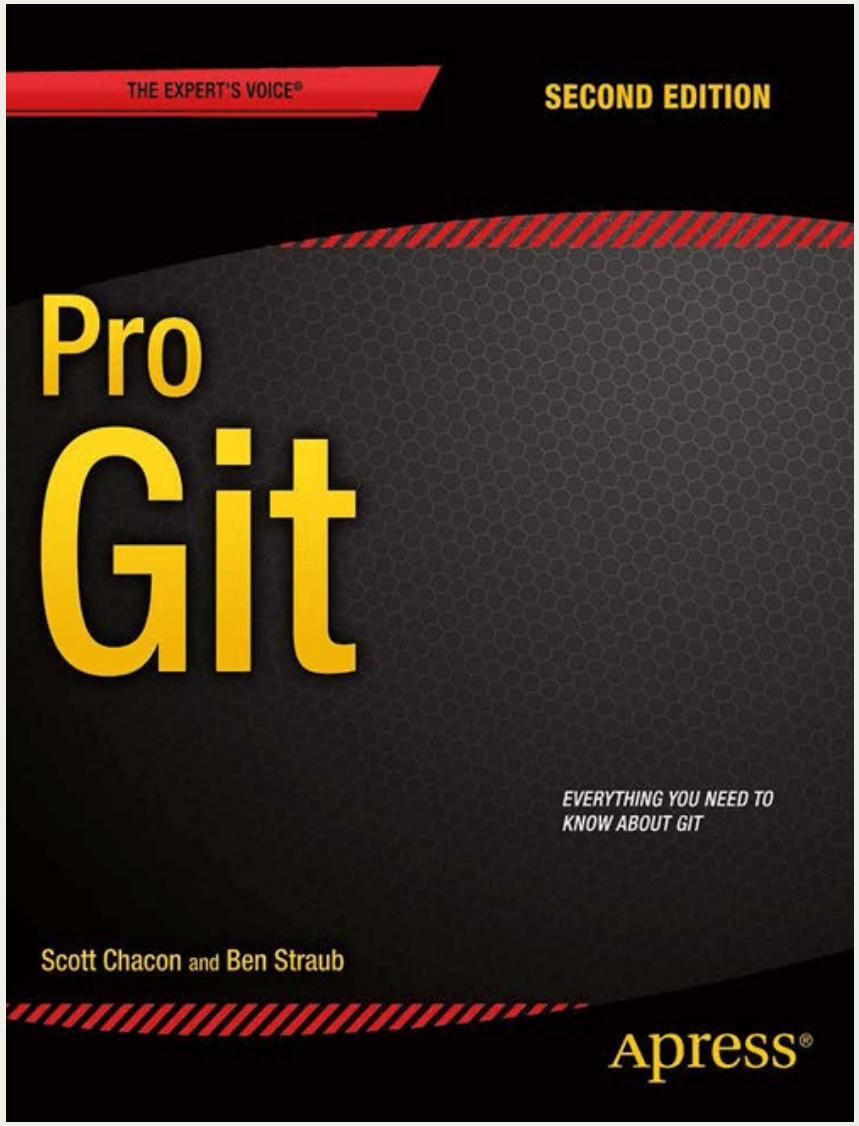
Process Solutions Development

Git: Not as dangerous as it seems!

Source?

The presented points within this course will largely be based on ideas & principles from:

- "Pro Git" by Scott Chacon and Ben Straub
  - The book is available for download from the official Git-Documentation
- What I found on the internet
  - The Git-Documentation (git-scm.com)
  - Atlassian.com
  - Other

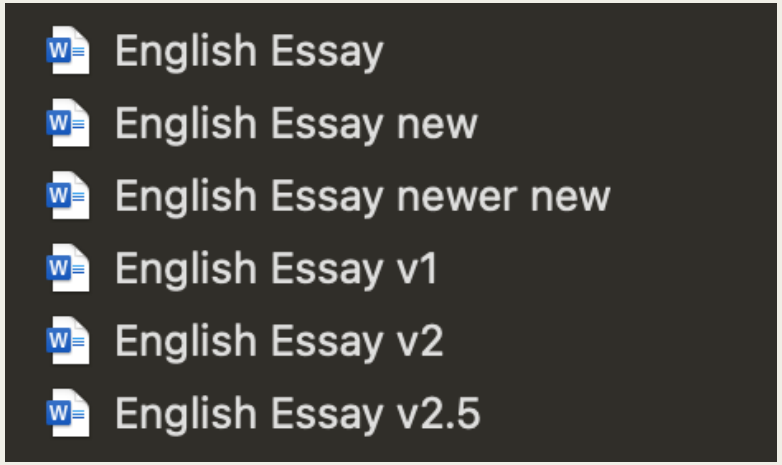


Version Control

What Git actually is

Version Control

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.

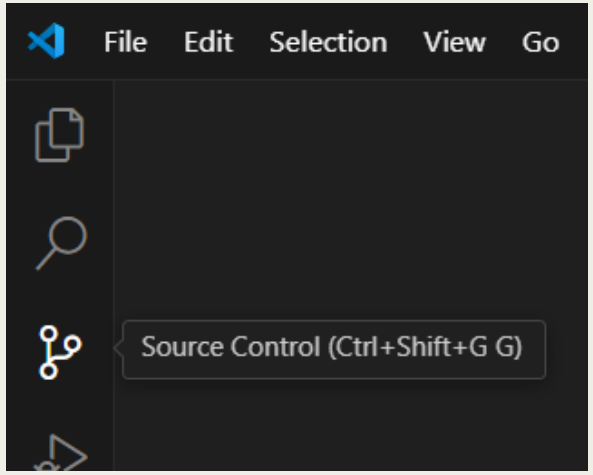


Example of version control

Version Control System

- A system that records changes to a file or set of files over time so that you can recall specific versions later.
- Categorized depending on how data is stored

- Version control allows..
  - Tracking of changes
  - History of changes
  - *(Collaboration between developers)*



Version Control is also known as Source Control

Local Version Control System

- The files are stored on your PC with timestamps.
  - Pretty standard for some programs
  - *Example*: RCS

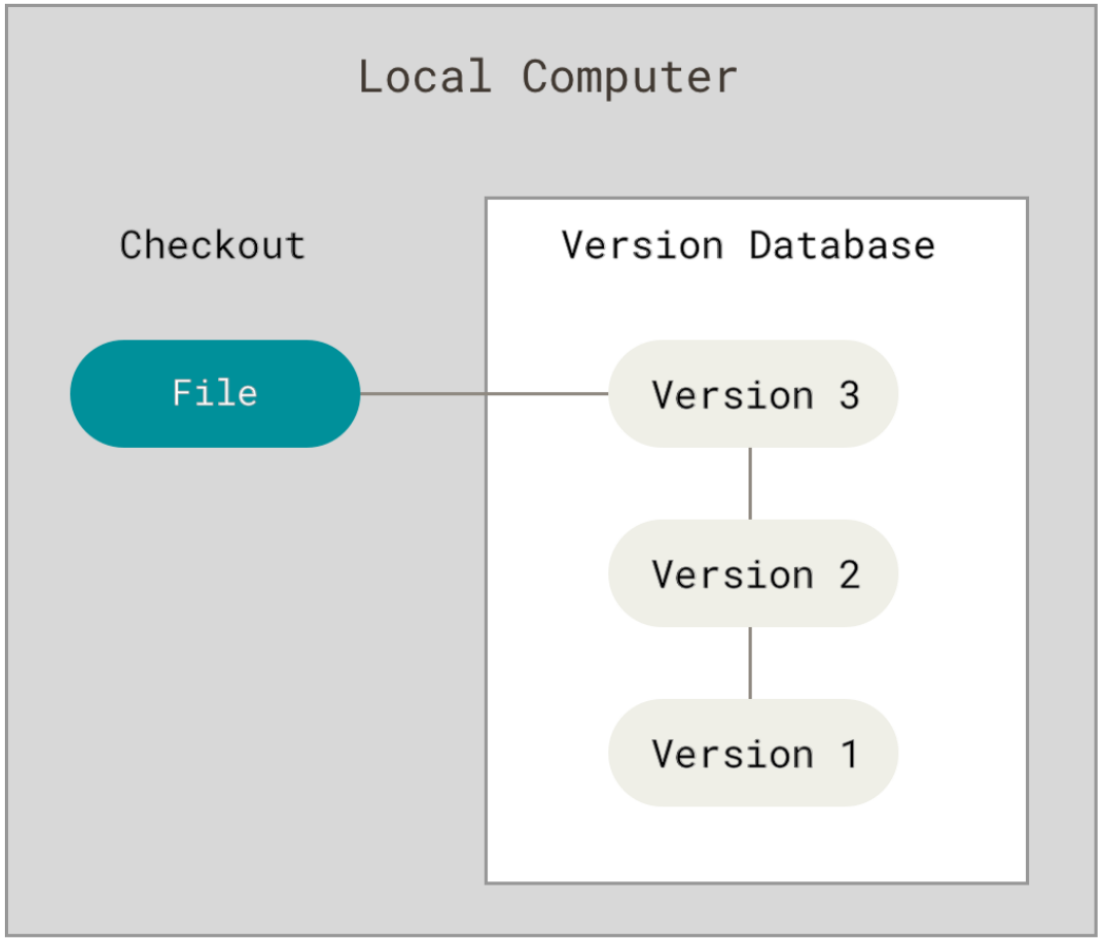


Figure 1. Local version control diagram

Centralized Version Control System

- All the versioned files are stored on a single server.
- Developers can pick and choose specific files to checkout and submit back into the repository.
  - *Examples:* CVS, Subversion, and Perforce

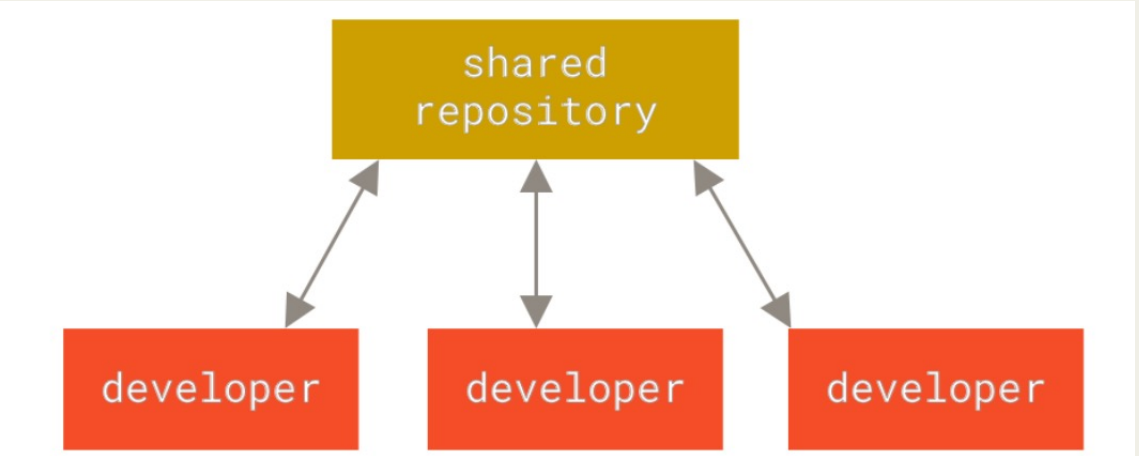


Figure 2. Centralized version control diagram

Distributed Version Control System

- Developers don't just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history
  - *Examples*: Git, Mercurial or Darcs

Logo for Git

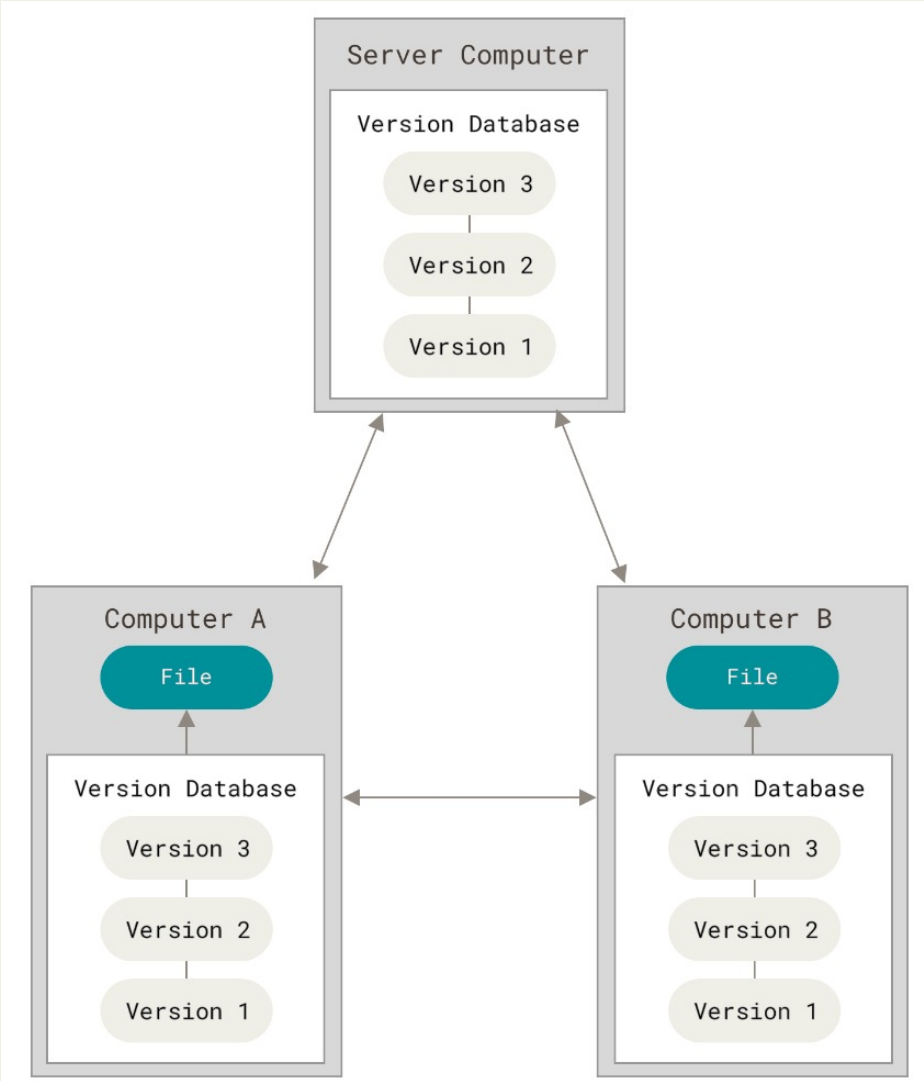
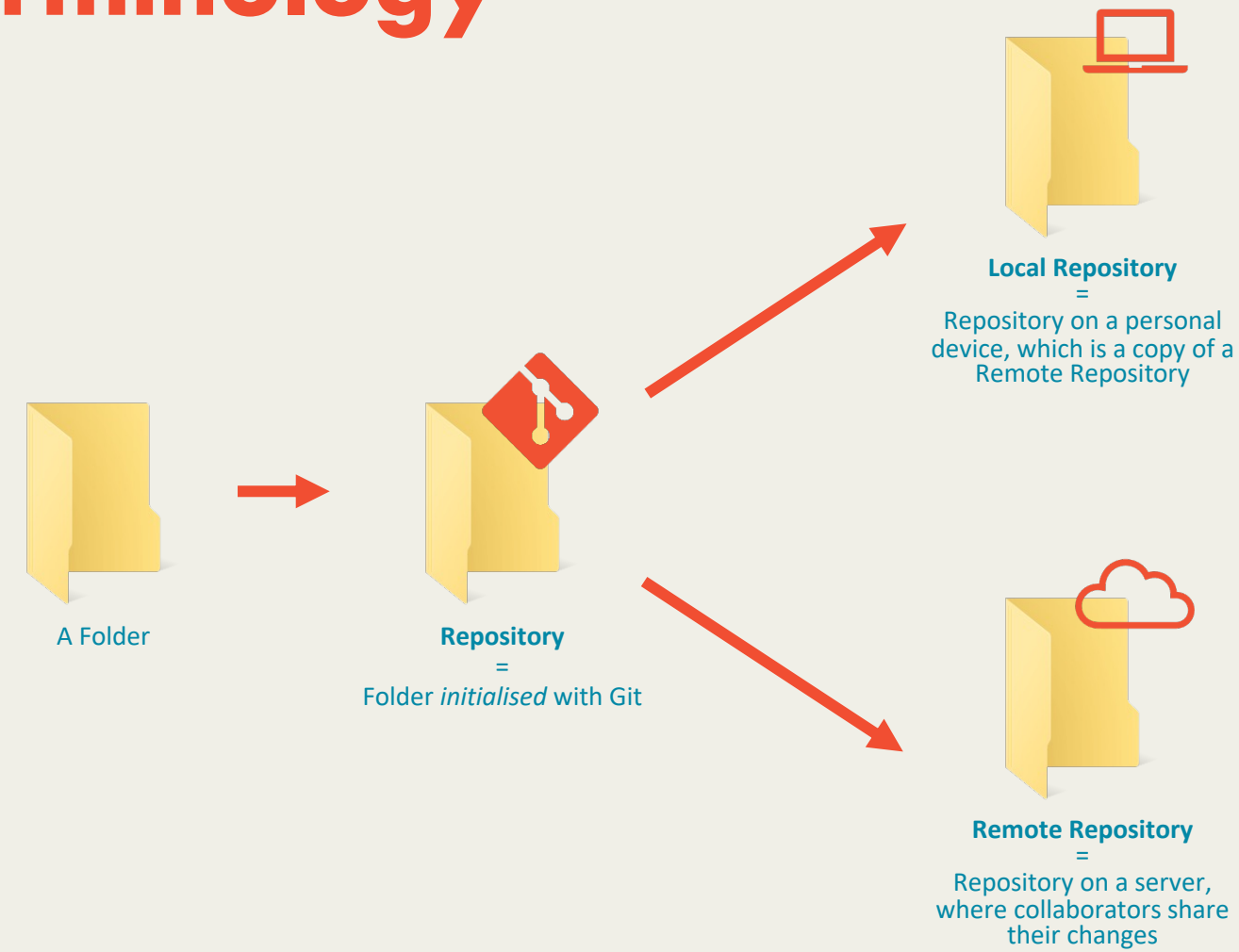


Figure 3. Distributed version control diagram

Back to Git

Let's just focus on that

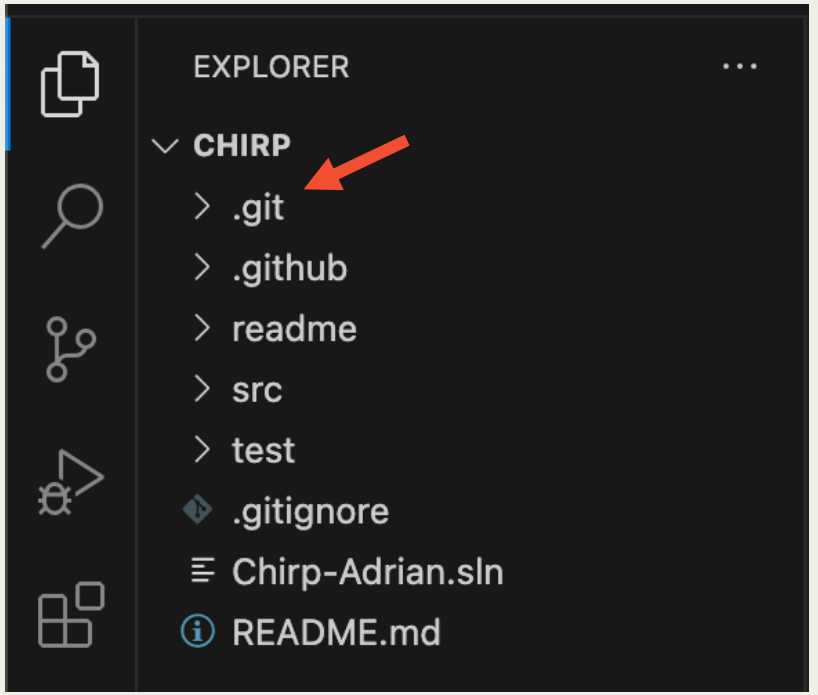
Some terminology



All of these are known as “projects”

What does "initialised with Git" mean?

- When a folder is initialised with Git, it means that Git has been set up to track changes in that directory.
- Initialisation creates a hidden subfolder (named *.git*) within the existing folder.
- The subfolder contains everything required for version control.
- All that you interact with and see related to Git, is within or via that subfolder.

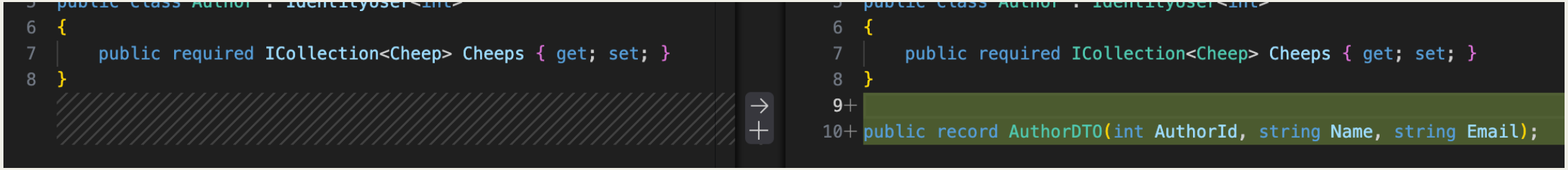



```
git init
```

Related terminal command

Changes

= Any addition, removal, or modification of anything within the repository.



Working tree showing addition of the record ***AuthorDTO***

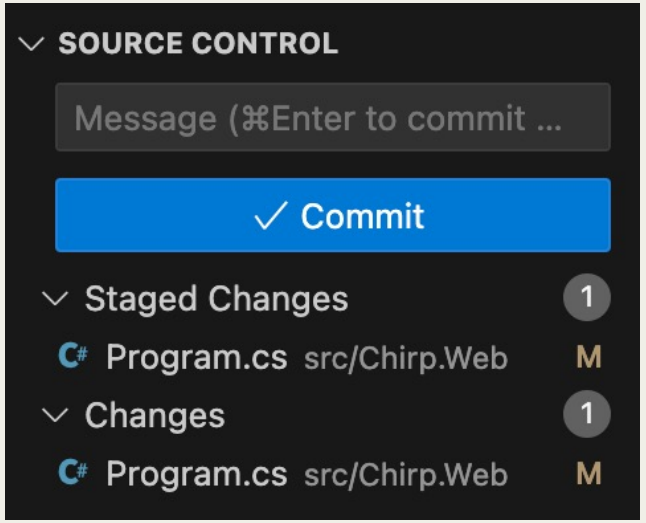
How to save changes

Changes are initially set as **unstaged changes**.

When you want to want to save a version of your changes, you can **stage the changes** (*git add*).

These changes are saved, and any changes you make to **files with staged changes** afterwards are regarded as **unstaged changes**. The staged versions are unaltered.

When making a **commit**, only **staged versions** of changes are included in the **commit**.



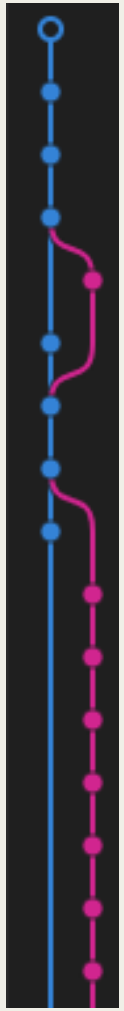

```
git add <file/directory>
```

```
git status
```

Related terminal commands

Commit

- To share your changes, you need to save your staged changes in a commit.
- A commit is a snapshot/version of the project saved by Git.
- A commit contains:
  - a set of changes.
  - one parent-commit
    - (Unless it is a merge commit)
- A commit can be shared to others from your local repository to the remote repository.
- A commit has a commit message, which should explain to other people what changes have been made.



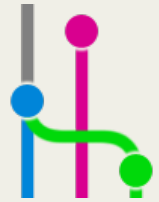
Commit history

```
git commit
```

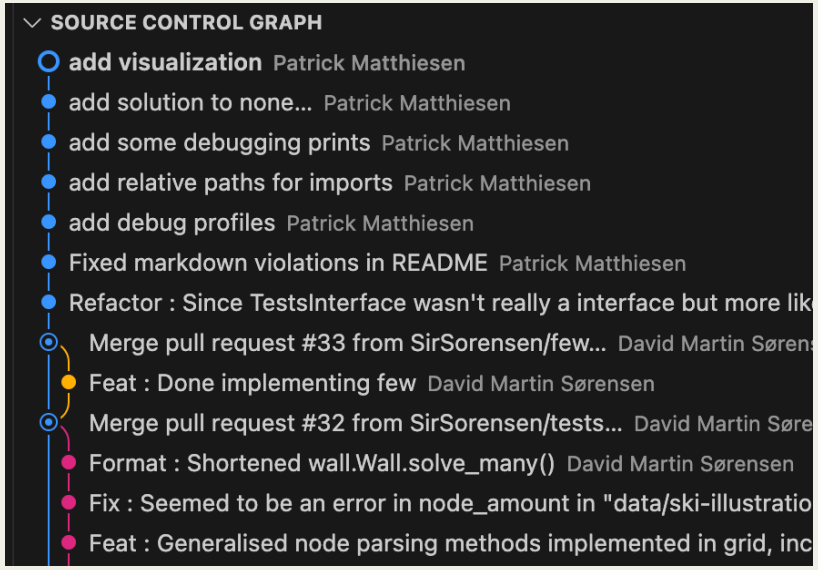
Related terminal command

Git Graph

- A git "graph" is a visualisation of the git history.
- I have used the Visual Studio Code extension "git graph"



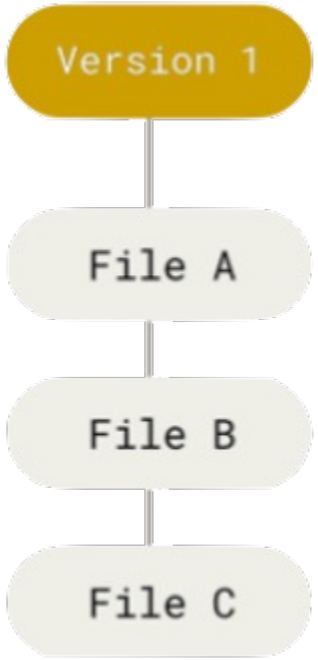
- It is now built into Visual Studio Code




```
git log --graph --all
```

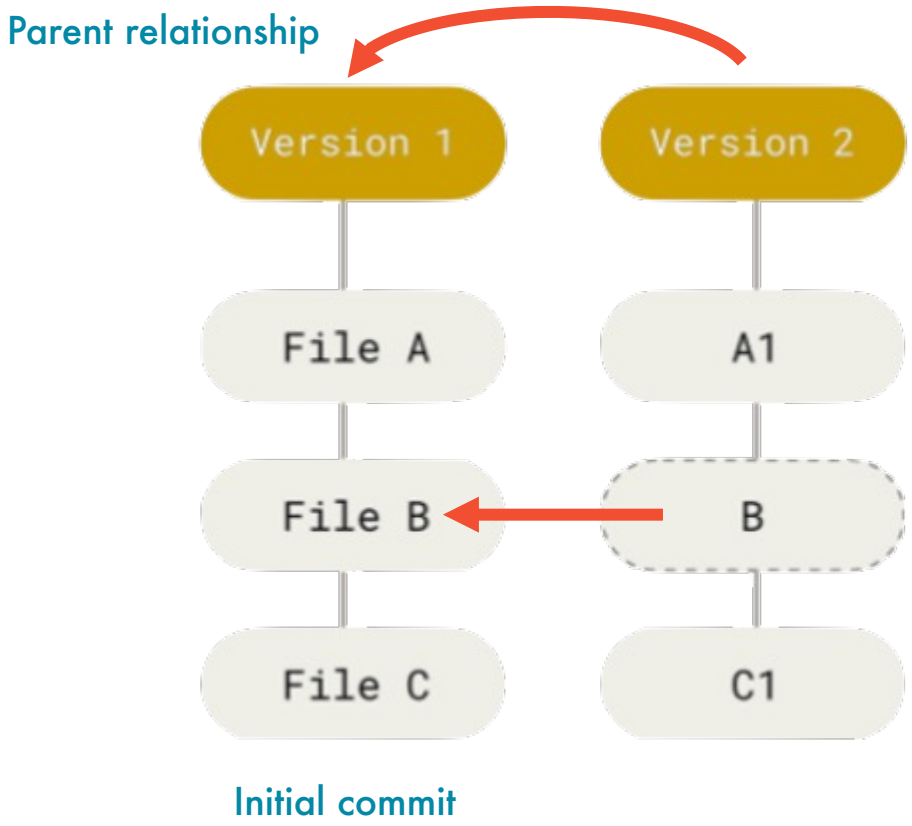
Related terminal command

What is contained in a commit?

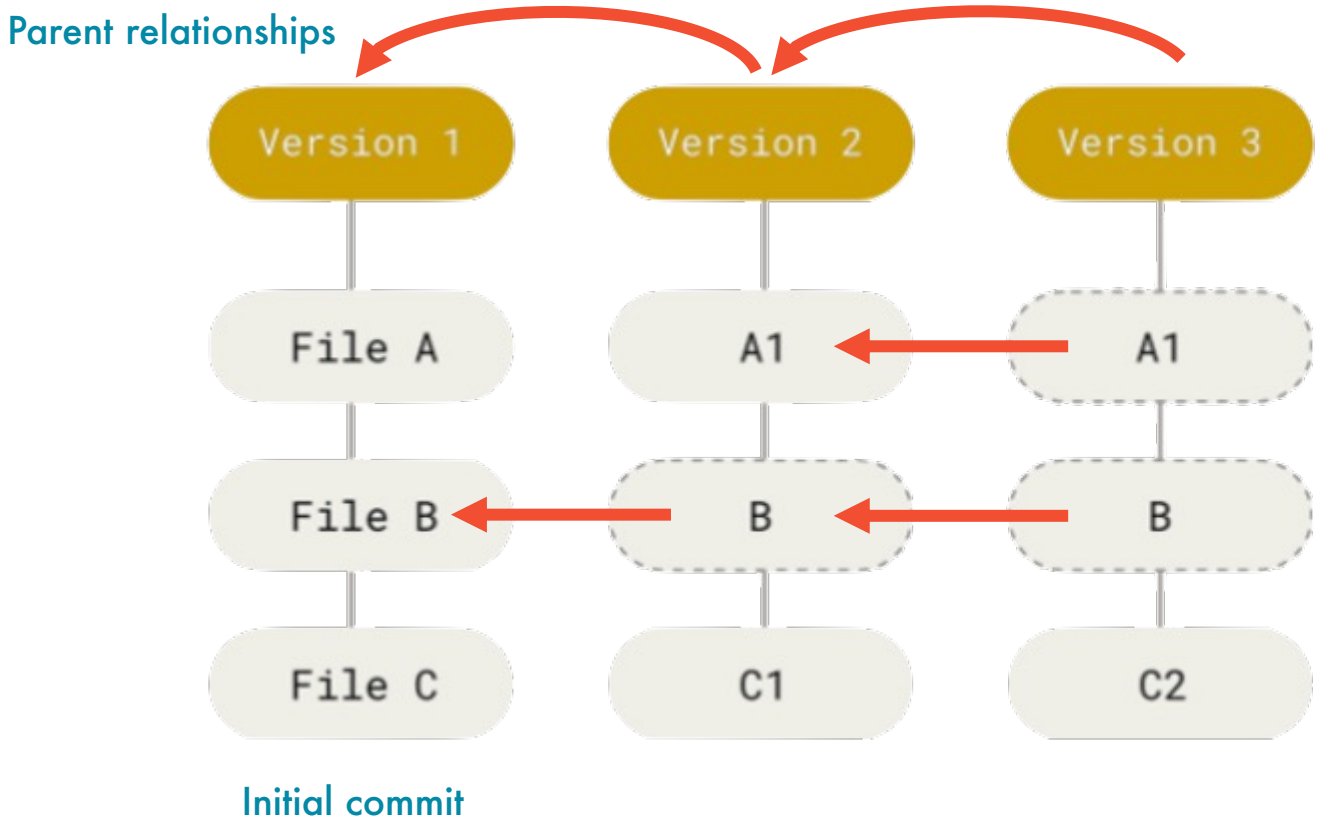


Initial commit

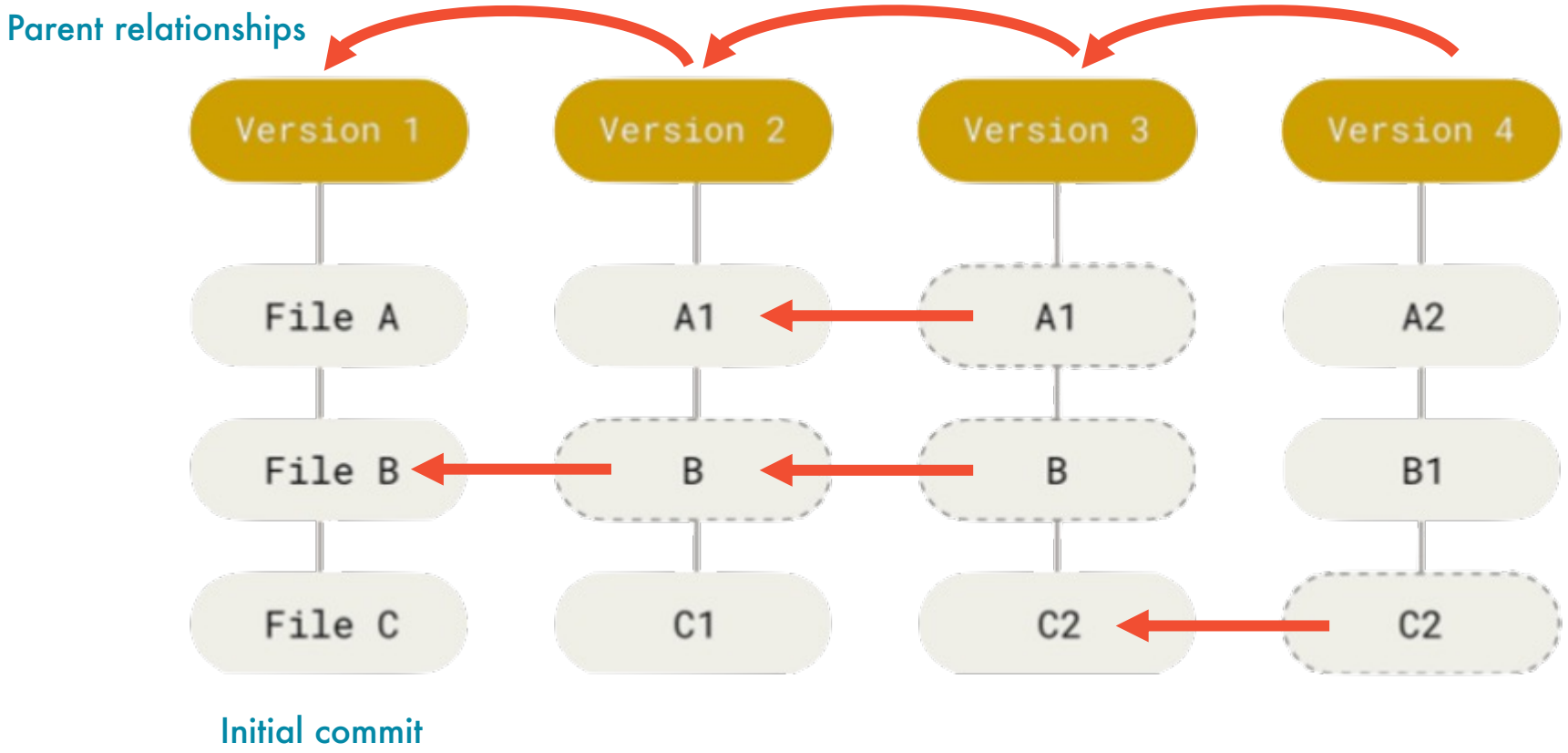
What is contained in a commit?



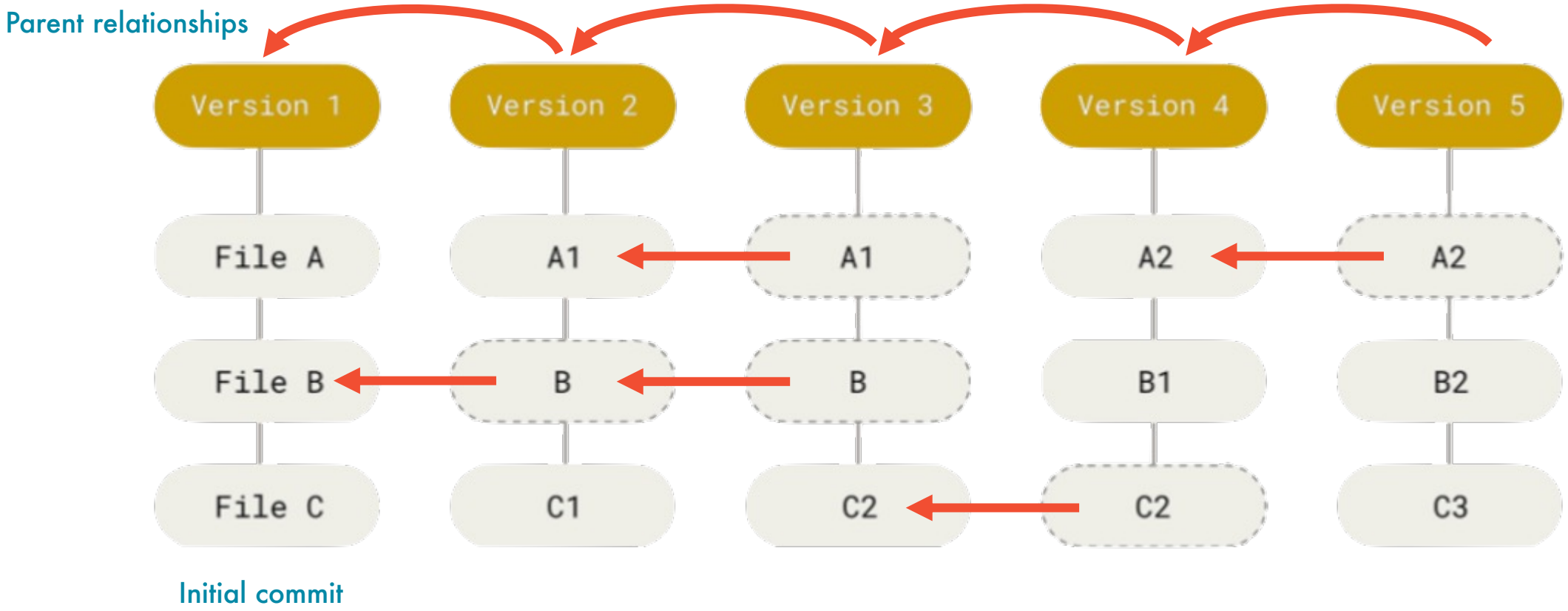
What is contained in a commit?



What is contained in a commit?

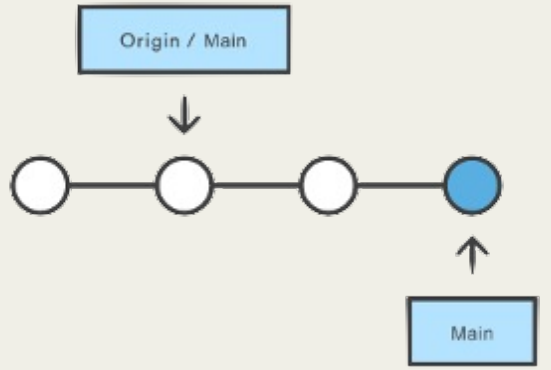


What is contained in a commit?

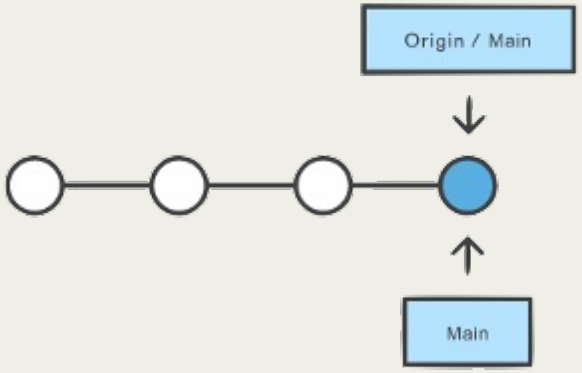


What happens during a push

- Save new commits on the remote repository
- Update the branch you pushed to
  - If pushed from “main” then you pushed to “origin/main”
- In this scenario, the history of the remote and the local is the same (pretty ideal)



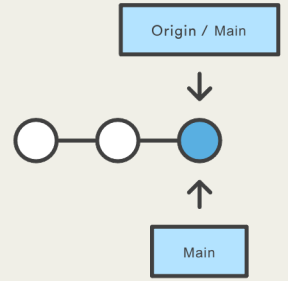
Before Pushing



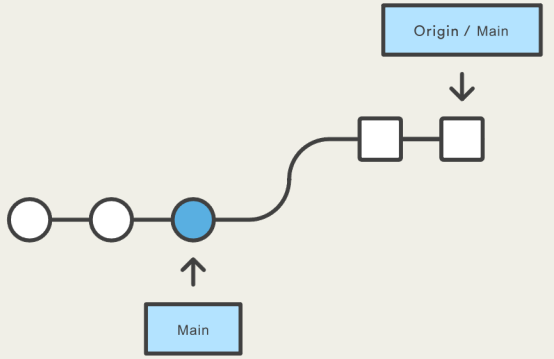
After Pushing

What happens during a pull

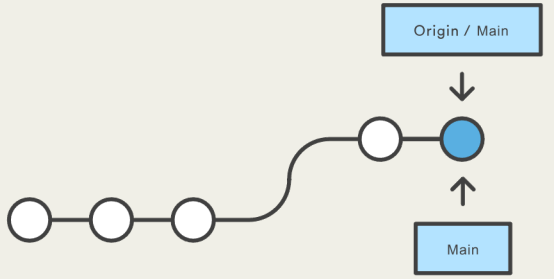
- Save new commits on the local repository
- Update the branch you pulled from
  - If pulled into “main” then you pulled from “origin/main”
- In this scenario, the history of the remote and the local is the same (pretty ideal)
  - This is known as fast forward
- (More on pulling later)



Before Pulling



After Fetching



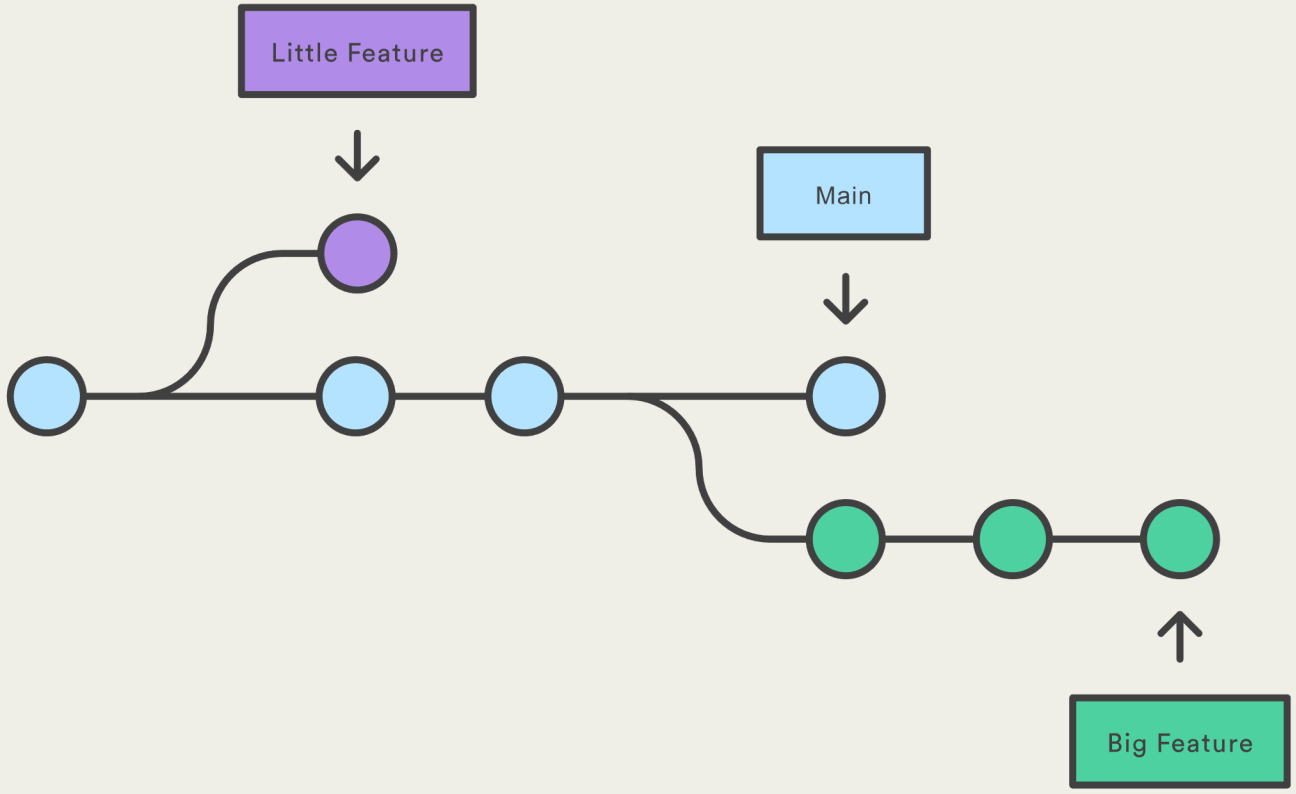
After Merging

Branches

Bladet på kvisten.
Kvisten på grenen.
Grenen på træet.
Træet på bjerget.

What is a branch?

- A branch is just a pointer
  - No seriously it's just a pointer
  - They have a name and a commit they point to.



What is a branch used for? (typically)

- Higher branches contain states of development
  - Prod, test, etc.
  - You should never develop directly on these branches
- Lower branches contain features (/ fixes)
  - The history of an isolated branch is that of a single feature
    - (It is up for a discussion what “a single feature” is)

```
git branch
```

Related terminal command

`git branch` commands

```
git branch
```

List all of the branches in your local repo.
(This is synonymous with `git branch --list`.)

```
git branch <b_name>
```

Create a new branch called `<b_name>`.
(This does not check out the new branch.)

```
git branch -d <b_name>
```

Delete the specified branch.
(Only executes if the branch has no unmerged changes.)

```
git branch -D <b_name>
```

Force delete the specified branch, even if it has unmerged changes.

```
git branch -a
```

List both remote-tracking branches and local branches.

Checkout & Merging

Jump around

Checkout

- You can checkout to a branch or a commit
  - Checking out to a commit is also known as “detached HEAD”
    - It just means that you are not working from a branch
- Change all working files to how it is in the specified branch or commit

```
git checkout <branch_name>
```

```
git checkout <commit_hash>
```

Related terminal commands

Git merge

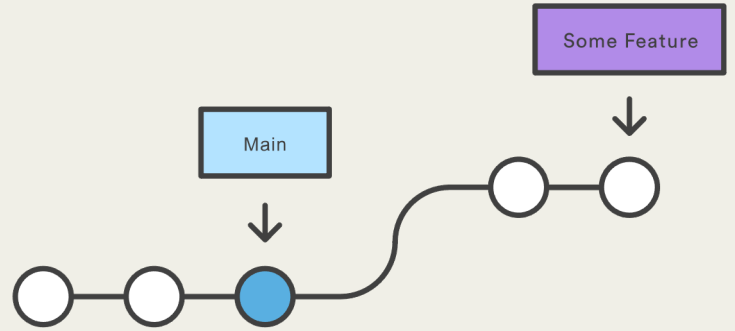
- Combines 2 branches into 1
- 2 types
  - Fast-forward
  - Non-fast-forward

```
git merge <branch_name>
```

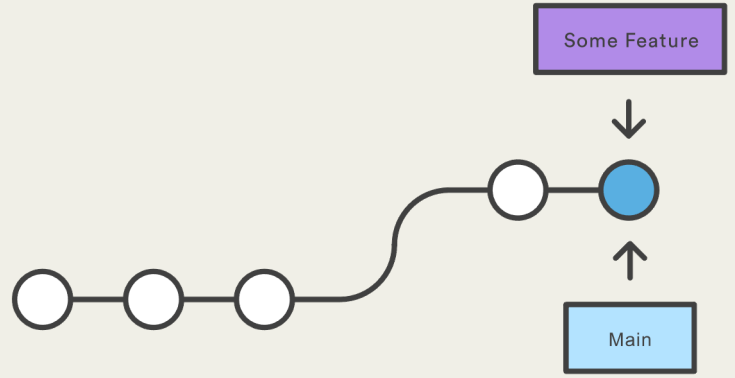
Related terminal command

Fast-forward

- A fast-forward merge can occur when there is a linear path from the current branch to the target branch.
- No merge commit is made
  - (Unless told to)



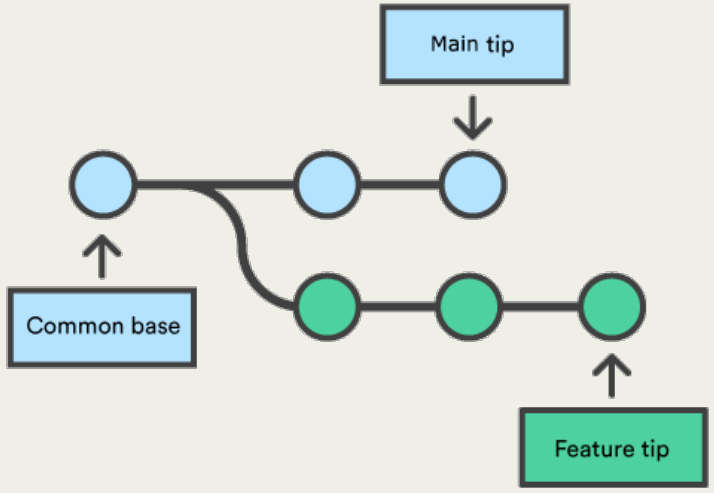
Before Merging



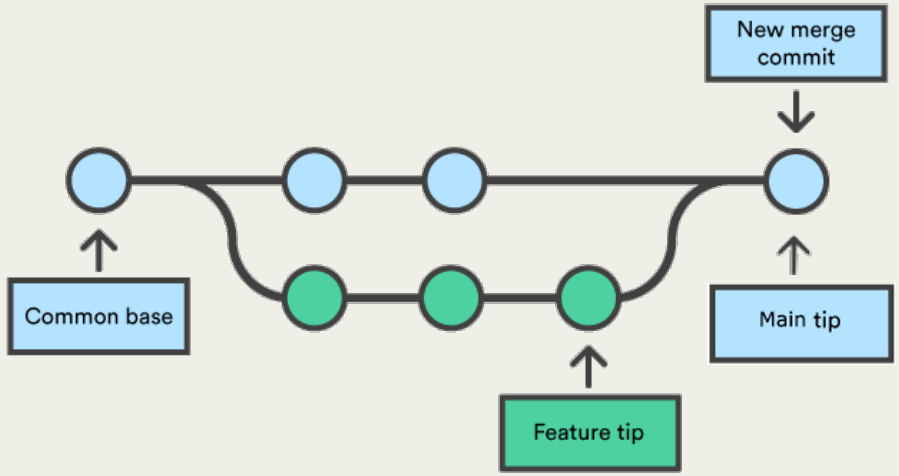
After a Fast-Forward-Merge

Non-Fast-forward

- Will occur when a fast-forward is not possible
- Manually resolve any merge conflicts
- A merge commit is made
  - A special commit that can have more than one parent
  - **Important note:** a merge commit is not only made if there is a merge conflict!



Before Merging



After a Non-Fast-Forward-Merge

Merge conflicts

- A merge conflict occurs when both branches to be merged have changed the same part of the same file.
  - Git does not know which is the correct version

- Git presents conflicts like this:

```
here is some content not affected by the conflict
<<<<<<< main
this is conflicted text from main
=======
this is conflicted text from feature branch
>>>>>>> feature branch;
```

Rewriting history

Quick note: Maybe don't

Full Desclaimer

I (David S.) am a rewriting git history-enjoyer.

- Can't stand non-fast-forward merges

We <u>don't</u> recommend that you don't do it in this course.

- It is (potentially) irreversible
- It can delete files and documentation permanently.

Good reasons to preserve history

Incremental history can be indispensable when

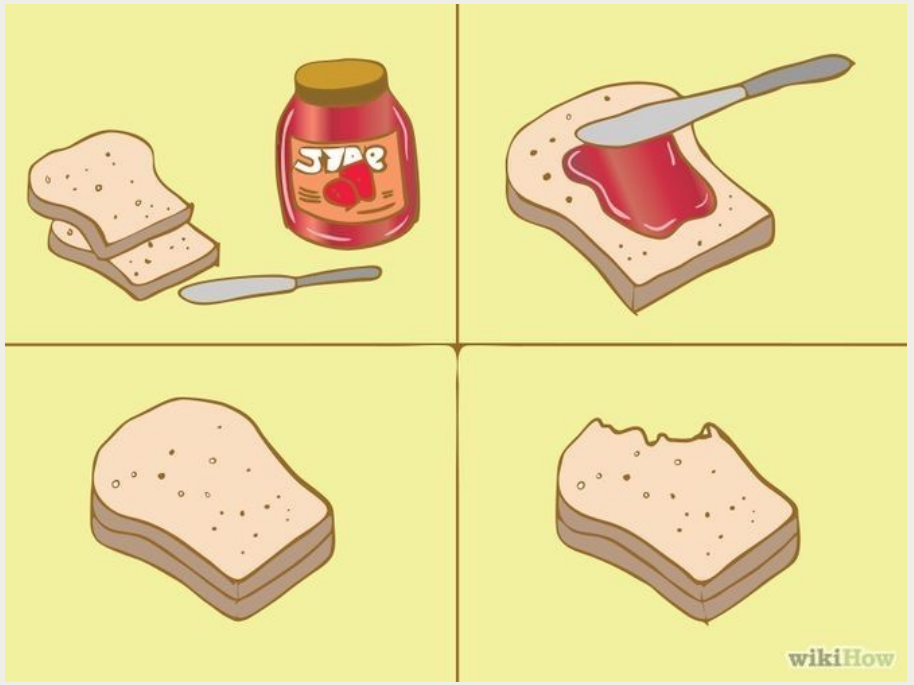
- Troubleshooting issues
- Identifying the motivations behind certain decisions
- Tracing prior development

```
(base) davidmsoerensen@Davids-MBP red-scare-group-i % git blame README.md
4b821676 (Patrick Matthiesen    2024-10-31 17:40:46 +0100  1) # Introduction
4b821676 (Patrick Matthiesen    2024-10-31 17:40:46 +0100  2)
9f6cf786 (David Martin Sørensen 2024-10-29 22:55:00 +0100  3) This repository is for the [Red Scare](https://le
arnit.itu.dk/mod/assign/view.php?id=208004) project for the course "Algorithm Design, MSc CS (Autumn 2024)" at
the IT University in Copenhagen.
9f6cf786 (David Martin Sørensen 2024-10-29 22:55:00 +0100  4)
4b821676 (Patrick Matthiesen    2024-10-31 17:40:46 +0100  5) ## How to run the program
9f6cf786 (David Martin Sørensen 2024-10-29 22:55:00 +0100  6)
```

Example output of `git blame README.md`

Why rewrite history then?

- Ensure clean & linear git history
  - Makes git history more readable/useable
- Ensure git history is ordered by feature
- Reduces merge conflicts (sort of)



Pulling *advanced*

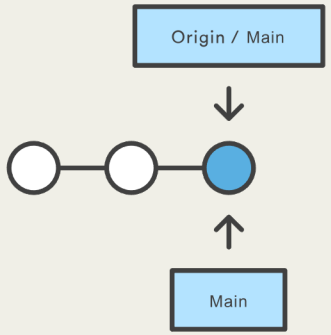
Hiv for helvede

What happens during a pull?

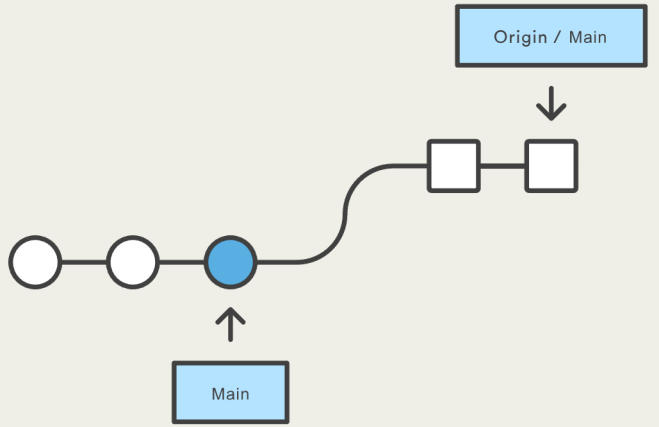
- `git pull` is a shortcut for `git fetch` and `git merge`

Pull: `git fetch`

- `git fetch` is a command that updates the local version of the remote version of your branch
  - Origin/main is the local version of the remote version
  - The remote version is not on your computer (it is on the remote)



Before Fetching



After Fetching

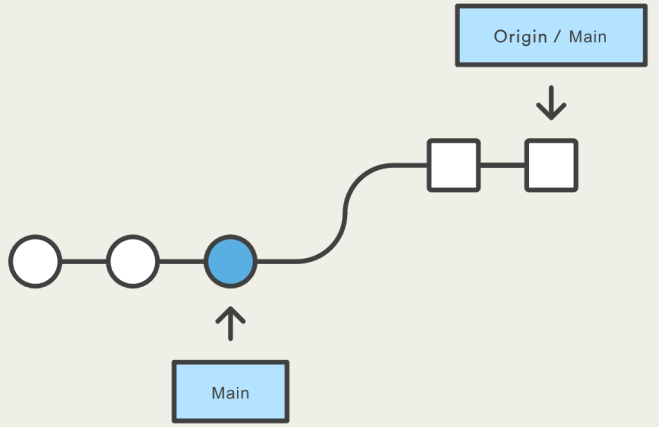
```
git fetch
```

```
git fetch --all
```

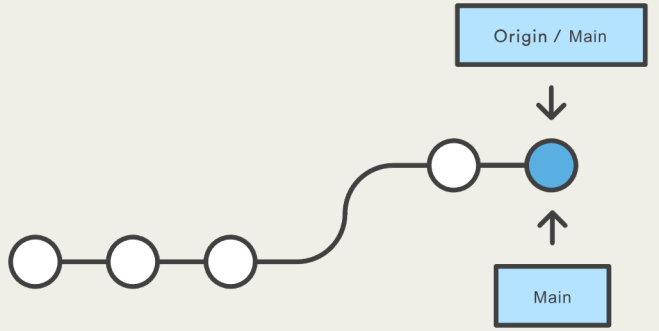
Related terminal commands

Pull: `git merge`

- Merge the local version of the remote version of your branch into your branch.
  - I.e. merge origin/main into main
- This merge can both be fast-forward and non-fast-forward
  - That's why you sometimes get merge conflicts when pulling



Before Merging



After a Fast-Forward-Merge

Stashing

Gamophobia: a fear of commitment

Ever tried to push before pulling?

- Git prevents you from pushing if it results in a non-fast-forward merge
  - For example:
    - When there a commits on the remote branch (origin/main) that you do not have in your local branch (main)
    - When you have rewritten the history (i.e. rebased)

Solution? `git stash`

- The `git stash` command takes your uncommitted changes (both/either staged and unstaged) and saves them away for later us.

```
git stash
```

- You can reapply previously stashed changes with `git stash apply`
  - The stash will still exist after this

```
git stash apply
```

- Alternatively, you can use `git stash pop` to reapply and delete the previously stashed changes.

```
git stash pop
```

- To delete a stash, simply use `git stash drop`

```
git stash drop
```

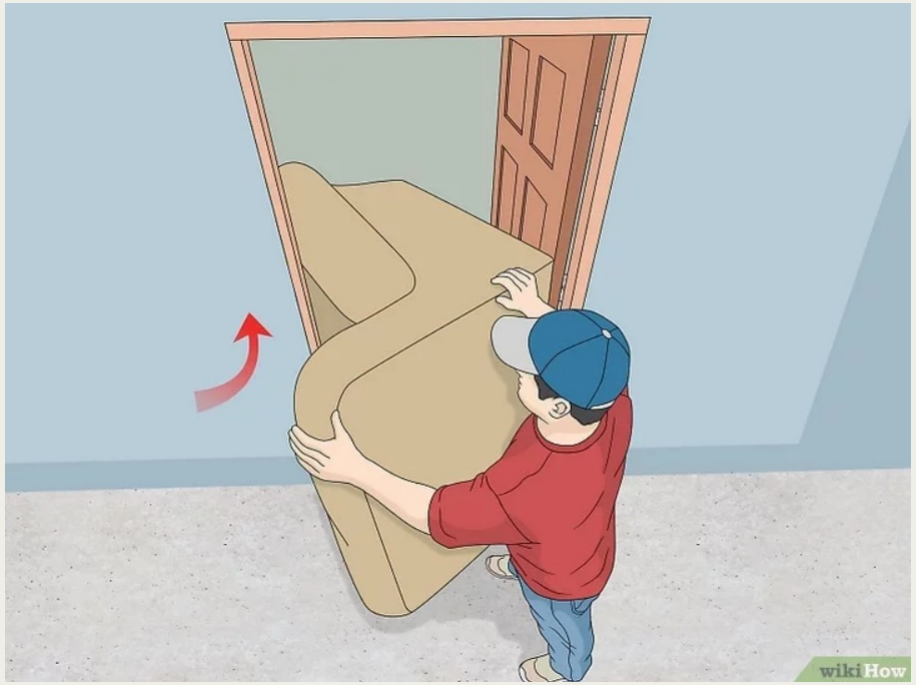
- Note: both `pop` and `apply` smushes the changes together so that nothing is staged, unless you add `--index`

```
git stash apply --index
```

```
git stash pop --index
```

Pushing *advanced*

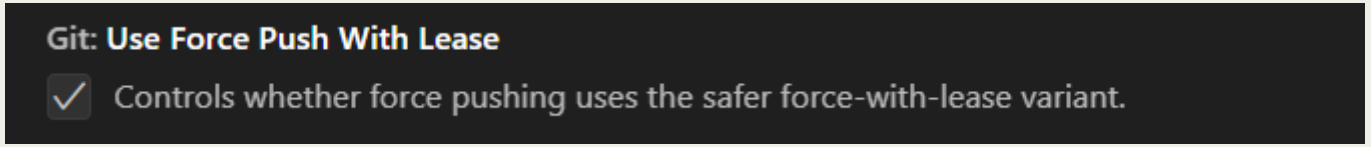
Skub, skub, skub til taget



Want to push anyways? Push *harder*

`git push --force`

- Overwrites the remote with your local version
- Ignores errors
- Variant: `--force-with-lease`
  - Does the same thing but stops and raises an error if there are new commits on the remote
  - Considered much safer
  - Is set as default for force pushing in many application (i.e. VSCode)



When should you force push?

- Force pushing is only necessary in instances when you can't fast-forward (i.e. when you have to rewrite history)
  - `rebase`
  - `reset`

```
git push --force-with-lease
```

```
git push --force
```

Related terminal commands

Undoing commits

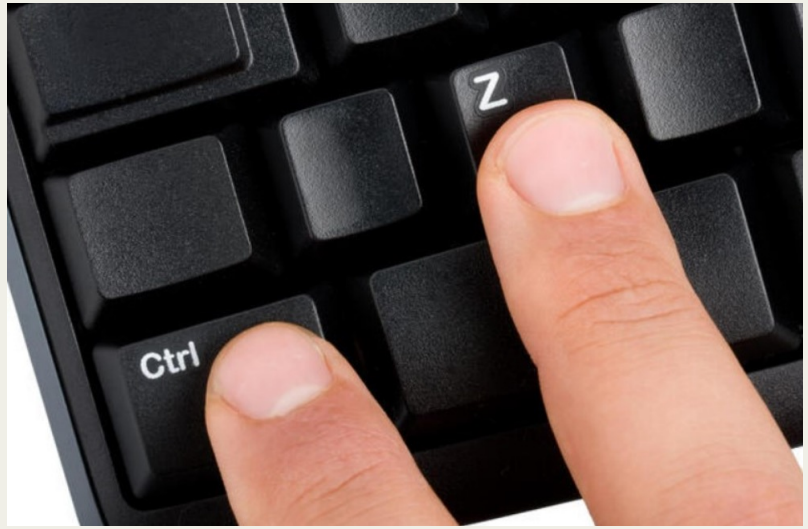
Many ways of doing it!

Typical ways

- `git revert`
- `git reset`

git revert

- Make a new commit which undos the other commit
  - Anything added is removed, anything removed is added, etc.




```
git revert <branch_name>
```

```
git revert <commit_hash>
```

Related terminal commands

git reset

- Set the pointer of a branch to somewhere else
- Often needs to push force afterwards
- Comes in 3 flavours
  - `--hard`
    - Reset branch-pointer and discard all changes of later commits
  - `--mixed` (default)
    - Reset branch-pointer and put all changes of later commit into unstaged commits
  - `--soft`
    - Reset branch-pointer and put all changes of later commit into staged commits

```
git reset <commit_hash>
```

Related terminal command

When do I use what?

- `git revert` if I want to undo changes commits
- `git reset --hard` as a panic button when rebasing, merging, etc.

Cleaning

Zug zug

Want to delete all the stuff?

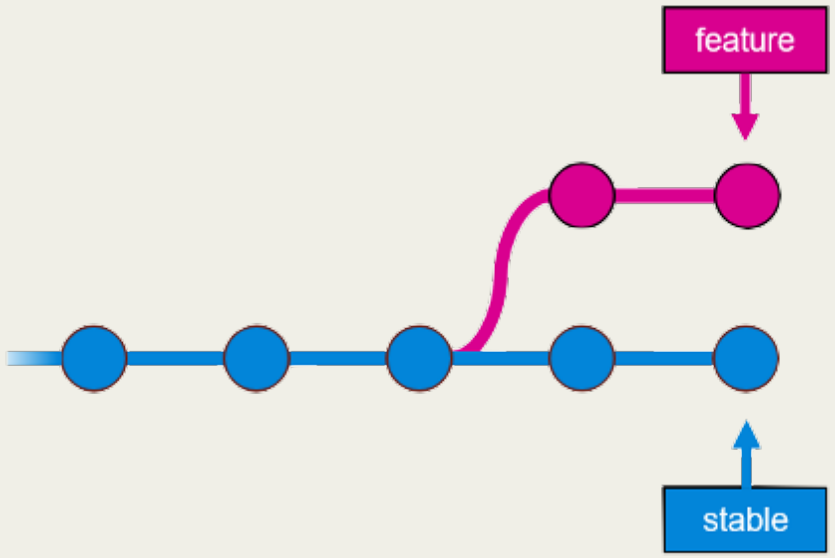
- Git does not remove untracked files when pulling.
- If you want to delete all untracked directories and files, and ignores files: `git clean -dfx`
- If you want to delete all unstaged changes: `git restore .`
- If you want to delete all staged and unstaged changes: `git reset --hard HEAD`

Rebase

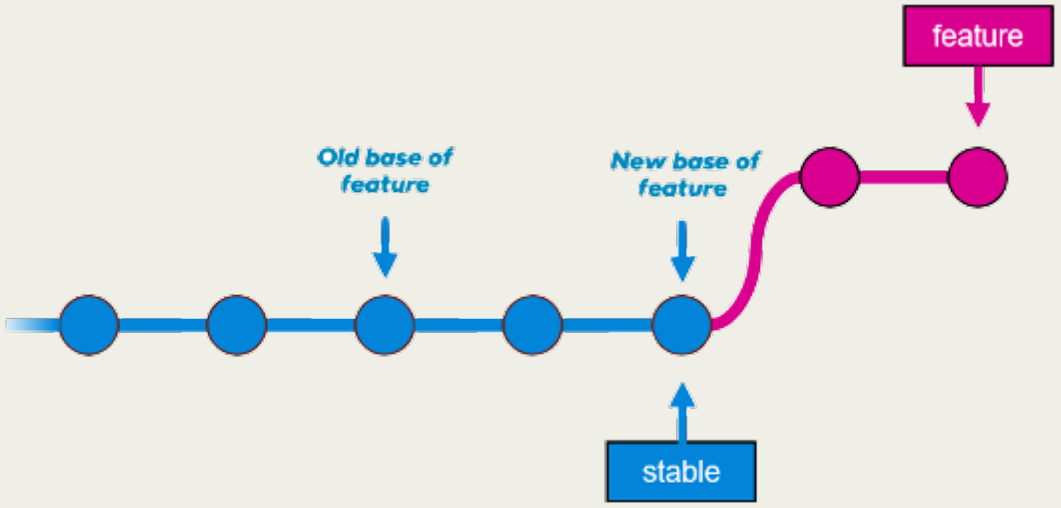
For educational purposes

Rebasing in a nutshell

- Rewrite history by making commits of a branch come after base commit and its predecessors

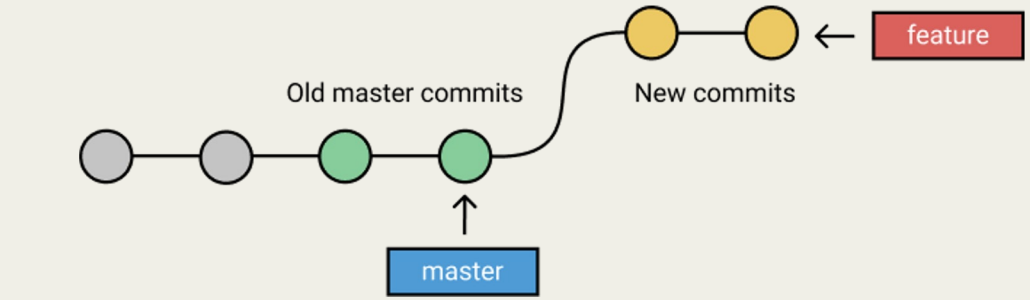


Before rebase

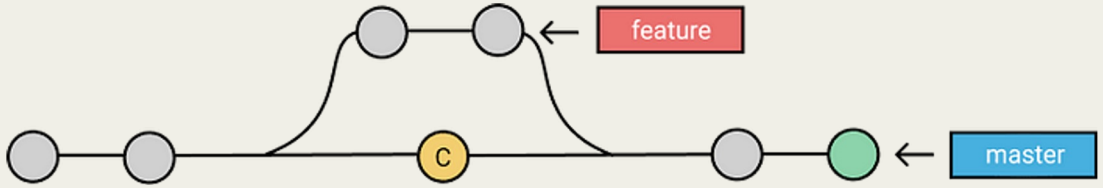


After rebase

Rebasing ensures linear git history



Linear



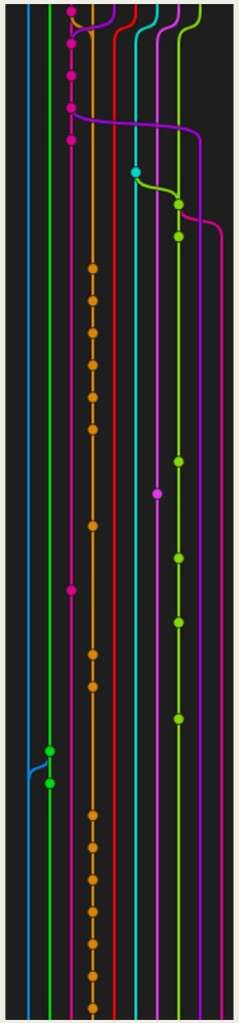
Non-linear

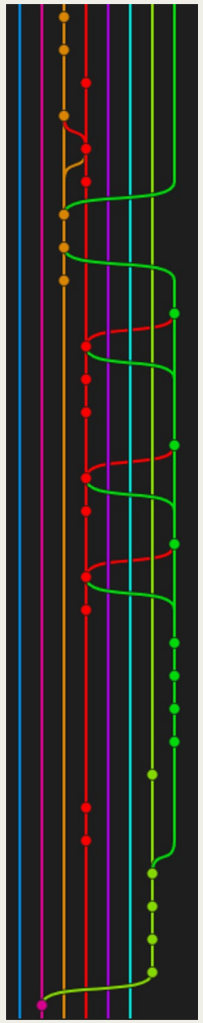
"A linear Git history is like a well-maintained highway, a direct route from point A to point B with no confusing side roads or turn-offs.

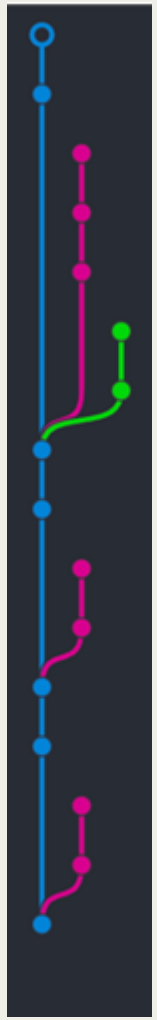
Every commit directly precedes the commit that follows it, and there's no overlapping or intertwining of different versions.

Contrast this with a non-linear history, where you may have multiple developers creating separate branches, each with its own commits."

Make your git graph readable







With that said, don’t

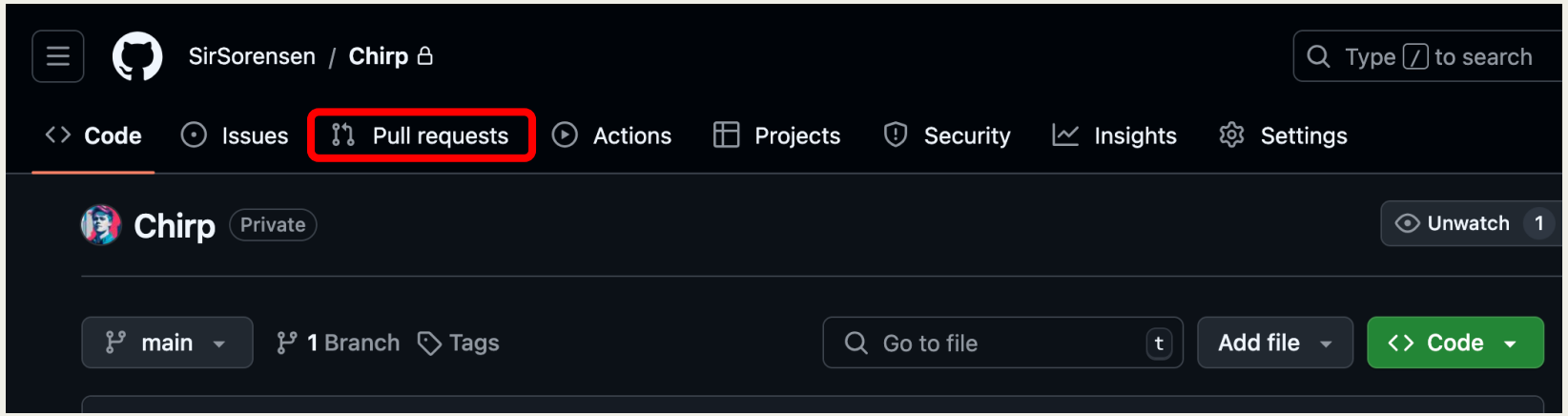
- Rebasing isn’t always easy
  - Especially not if you have merged into this branch from another branch during the lifetime of this branch
- It requires a great overview of changes in both source and target branches

Pull requests

Merge into stable but better

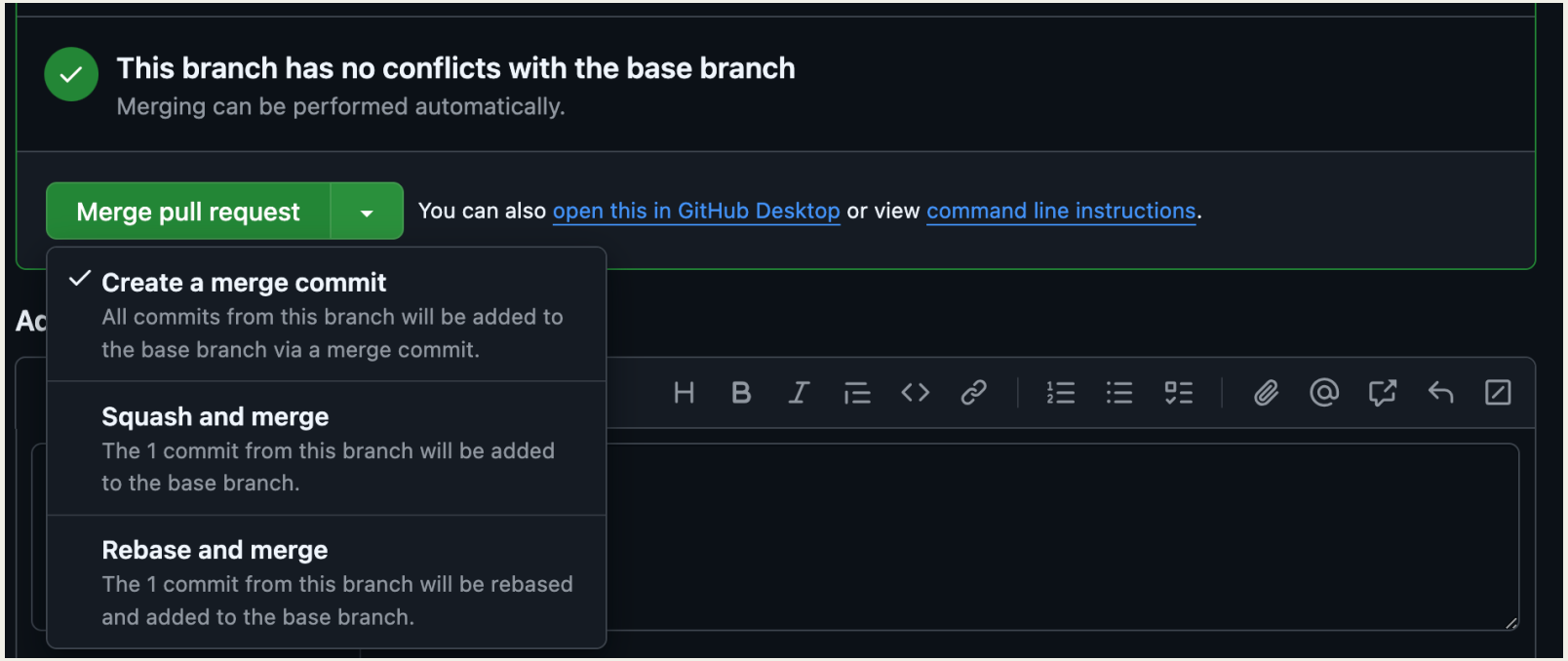
Pull request?

- Something you do on the *GitHub* website
- You can configure tests for a pull request (Actions/Workflows)
- This is (often) where review happens
  - Stops developers from smashing unreviewed work into production



Types of results

- When concluding a pull request, you can choose between



  - Squah commit = collect all commits of the branch into one
    - Is also kind of rewriting history

Use scenario (ideal)

- A feature is done (ready for production perhaps) and the branch should therefore be merged into main
- A developer creates a pull-request for the feature-branch
- The rest of the team reviews the code, discusses it, and alters it.
  - Possible by commenting and sending it back to the requester
- Another developer merges the feature into the official repository and closes the pull request.

Best practices!

Ideal is not reality

Commit messages (Conventional commits)

- A standardized format for writing commit messages
  - ```
    <type>[optional scope]: <description>

    [optional body]

    [optional footer]
    ```
  - **Feat** : [commit message]
    - A new feature
  - **Fix** : [commit message]
    - A bug fix
  - **Refactor** : [commit message]
    - Code change that doesn't fix a bug or add a feature
  - **Perf** : [commit message]
    - Improve performance
  - **Style** : [commit message]
    - Changes that do not affect meaning (e.g., whitespace, formatting)
  - **Docs** : [commit message]
    - Documentation-only changes (i.e. documents, readmes or comments)
  - **Test** : [commit message]
    - Adding or updating tests
  - **Chore** : [commit message]
    - Miscellaneous tasks, like updating dependencies.
  - **Revert** : [commit message]
    - A commit of revert simply reverts changes made in another commit. (Personally, I think it's better to revert a commit using git.)

Read more here
    ```

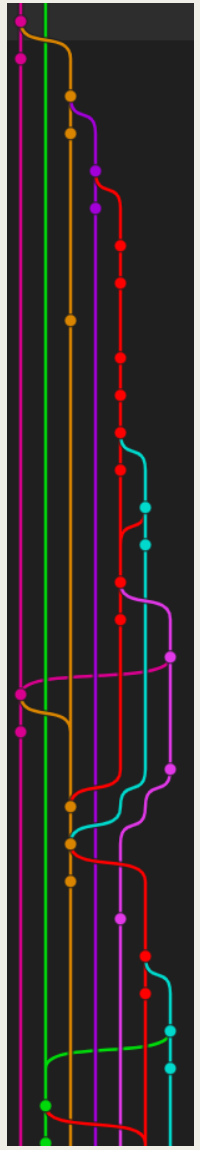
Commit messages part 2

- Coauthors
- Assume the reader does not understand what the commit is addressing.
- Why was this commit necessary?
- Be clear and concise
  - However, it can (and sometimes should) be multiple lines
- If you want to write multiple things, then make multiple commits
- It takes practice

1. `git commit -m 'Add margin'`
2. `git commit -m 'Add margin to nav items to prevent them from overlapping the logo'`

Branches

- Branch frequently
  - Make a branch per feature
- If you can work on multiple branches, then do it.
- Make pull requests into main instead of just merging
- Delete old branches
- Branch names should follow best practices
  - feature-[feature title]
  - bugfix-[bug title]



This is 1 branch 😱

Misc.

- Commits should be small but not microscopic
  - A commit does not have to be a fully done product, but should not break the code either
- Do what you can to merge with fast-forward
  - Pull changes before starting work
  - (Auto)stash