somebody else pushed two new commits to remote.

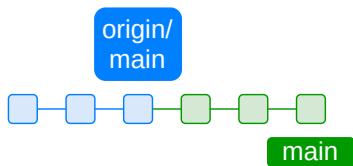


Remote



Push and Pull in Action

Local

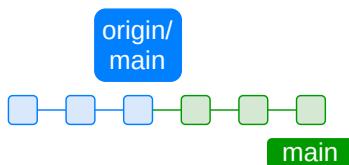


Remote

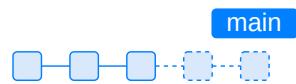


Push and Pull in Action

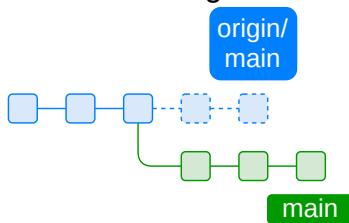
Local



Remote

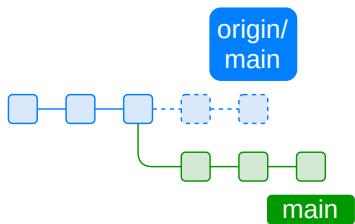


Fetch for changes from remote with `git fetch`



Push and Pull in Action

Local

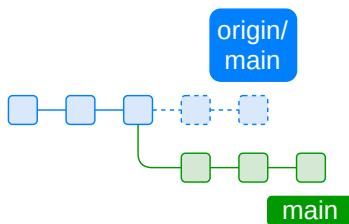


Remote



Push and Pull in Action

Local



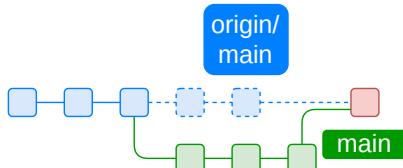
Remote



To apply changes we have found with `git fetch`, we use `git pull`

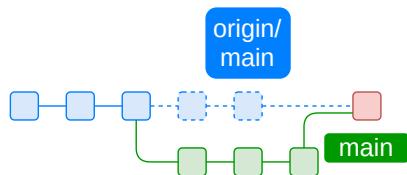
By default this will **merge** changes and create a so called "merge commit".

In some cases, creating a merge commits is considered **bad practice**. More on that later...



Push and Pull in Action

Local

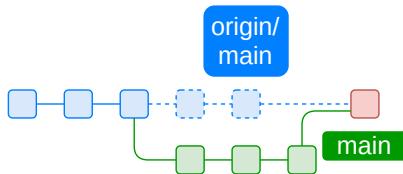


Remote



Push and Pull in Action

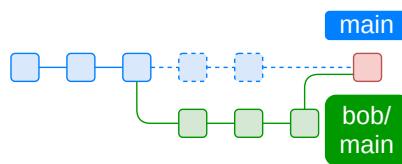
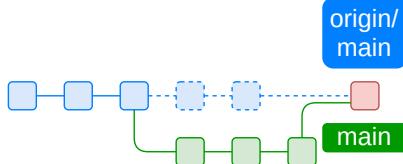
Local



Remote



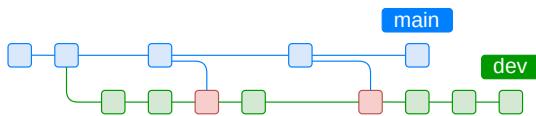
To push our changes to remote, we use `git push`



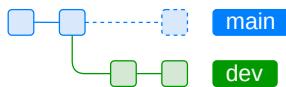
Now both, our local and remote branch have the same history and we are done.

Merge and Rebase

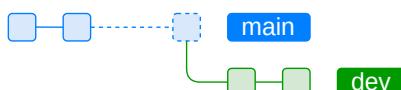
To stay up to date with the changes in `main`, we **merged** them into our `dev` branch twice until now. As a result we got two (red) **merge commits**, each with their own commit message.



This is considered **bad practice**. Instead we should rebase.



```
git rebase main
```



While a `merge` keeps your history, `rebase` will rewrite it for all commits in our `dev` branch!

Dealing with Merge Conflicts

Merge conflicts occur when **git is unable to merge changes** from two commits since both change the same lines in a file.
They can happen when...

- **merging** a branch.
- **rebasing** a branch.
- **cherry picking** a commit.

Now **you** have to decide which code to keep!

Example

```
$ git merge feature-main
Auto-merging reader.cpp
CONFLICT (content): Merge conflict in reader.cpp
Automatic merge failed; fix conflicts and then commit the result.
```

Dealing with Merge Conflicts

Git changed the `reader.cpp` file in a way, so now have to decide what to keep.

```
<<<<< HEAD
int main(int /*argc*/, char** /*argv*/) { }
=====
int main(void) { }
>>>>> feature-main
```

Once we are done, we can commit our decision.

```
git commit -m "Merged feature-main into main"
```

Reverting Changes

Create a new commit that is the "inverse" of the specified commit.

```
git revert <commit-hash>
```

Reset the current branch to the last commit. No files are changed.

"<commit-hash>" allows you to specify a specific commit to revert to.

⚠ "--hard" resets all files in the current working directory back to the specified commit.

```
git reset [--hard] [<commit-hash>]
```

Shows a history of SHA1 commit hashes that were added or removed.

Allows to restore removed commits if they were not garbage collected yet.

```
git reflog
```

Cherry Picking

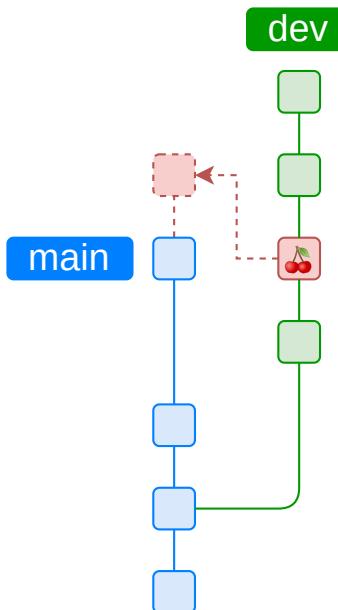
Useful when you committed changes to the [wrong branch](#).

Allows you to apply the changes from one commit to your current branch.

Use sparingly to prevent [duplicate commits](#), which will result in merge conflicts later on.

```
git cherry-pick <commit-hash>
```

Sometimes merging or rebasing is more appropriate.



.gitignore

Allows you to specify intentionally untracked (ignored) files.

Patters:

- “!” negates the following pattern.
- “*” matches anything except a “\”.
- “**” matches anything before or after the given path.

Example

```
# Build directory
build/
!build/bin/**

# Debug files
*.dSYM/
*.su
```

A collection of .gitignore templates: <https://github.com/github/gitignore>

.gitattribute

Allows you to specify attributes for paths. For example to normalize line endings.

Line format:

```
pattern attr1 attr2 ...
```

Here the pattern is the same as for .gitignore without negative patterns.

Example

```
*          text=auto
*.txt      text
*.vcproj   text eol=crlf
*.sh       text eol=lf
```

.gitattribute generator: <https://gitattributes.io>

The right Tool for the right Job

Sometimes using git in a terminal is too cumbersome or degrades to just copy and pasting complex commands from Stack Overflow...

- There exist a bunch of graphical user interfaces for git³.
- They support you in performing "complex" actions like merging, rebasing and solving merge conflicts.
- Most code editors come with basic git support (like in visual studio (code)).
- Two popular alternatives:
 - Sublime Merge
 - GitKraken

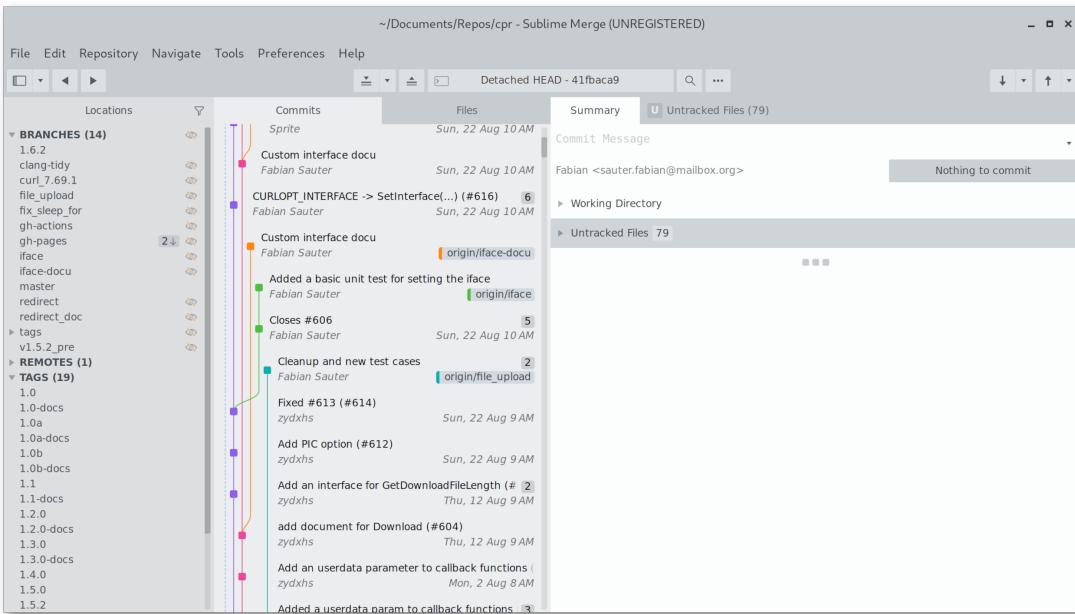
³<https://git-scm.com/downloads/guis>

Sublime Merge⁴

Platforms: Linux, Mac, Windows

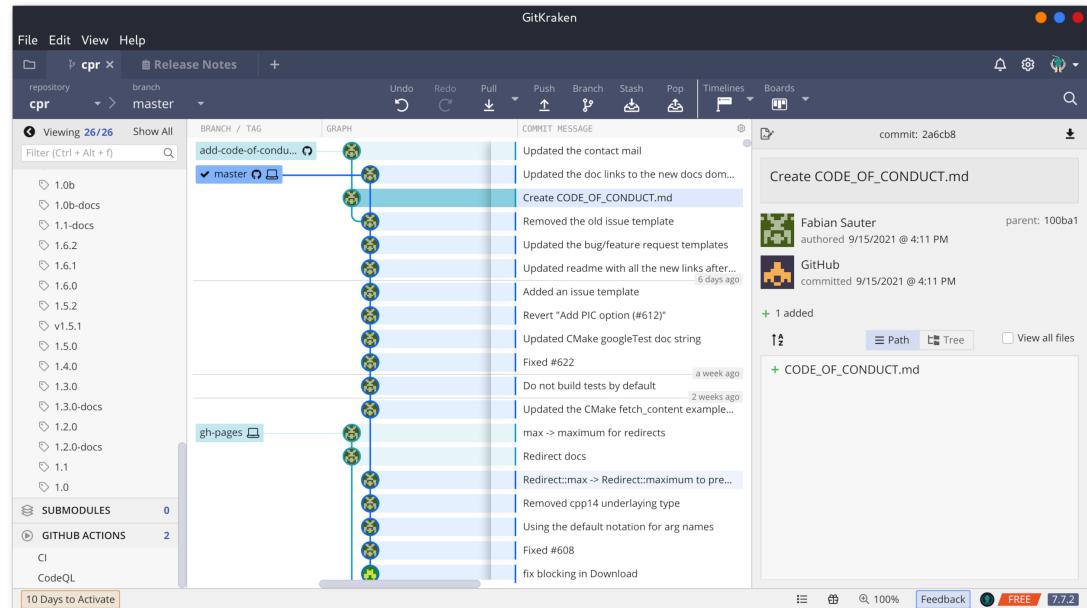
Price: \$99/user, \$75 annual business
sub, free eval

License: Proprietary



⁴<https://www.sublimemerge.com/>

GitKraken⁵



Platforms: Linux, Mac, Windows
Price: Free / \$29 / \$49
License: Proprietary

⁵<https://www.gitkraken.com/>

Summary

1. a) Create repository

```
git init
```

1. b) Clone repository

```
git clone <remote>
```

2. Make changes

3. Add changes

```
git add <file>
```

3. Commit changes

```
git commit -m "Some message."
```

4. Done? No - Go back to 2. Yes - Continue

5. Rebase

```
git pull --rebase
```

6. Push

```
git push
```

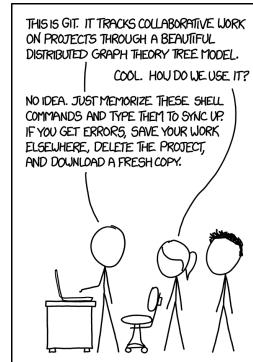


Figure: "Git" by xkcd