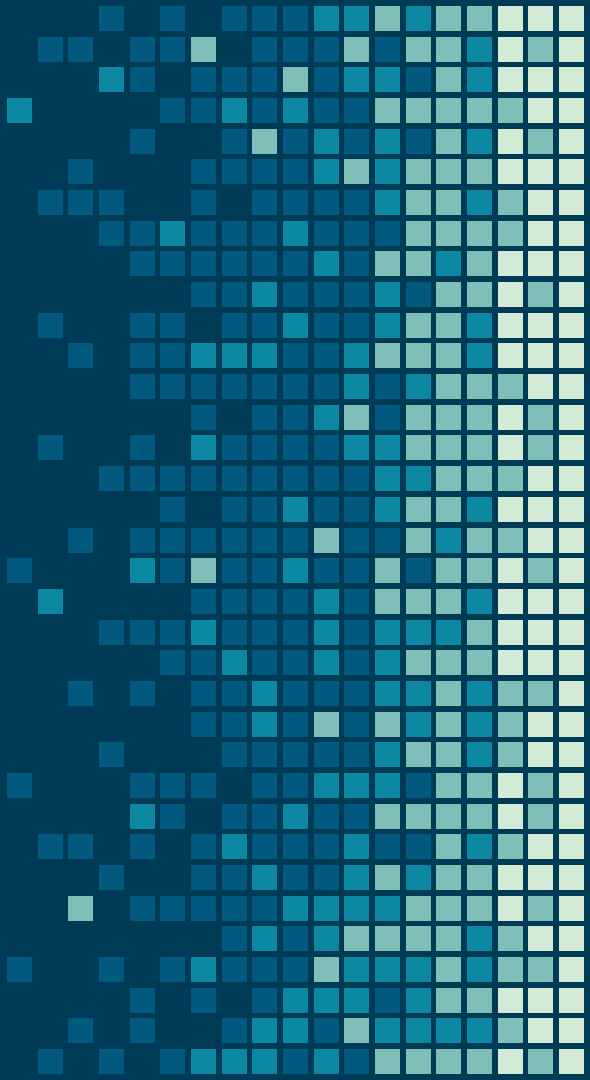
An Introduction to Git for Scientists

Dr. Tim Schäfer
ESI for Neuroscience
Fries Lab Academy
March 06, 2023



About this Presentation

- The title is *Git for Scientists*: the first part explains how Git can help scientists, but the rest is not specific to science.
- Keep in mind that git is a very complex tool for distributed software development, and you do not need all of its features.
- The recommendations given come from me (Tim) and are *my personal opinion*. As coding platform, I primarily use and thus focus on GitHub. Feel free to use another one, they all work.
- This is a presentation, not a hands-on git workshop. The focus is on concepts, not git commands.
- The git CLI can do everything and is shown here, use a GUI or the git integration of your favourite IDE later if you prefer.

Overview

- Part 1: What version control can do for science
  - Versioning, Collaboration, Reproducibility
- Part 2: Git basics: Local git
  - Repositories, commits, diffs, conflicts
- Part 3: Distributed git
  - Platforms, Distributed Git Workflow, Branches, Conflicts
- Part 4: Suggestions for getting started with git
  - Git, GitHub, IDE integration

1.
Science and version control

A step towards reproducibility

Version Control

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

Version Control was developed to:

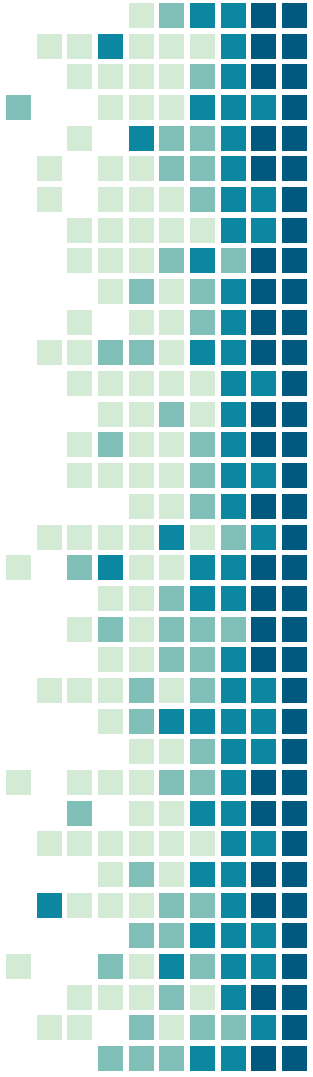
- Manage and access different versions of files
  - Typically source code, but it can be anything.
- Work on a complex project with several people
  - … and solve the resulting versioning conflicts that will occur.

- Previous solutions:
  - Manually create many copies of your project directory:
  - Copy or send files back and forth, e.g., via email,
    and compare them manually or via diff/patch tools.

```
my_projects/
  projectA_2023_02_02/
  projectA_2023_02_05/
  projectB_2023_02_12/
  projectA_2023_03_11/
```

Version Control can solve the following problems for you

- You messed up your Matlab script today, and you want to recover yesterday's version.
- You need to know the settings you used to compute the data for paper *XY*, but you changed the script in the meantime.
- You received a new version of a script that circulates in the lab from a colleague, and you want to see the changes compared to your older version.
- You are improving a script over time in an iterative fashion, and you want colleagues to be able to always access the latest version, without sending around emails after every change.
- Both you and a colleague made changes to different parts of a script, and you want to merge your changes into a new version.
- You want to collaborate with colleagues when writing a script or software, and allow them to see your code, and directly make or suggest changes.



What Version Control can do for you

- Restore last/earlier version
- Find changes compared to other version
- Find version used for paper XY
- Always make latest stable version available to colleagues
- Collaborate on a project

Version Control to improve Reproducibility in Science

- Replication Crisis
- Open Science: first step to improve replicability is to improve reproducibility
- Reproducibility requires access to:
  a. Dataset
  b. Data analysis pipeline
  c. Software Environment to run the Pipeline
- Versioning and Version Control are part of this
  a. Public data repositories, Free Software, Docker / Singularity

Social Coding Platform (SCP)* benefits for Scientists

*GitHub, Sourceforge, Gitlab, BitBucket, ...

- Note: SCPs are build around VC, they act as a VC server.
- They are social
  - Scientific collaboration
  - Networking, visibility for your projects, jobsearch
- Easy integration with other services
  - E.g., publish a Python package from your Git repository, host project website, CI, get a DOI for a software version, build and publish documentation, ...
- Side effect: Synchronize work across devices
  - What about backup?
- Note: SCPs often offer free professional accounts for scientists.

2.

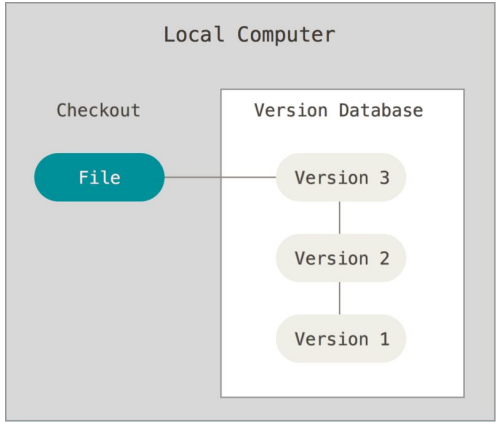
VCS and Git basics

Git concepts and vocabulary

Version Control System (VCS)

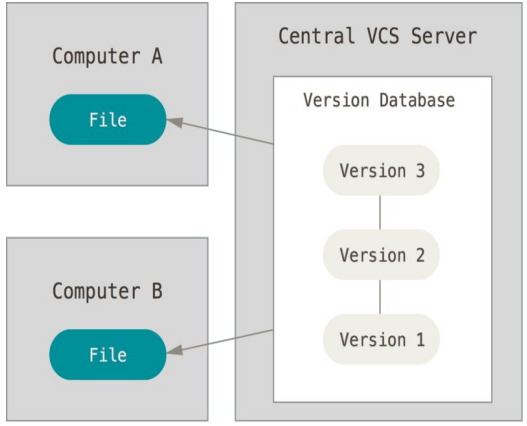
Local

E.g., Revision Control System (RCS)



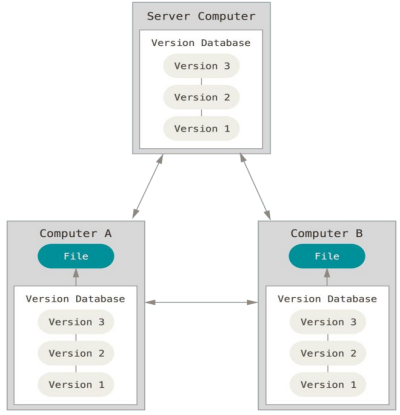
Centralized

Subversion, Concurrent Versions System (CVS)



Distributed

Mercurial, Bazaar, Darcs, Git



Distributed VCS and Servers

- Servers are optional for distributed VCS systems, but almost always used in practice
  - Team work, Access management, Backup, Integration with other services, …
  - Technically all repos are equal and the ‘authoritative’ server/repo is just a convention.
- If a Server exists:
  - Stand-alone (on-premises, private SaaS / cloud instance)
  - Social Coding Platform (GitHub, GitLab, BitBucket, Sourceforge, …)

Git

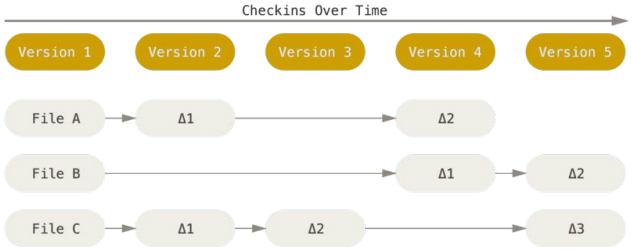
- widely used
- speed, distributed workflows
- Linus Torvalds, 2005, Linux
- Open Source (GPL)
- Data integrity
- Secure against data loss
- Powerful & complex
  - See, e.g.: De Rosso, S.P., and Jackson, D. "Purposes, Concepts, Misfits, and a Redesign of Git." ACM Press, 2016. 292–310.

Git

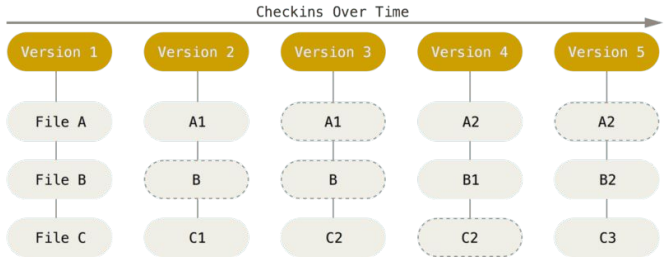
- widely used
- speed, distributed workflows
- Linus Torvalds, 2005, Linux
- Open Source (GPL)
- Data integrity
- Secure against data loss
- Powerful & complex
  - See, e.g.: De Rosso, S.P., and Jackson, D. "Purposes, Concepts, Misfits, and a Redesign of Git." ACM Press, 2016. 292–310.

What is special about Git?

- Each copy is the full repo
  - Most actions are local and thus fast
  - It works offline
- Safe
  - Data integrity via checksums
  - Designed to add data, not delete it (but it's possible if absolutely needed)
- Filesystem-like approach: Snapshots instead of Deltas



Delta approach



Snapshot approach

Git first time setup

- Notes:
  - Command line interface (CLI) versus IDE / graphical UI
  - **Concepts, not commands**
  - git CLI follows convention: `git <git_command> <cmd_options>`

    `git help <git_command>`

Git first time setup

- Git Installation: pre-installed on many systems, otherwise download from `https://git-scm.com`

- Commits have an author
  - `git config user.name # Show it.`
  - `git config --global user.name "Jane Doe" # Set it.`
  - `git config --global user.email "me+code@institute.org"`

- Many more `git config` options are available, e.g., Credentials storage. Check the documentation: `git help config`

Getting a git repository – Two Options

Turn your existing project folder into a Git repository

```
cd myproject/
git init
```

Clone a remote repository (e.g., after creating an empty one on the Github website)

```
git clone https://github.com/myuser/myproject
cd myproject/
```

- Creates subdirectory `.git` and other special files, e.g., `.gitignore`

Saving files to version control: commit

And working alone on a project on a single computer

```
cd myproject
git add *.py subdir1 subdir2
git commit -m "add my project"

# Now change some files
git status
git add changed_file1.py file2.m
git add file3 dir2/
git commit -m "add feature XY"

# Repeat.
```

Important: *With git, users decide which files and changes are added to the database, and under which label (commit message). It does not happen automatically.*

Saving files to the database is known as committing.

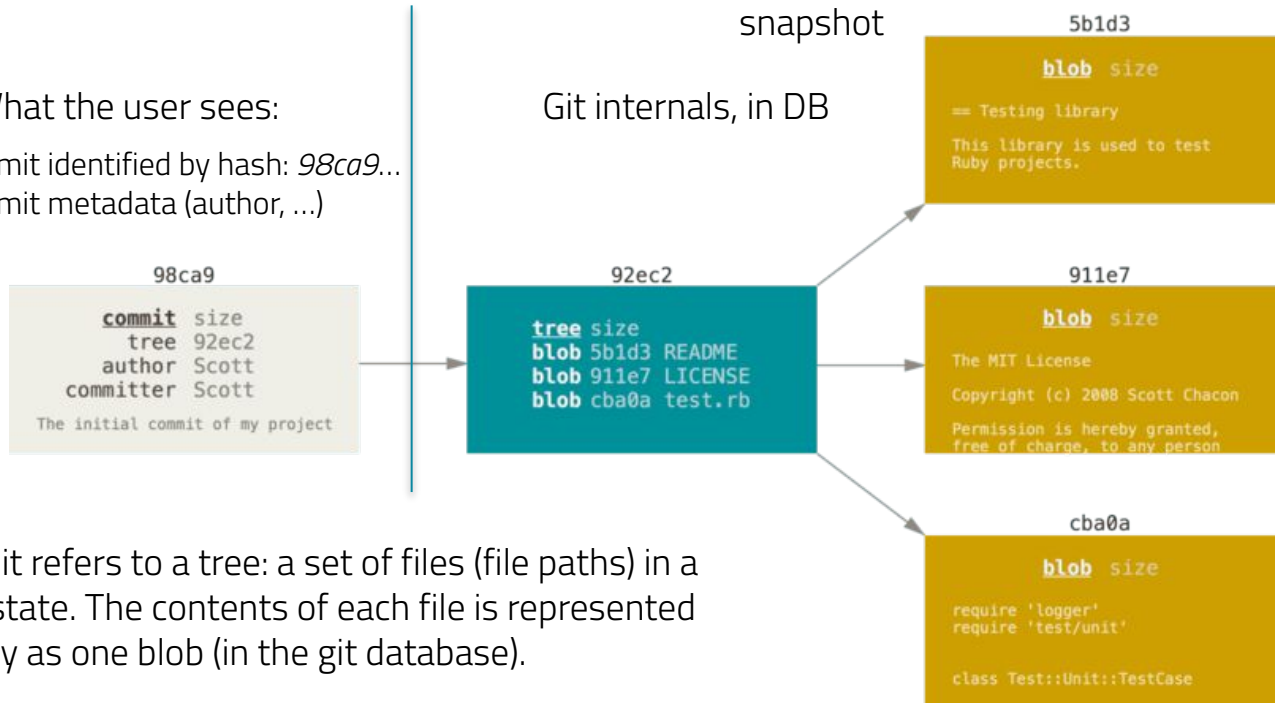
A commit should be made whenever something is done.

What is a commit?

What the user sees:

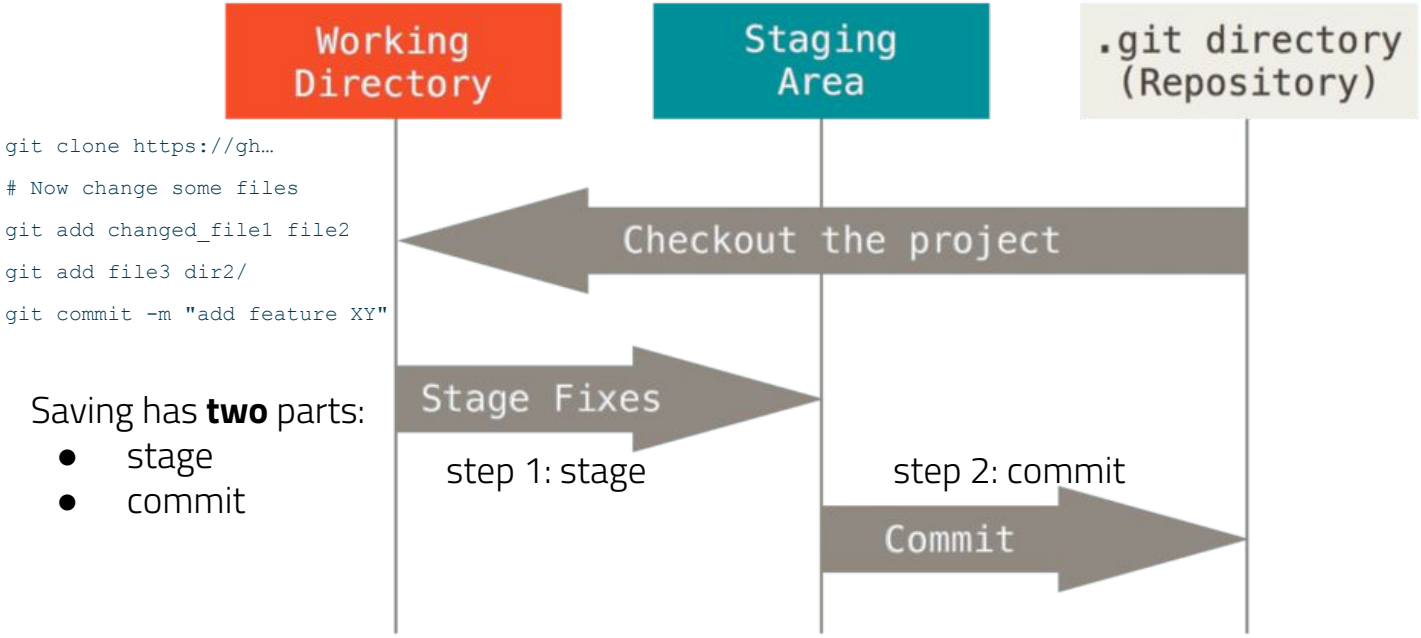
- commit identified by hash: *98ca9…*
- commit metadata (author, …)

Git internals, in DB



A commit refers to a tree: a set of files (file paths) in a certain state. The contents of each file is represented internally as one blob (in the git database).

The three areas of a git repo: working dir, staging area, repo (database)

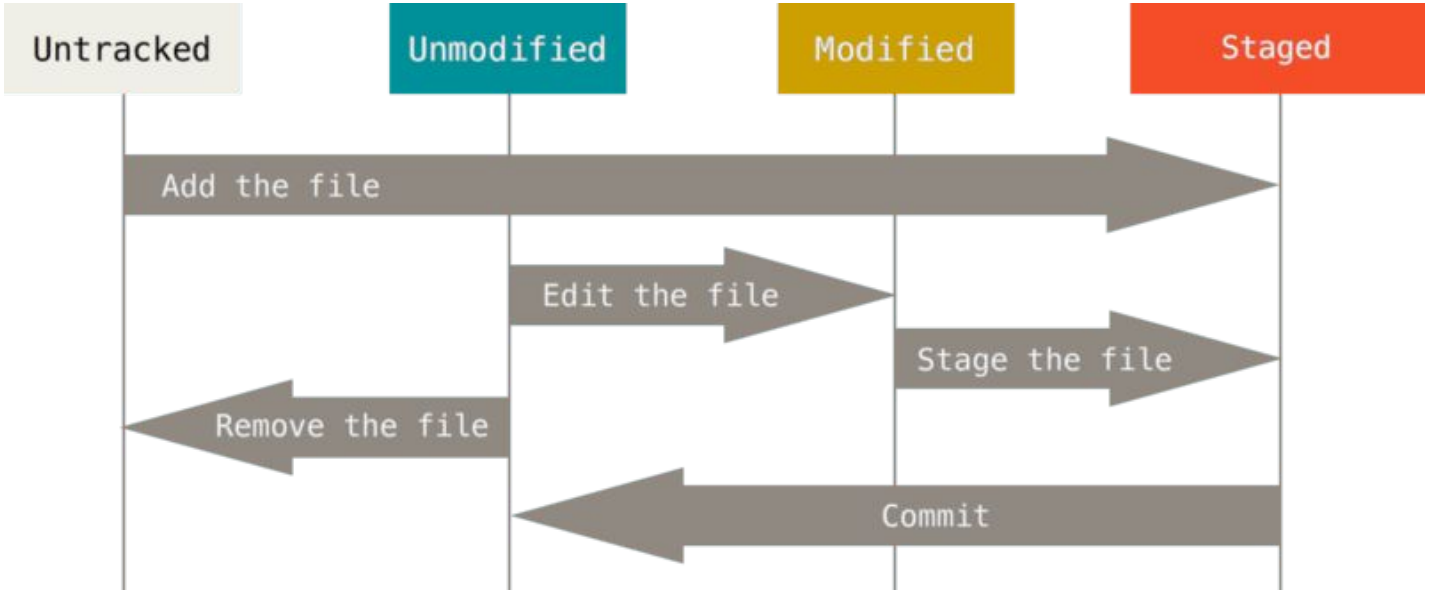


Saving has **two** parts:

- stage
- commit

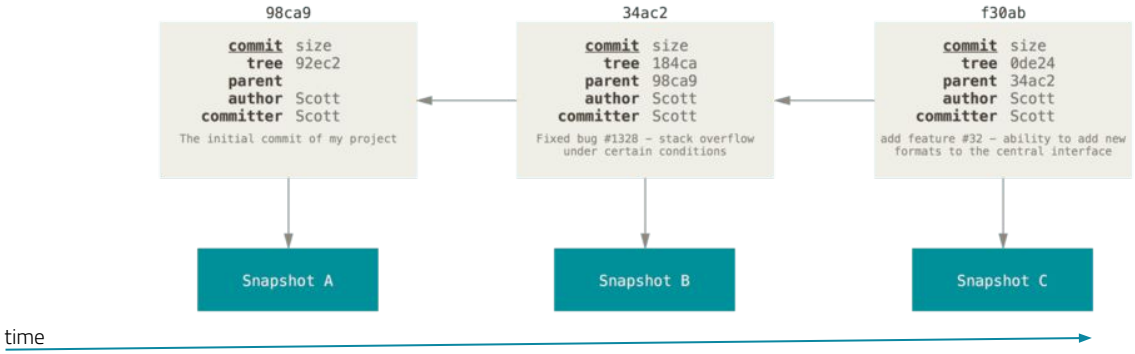
Adding changes to a repository: file states

The possible states of files



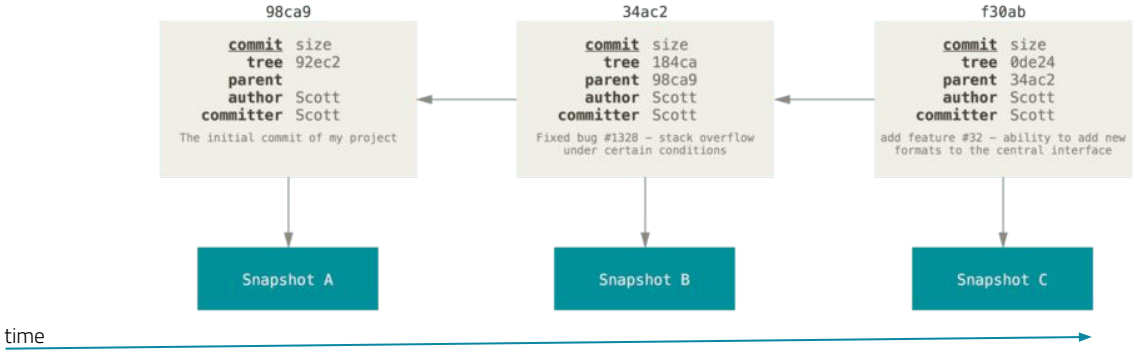
Note: Visualization of Commit History

- Initial commit in a repo: root commit, no parent.
- All other commits have a parent commit (one or more).



Note: Visualization of Commit History

- Initial commit in a repo: root commit, no parent.
- All other commits have a parent commit (one or more).



Simplified representation:

Histories of commits: Commit trees and Branches

Simplified representation:

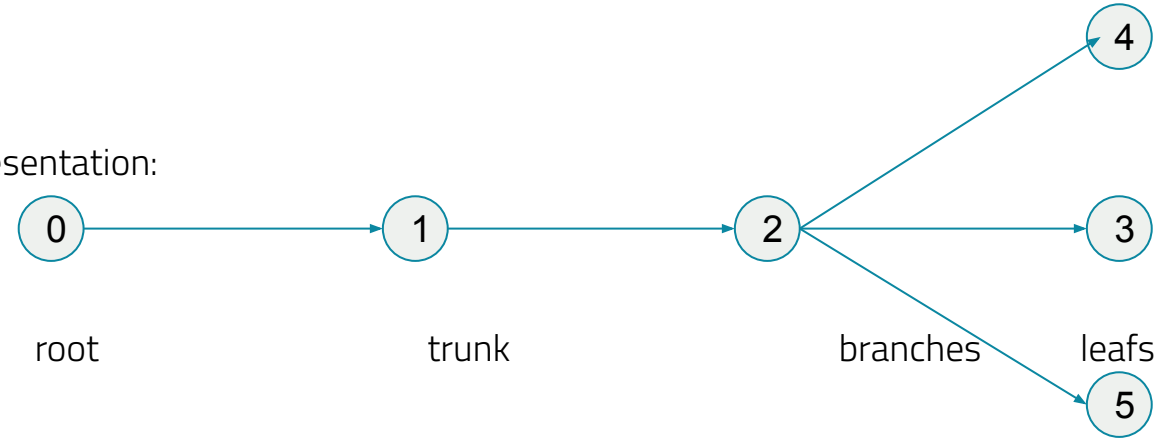
root

trunk

leaf / tip

Histories of commits: Trees and Branches

Simplified representation:



Checking the status and history

- Show the *diff* (difference) for a changed file (compared to last commit)
  - `git diff path_to/file.py`
- Show repo status (changed files, staged files, …)
  - `git status`
- Show history (previous commits and commit messages)
  - `git log`
- Browse using web interface (e.g., `gitweb`) or other GUI

Remember:
```
git help diff
git help status
git help log
```

Recap: Basic Git Concepts

Repository

A git project folder, including a database storing the snapshots of your files and metadata. Turn a folder into a repo with *git init*.

Checkout / Snapshot

All project files from a repository at a certain point in time.

Staging, to stage

Mark changes (changed files) to be written to the repository on the next commit. Stage a file with *git add <file>*.

To commit, a commit

Add a set of changed to the versioning database. Commit with *git commit -m <msg>*.

3.
Distributed Git

Remotes, Servers and Collaboration

Very simple workflow: 1 person, 2 PCs

We now have a *server*, e.g., a repo on Github.com or a local git server.

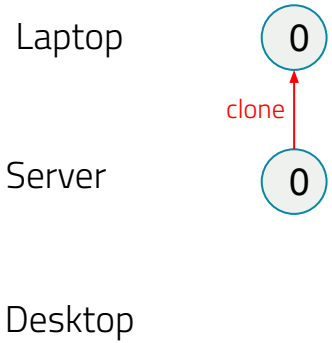


Image credit: https://www.atlassian.com/git/tutorials/syncing/git-pull

Very simple workflow: 1 person, 2 PCs

We now have a *server*, e.g., a repo on Github.com or a local git server.

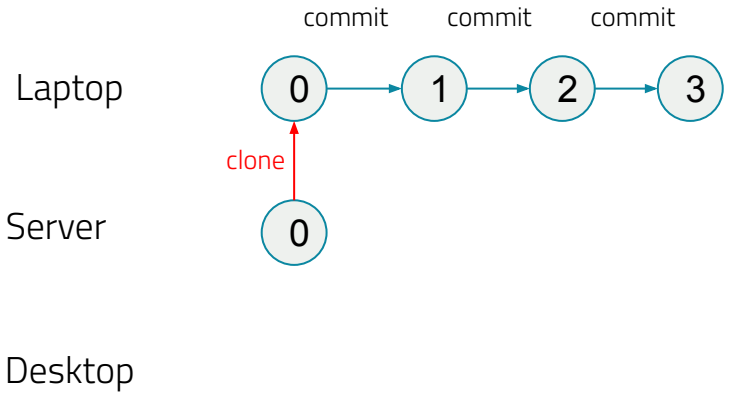


Image credit: https://www.atlassian.com/git/tutorials/syncing/git-pull

Very simple workflow: 1 person, 2 PCs

Sync changes to server: git push

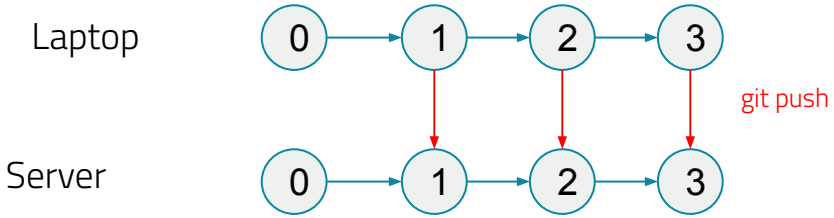


Image credit: https://www.atlassian.com/git/tutorials/syncing/git-pull

Very simple workflow: 1 person, 2 PCs

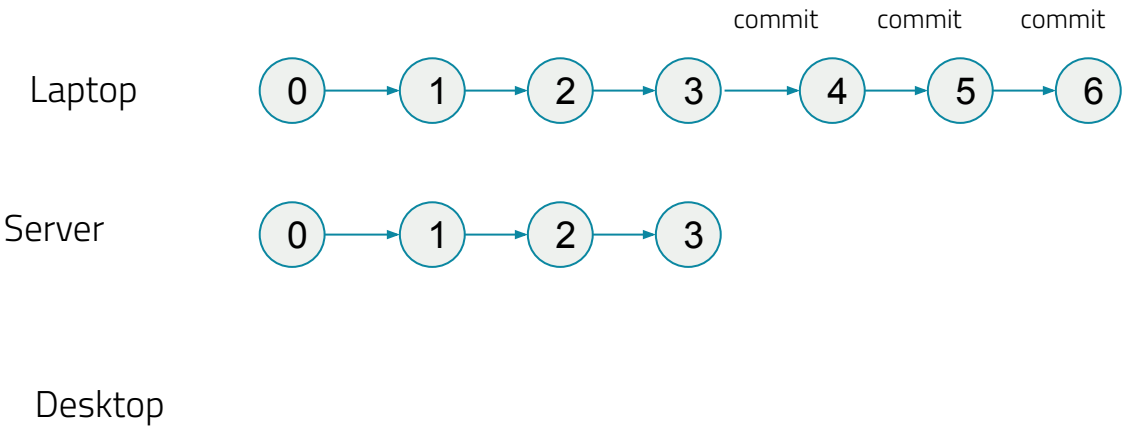


Image credit: https://www.atlassian.com/git/tutorials/syncing/git-pull

Very simple workflow: 1 person, 2 PCs

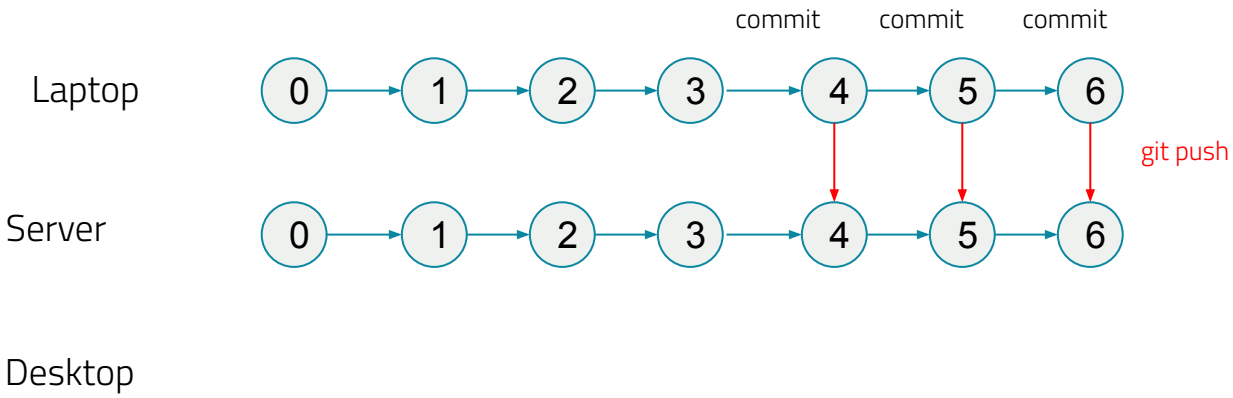


Image credit: https://www.atlassian.com/git/tutorials/syncing/git-pull

Very simple workflow: 1 person, 2 PCs

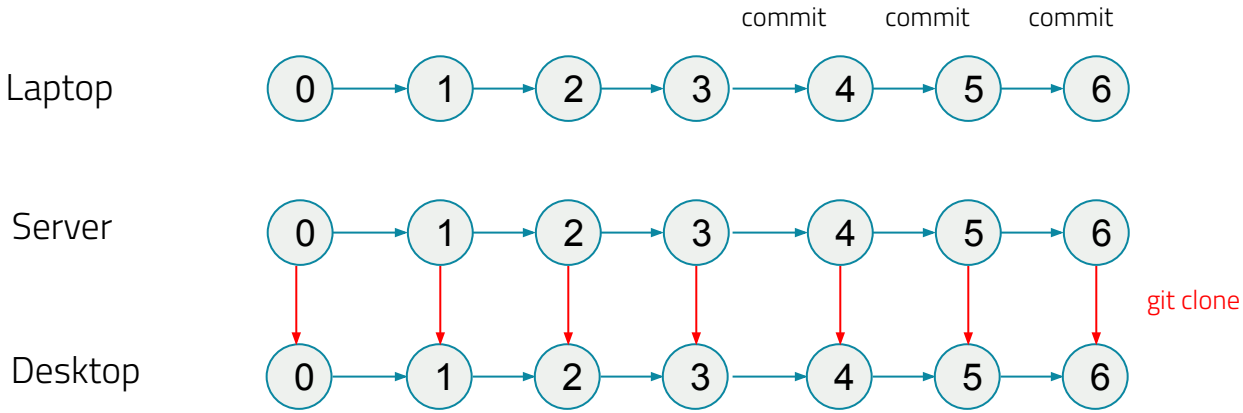


Image credit: https://www.atlassian.com/git/tutorials/syncing/git-pull

Very simple workflow: The git commands

Your Desktop PC at work

Your Laptop at Home

```
create repo at
https://github.com/me/pr1
```

server

```
git clone https://github.com/me/pr1
```

```
git clone https://github.com/me/pr1
cd pr1/
git add file2 file5
git commit -m "Work at ESI done."
git push
```

```
cd pr1/
git pull
git add file2 file5 file7
git commit -m "Work pkg1 at laptop done."
git add file3 file7
git commit -m "Work pkg2 at laptop done."
git push
```

```
git pull
# Start working, then commit & push.
```

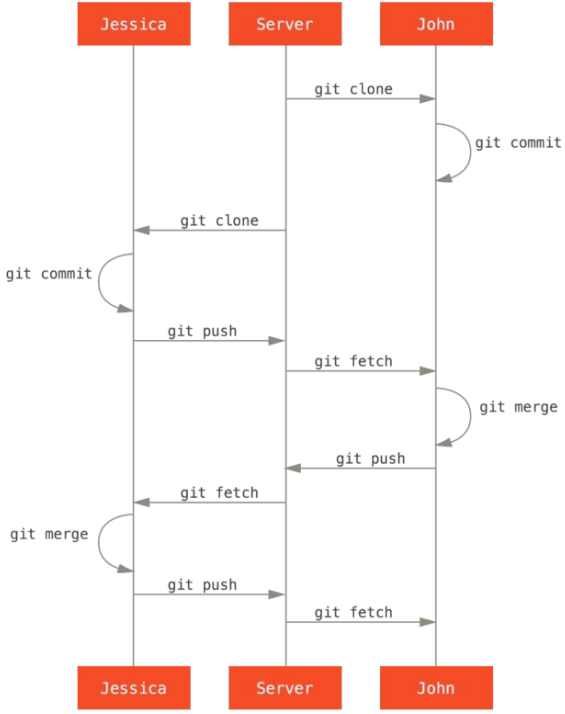
Time

The Git Cycle

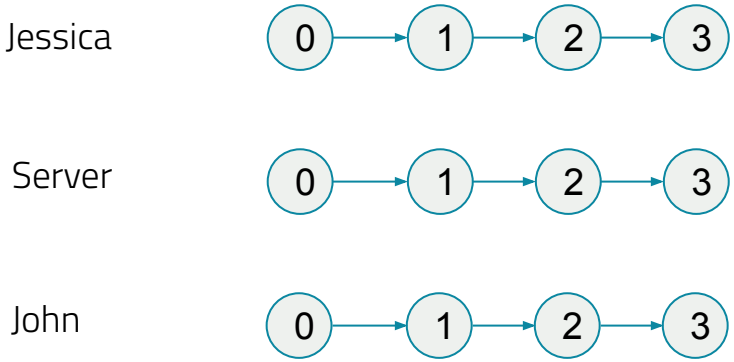
- `git pull`
- repeat:
    1. change stuff
    2. `git add`
    3. `git commit`
- `git push`

New Workflow: 2 Developers, central server

- Everyone on the team has direct write access
- Almost identical to the Desktop PC + Laptop scenario
- But: people work **in parallel**, so there can be **conflicts**!

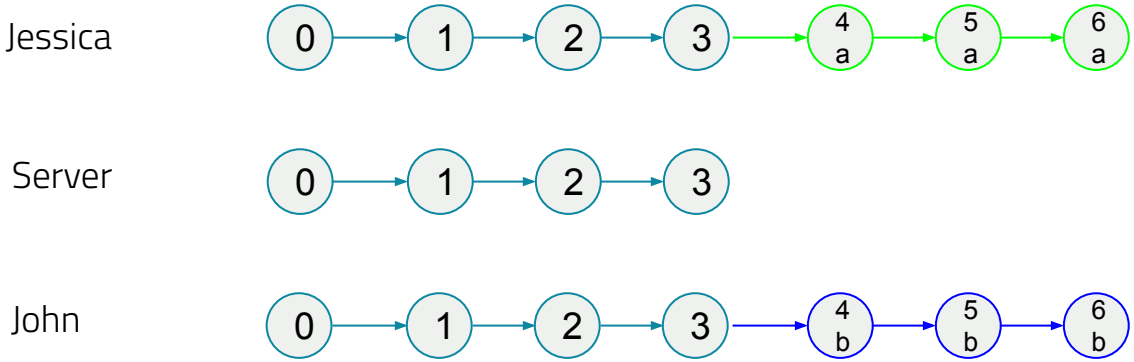


Auto-merging and Conflicts



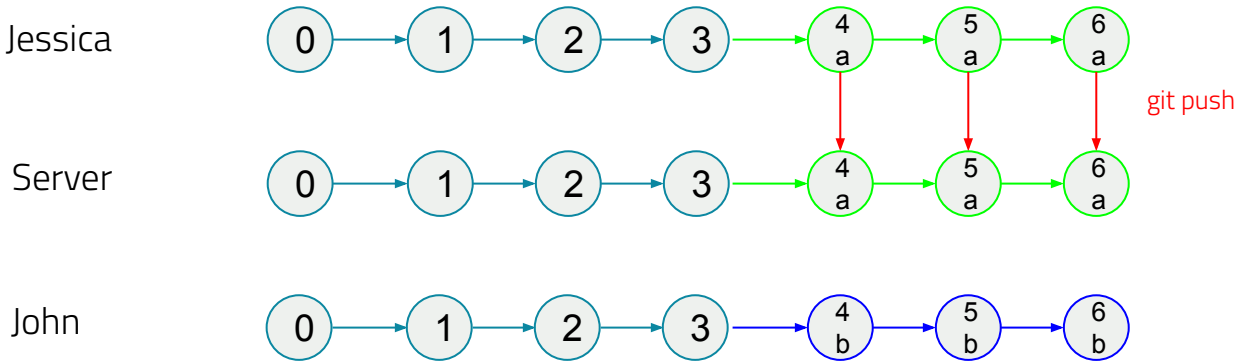
Auto-merging and Conflicts

Two developers working in parallel



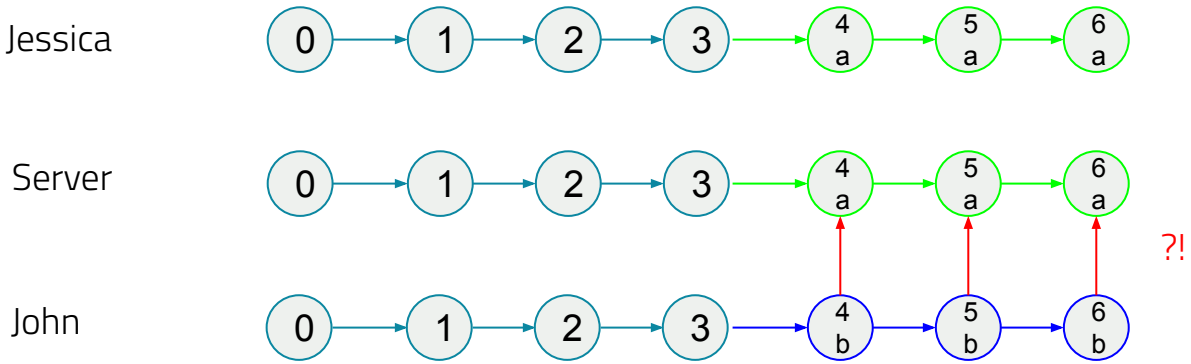
Auto-merging and Conflicts

Two developers working in parallel



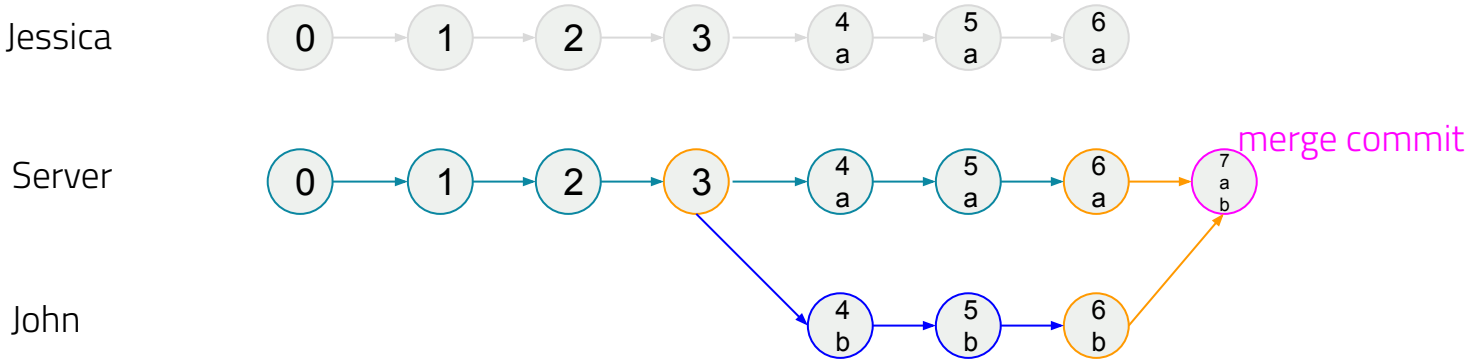
Auto-merging and Conflicts

Two developers working in parallel



Auto-merging and Conflicts

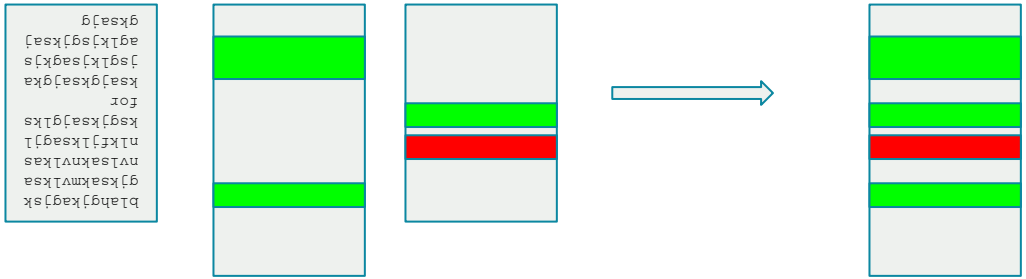
3-way merge: this creates a merge commit from 3 older commits.



Jessica's changes: 6a - 3
John's changes: 6b - 3

Merging scenario 1: auto merge possible

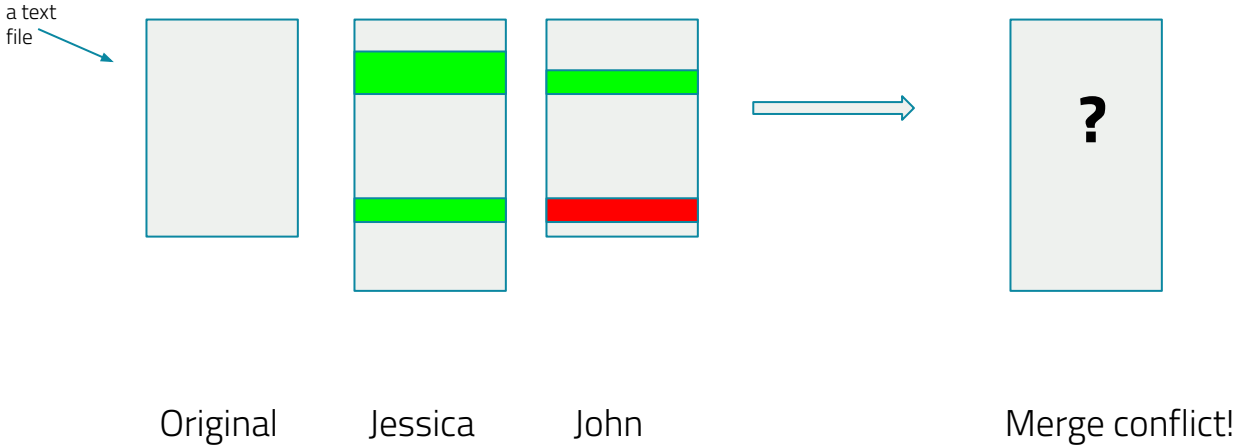
a text file



Original Jessica John Okay: auto merge

added
deleted

Merging, scenario 2: merge conflict



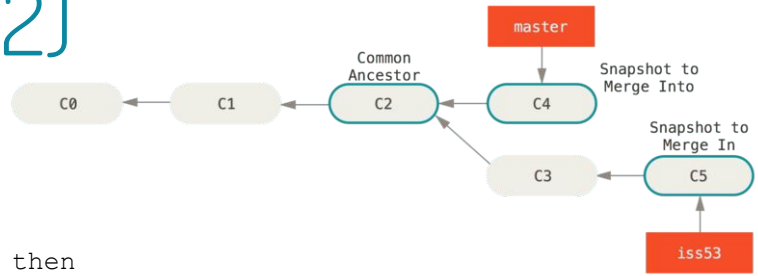
Merging and merge Conflicts

- For non-conflicting changes, merges happen automatically: git does it for you!
- If we changed *the same line in the same file in different ways* on the two branches, a **conflict** occurs.

Merge Conflicts (1/2)

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in
index.html
Automatic merge failed; fix conflicts and then
commit the result.
```

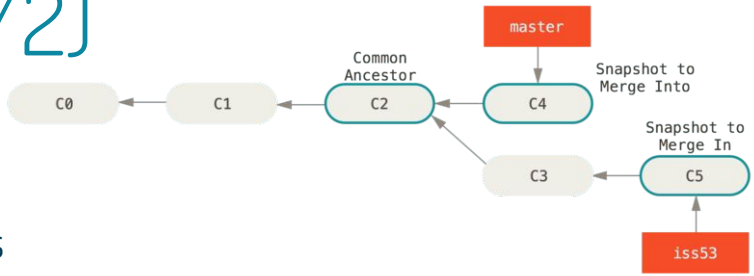
```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

no changes added to commit (use "git add"
and/or "git commit -a")
```



We find somewhere in the *index.html* file:

```
<<<<<<< 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
```

Merge Conflicts (2/2)

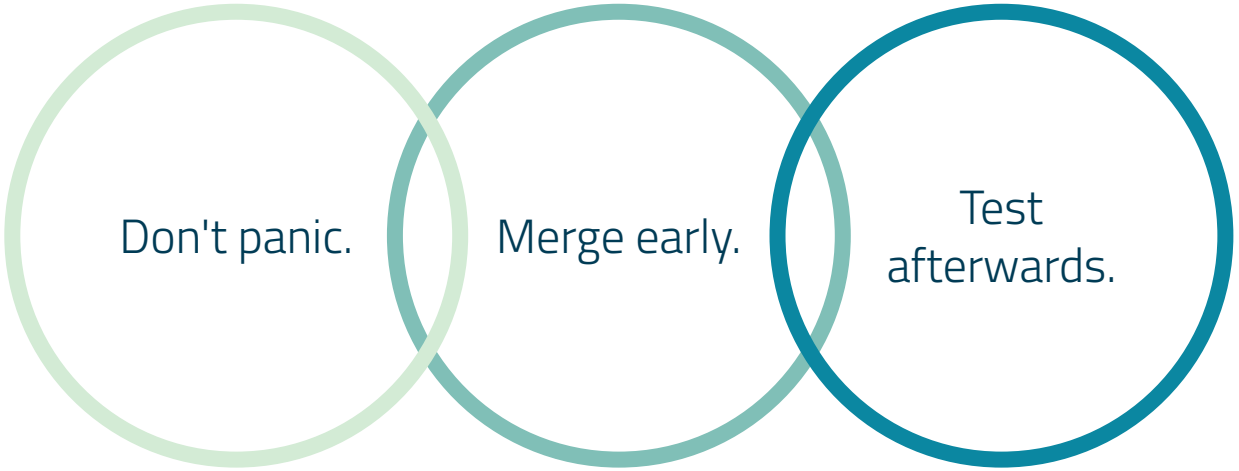


- Talk to your collaborator who introduced the changeset that is conflicting with yours.
- Your options are: accept yours, accept theirs, or merge manually. Also remove the `<<<<<<<`, `=======`, `>>>>>>>` lines.
- Once done, use `git add` on the file to mark the conflict as resolved, then commit.

We find somewhere in the *index.html* file:

```
<<<<<<< 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
```

Recommendations for Merge Conflicts



Undoing uncommitted changes (1/2)

- Unstage a staged file with *git reset*

```
$ git add *  # Stage CONTRIBUTING.md by accident.
$ git status   # Notice it.
$ git reset HEAD CONTRIBUTING.md   # Unstage.
```

Unstages the changes, so the file CONTRIBUTING.md is in state *modified* afterwards.

Alternative with git restore:

```
git restore --staged CONTRIBUTING.md
```

- Change a local commit (msg, files) with *amend*

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

Overwrites the old commit with the new one. Does not make sense if the commit has already been pushed to a remote.

Undoing uncommitted changes (2/2)

- Discarding your changes to a locally modified file

```
$ git status
$ git diff CONTRIBUTING.md
$ git checkout -- CONTRIBUTING.md
```

Dangerous: discards your changes, without ever committing them. The file CONTRIBUTING.md is in state *unmodified* afterwards.

Alternative:

```
git restore CONTRIBUTING.md
```

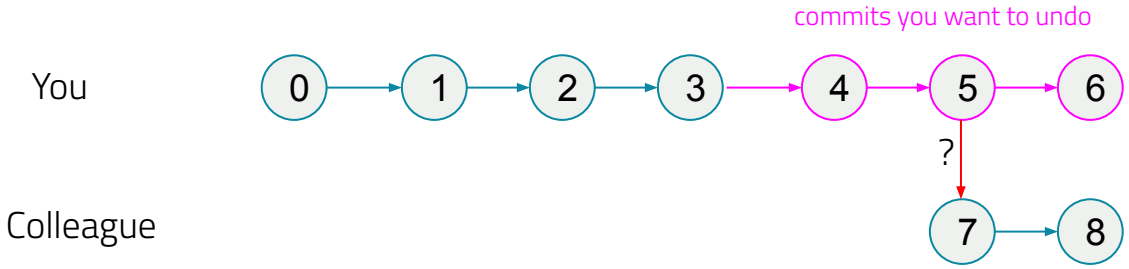
- Alternative: Discard *all* local changes securely

```
$ git stash
```

Will be discussed in more detail later.

Undoing **committed** changes

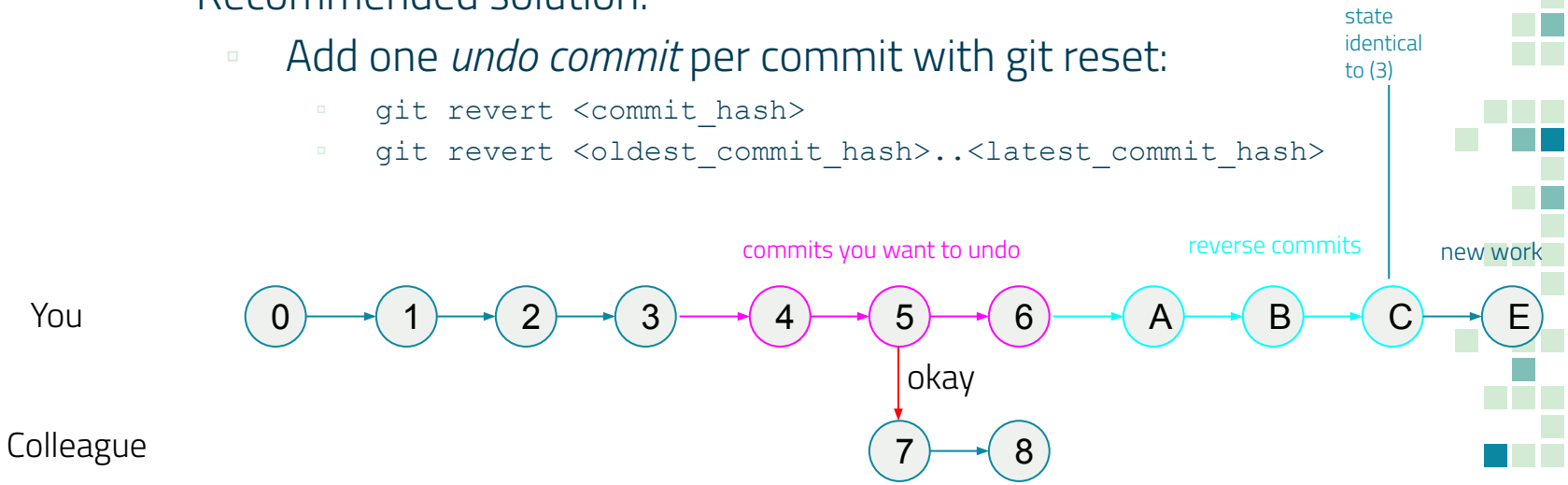
- This is dangerous territory if you have pushed already!



Others may have continued **based on the state you are about to undo**!

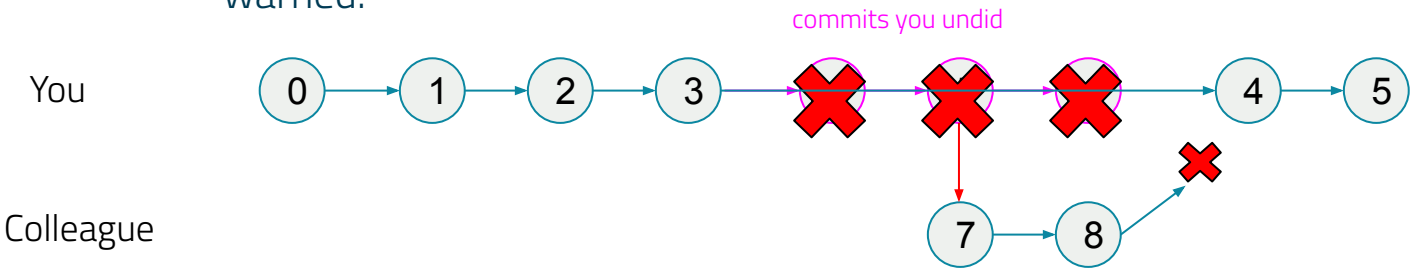
Undoing committed changes: The save version

- Recommended solution:
  - Add one *undo commit* per commit with git reset:
    - `git revert <commit_hash>`
    - `git revert <oldest_commit_hash>..<latest_commit_hash>`



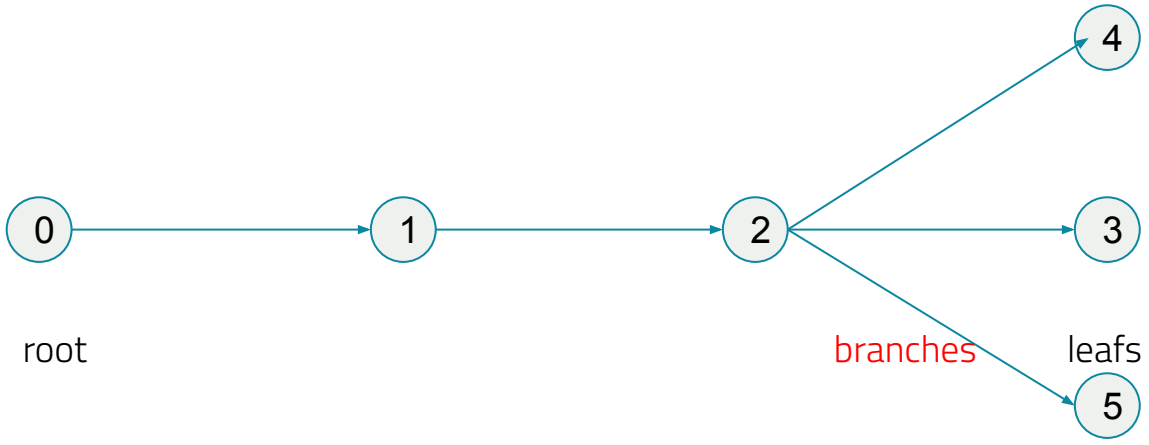
Undoing committed changes: The dangerous version

- You are working alone, do not want the undo commits in the history, and are prepared for more trouble that may happen in the future?
  - Read up on `git reset --hard` and `git push --force`. You have been warned.



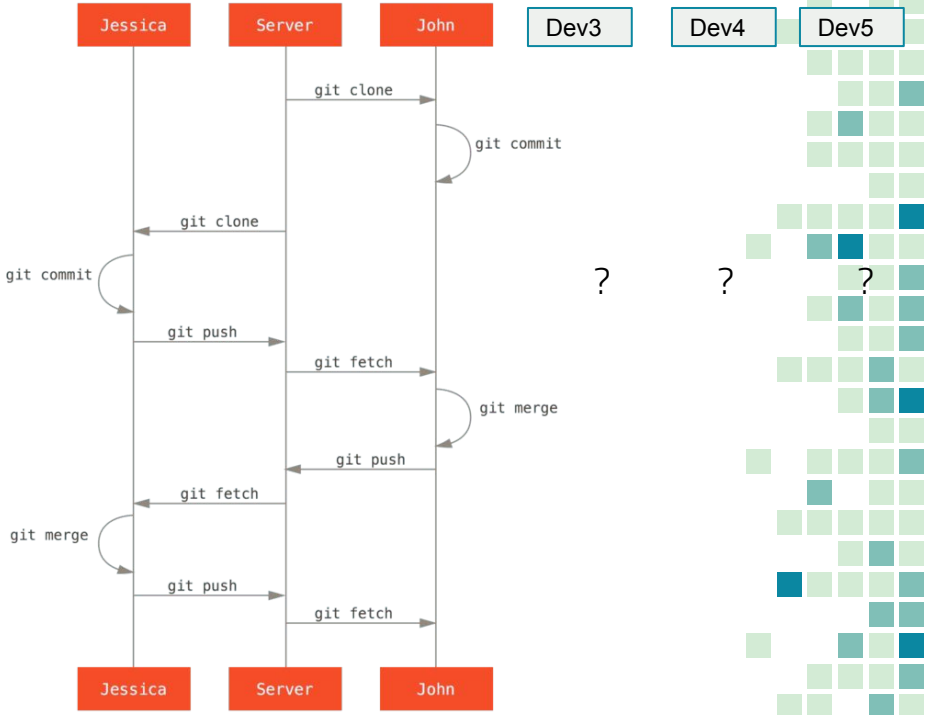
Note: This may actually be needed in some situation, e.g., if stuff got into the repo that really must not be there (e.g., from a legal or security point of view). Just talk to colleagues **first**!

Branches



Simple Workflow: Does it scale?

- Everyone on the team has direct write access
- People work on the same code
- Not feasible for complex software or more than 2 developers
  - broken state from another person
  - many conflicts



Solution for larger projects: Branching and Branches

Purpose

Diverge from main line of development, continue working in a separate copy without affecting the main branch. (Maybe unite later?)

Usage Examples

Start working on a Linux Version of a Windows software, or a new major version: new features in 1 branch, but same security fixes.

Note: The default branch is often called *main* or *master*.

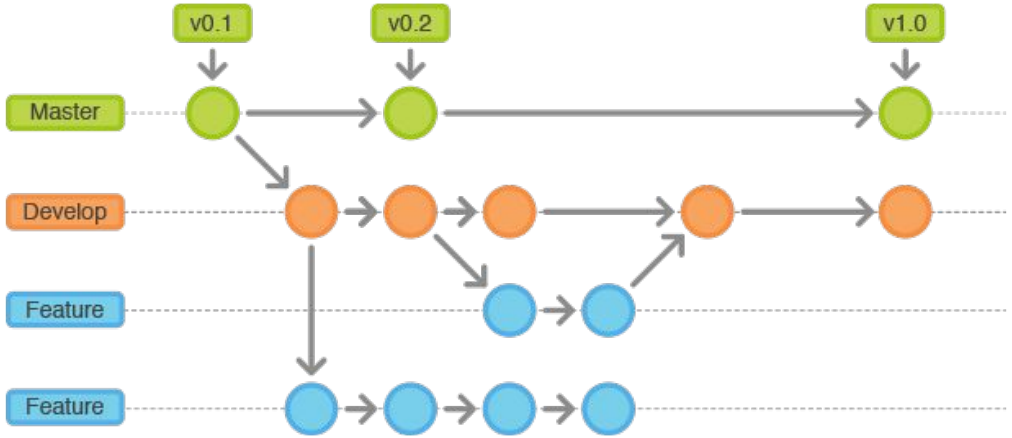


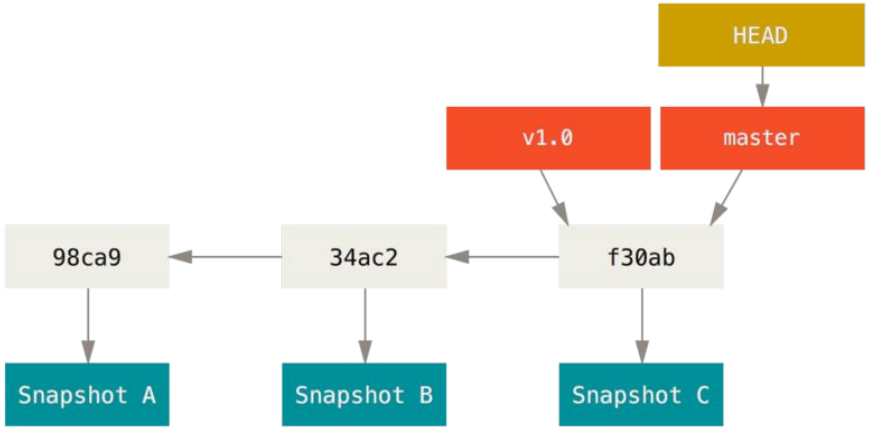
Image Credit: *Lutece* CMS, https://lutece.paris.fr/support/wiki/git.html, License: BSD-3-Clause

Branches, tags and HEAD

Branches are lightweight

A branch in Git is simply a lightweight movable pointer to a commit.

The pointer is moved automatically when you commit.



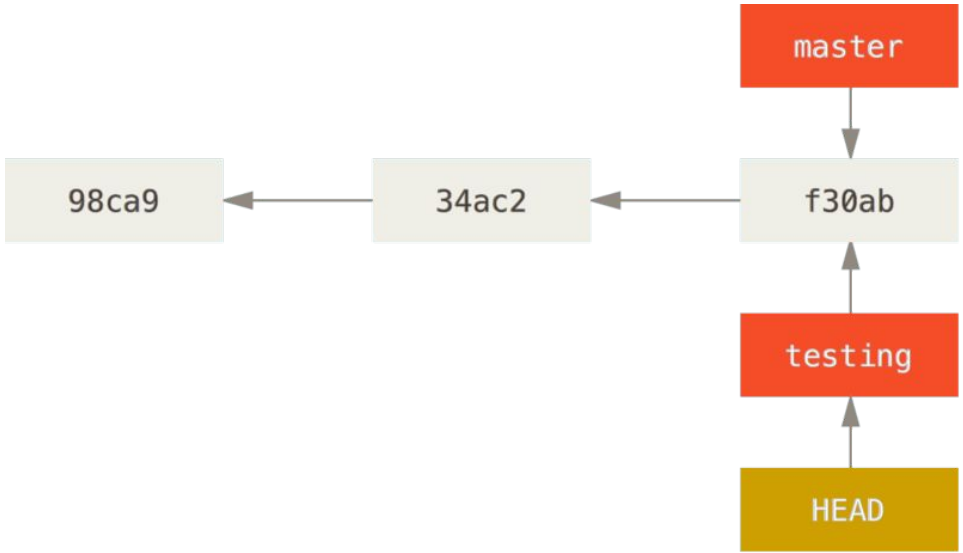
Head: the snapshot of your last commit on that branch.

Changing the active branch

Branches are lightweight

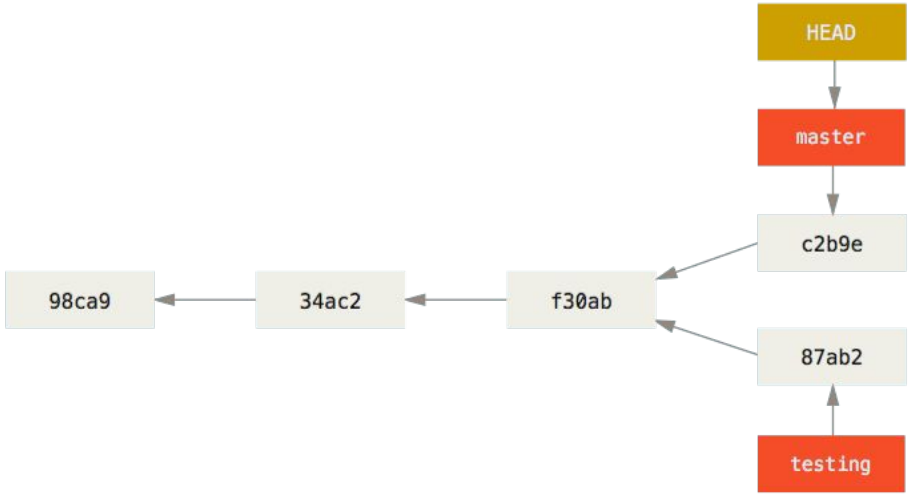
- Changing the branch just moves the pointer named *HEAD* to a new commit.
- Example:

```
git checkout testing
```

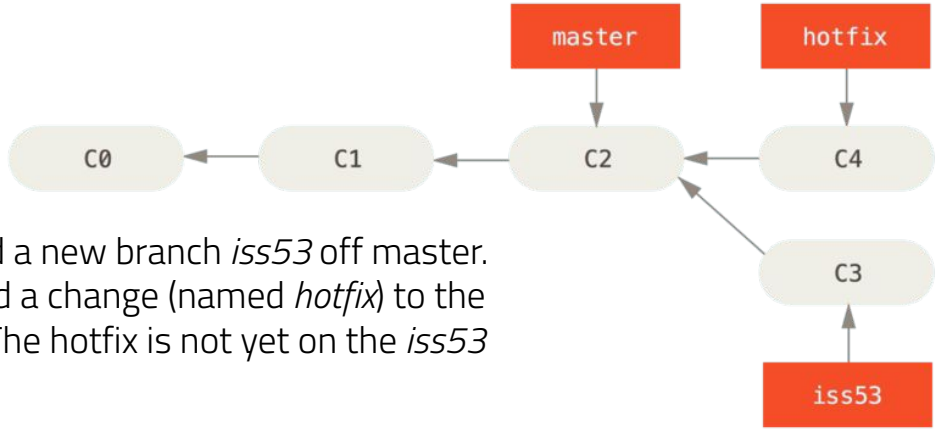


Diverging Branches

- After committing to a new branch, the branches have diverged.

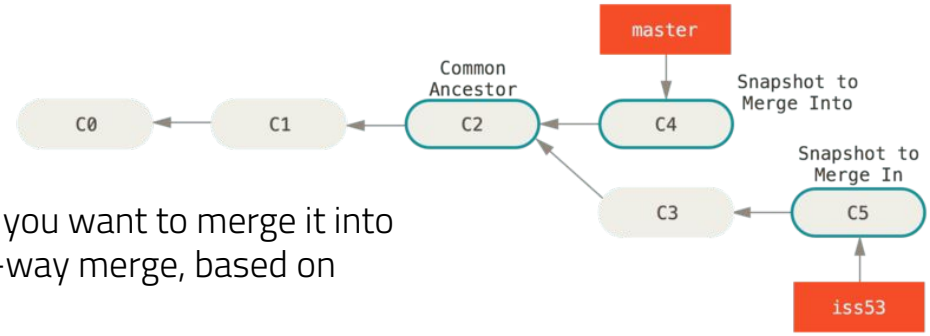


Branching: A practical example (1 / 3)



You have created a new branch *iss53* off master. Then, you applied a change (named *hotfix*) to the master branch. The hotfix is not yet on the *iss53* branch.

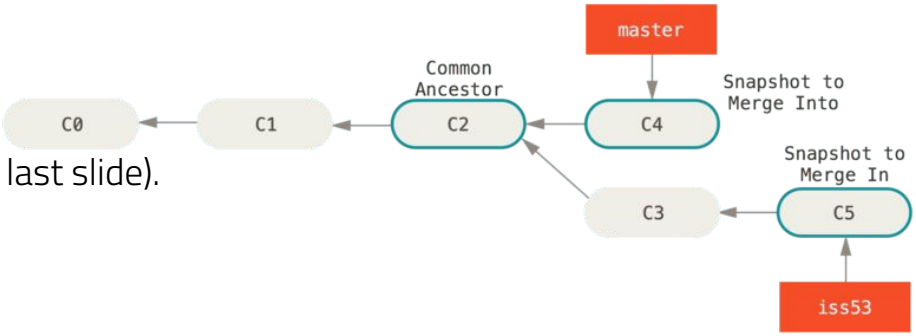
Branching: A practical example (2 / 3)



The branch *iss53* is done and you want to merge it into master. This involves a three-way merge, based on commits C2, C4 and C5.

Git cannot simply move the pointer, it has to create a new snapshot (from a merge of C2, C4 and C5), and thus a new commit. This commit is a *merge commit*, it is special because it has *several parents*.

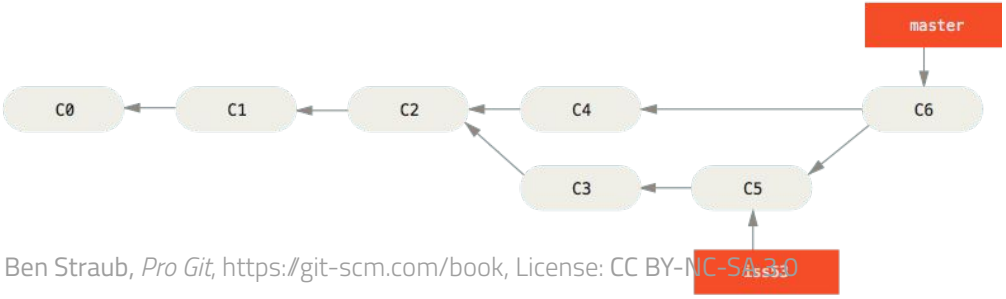
Branching: A practical example (3 / 3)



Situation before merge (from last slide).

```
git checkout master
git merge iss53
```

The result after the merge: C6 is the merge commit, on the master branch.



Remote Branches

Tracking of remote branches

```
git checkout -b iss53
git push
git push origin iss53
```

Get info on new remote branches

```
git fetch
git fetch --all
git checkout iss76
git checkout remote2/iss76
```

Rebasing

Alternative to merging.

Only usable for series of commits (in different branches) that have not been pushed.

Allows for "cleaner" history (i.e., not the true history, but one that is easier to read/understand).

Works by making parallel changes on different branches appear to have happened sequentially on a single branch.

It is a matter of taste whether you want to use it.

See `git rebase`.

More info on Remotes

- Remote: another repository representing the same project
  - Can be on same computer or on remote server
  - May lack some of your commits, and/or may contain commits (or branches) you do not have locally.
- The default remote is known as the origin.
  - You can add remotes, change the origin to another remote, or remove the origin.

Remotes

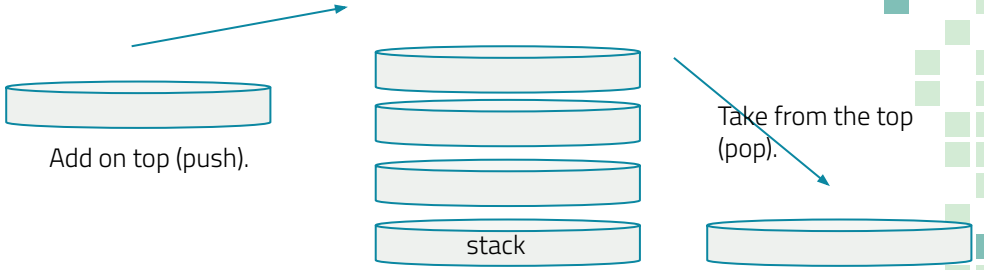
```
$ git clone https://github.com/esi-neuroscience/syncopy
$ cd syncopy
$ git remote -v
origin    https://github.com/esi-neuroscience/syncopy (fetch)
origin    https://github.com/esi-neuroscience/syncopy (push)

# Add another remote and fetch from it.
git remote add mycol https://github.com/my_colleague/syncopy
get fetch mycol

# Show origin, including tracked and new/stale branches.
git remote show origin
```

Tags

- A name for a commit/state
  - E.g., "v1.0" or "biolpsych_paper"
  - Very useful to quickly find a specific version later
  - Should use this for every release
- On GitHub, releases are created from tags.
  - See also: software versioning.

```
$ git log --pretty=oneline
$ git tag -a v1.2 9fceb02
$ git show v1.2
$ git push --tags
```

The git stash



- Scenario: You need to switch branch but have work in progress in an ugly state that you do not want to commit.
- Add to stash to have it secured.
  - Can be (ab)used to clean your working directory in a safe way.

```
$ git status           # Will show modified files, maybe some of them stashed.
$ git checkout dev     # Does not work!
$ git stash            # Store the changes on the stash
$ git checkout dev     # Works. Now do some stuff.
$ git stash list
stash@{0}: WIP on master: 049d078 Create index file
stash@{1}: WIP on master: c264051 Revert "Add file_size"
$ git checkout main
$ git stash apply      # Apply the latest one. Or: git stash apply stash@{1}
```

Git Concepts – Remote

Clone (and origin)

A copy of a source repository, with all history. (The repo from which you copied is the origin.)

Pull

To retrieve changes (zero or more commits) from a remote repository (typically the origin).

Push

To send local changes (commits) to a remote repository (typically the origin).

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

```
git push origin master
```

Fetch

Download data from a remote into your local repo under the remotes namespace (without merging anything or changing your files in any way).

Undoing things remotely

- Discarding your changes to a locally modified file

```
$ git status
$ git diff CONTRIBUTING.md
$ git checkout -- CONTRIBUTING.md
```

Dangerous: discards your changes, without ever committing them. The file CONTRIBUTING.md is in state *unmodified* afterwards.

Alternative:

```
git restore CONTRIBUTING.md
```

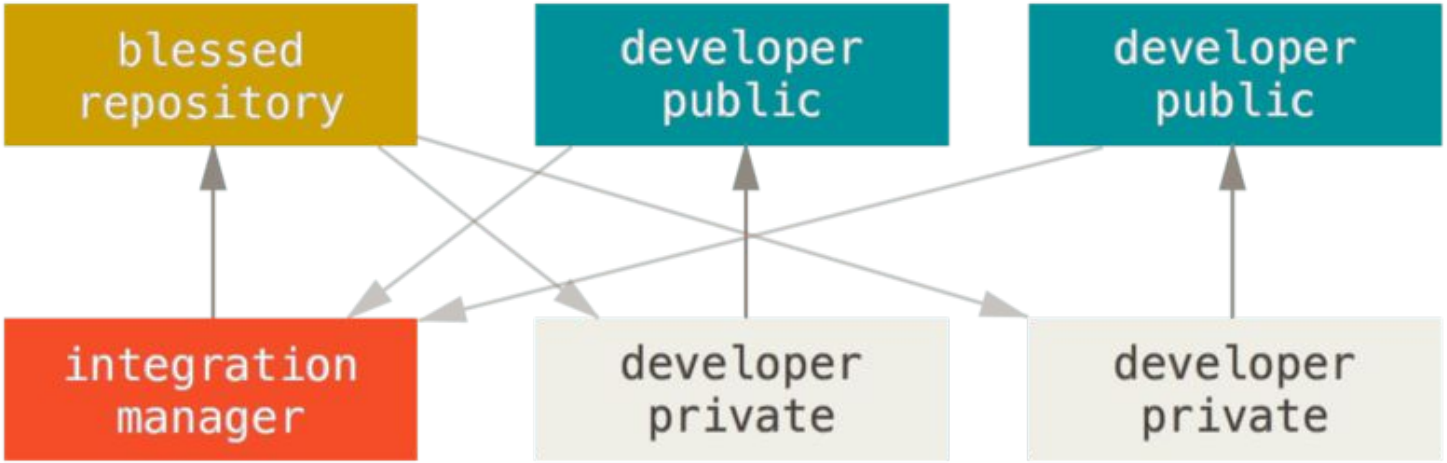
- Change a local commit (msg, files) with *amend*

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

Overwrites the old commit with the new one. Does not make sense if the commit has already been pushed to a remote.

Workflow: Contributing a change to a public project on GitHub

Most projects used the *Integration Manager* workflow, where devs create PRs against the blessed repo and an integration manager decides on them (with help of test suite):



See https://martinfowler.com/articles/branching-patterns.html for more workflows.

Workflow: Contributing a change to a public project on GitHub in 3 steps

1) Talk + Fork

Create an issue and talk to people to see whether they agree to have this changed.

Fork the project to your GitHub account, and clone your fork to your local computer.

2) Make changes

Checkout the proper branch (typically master or develop, but check the project) you want to branch off from.

Create a new branch and commit your changes into it.

3) Create Pull Request

Find your branch on your fork at the GitHub website, and click the *Create Pull Request* button.

Adapt your changes until they pass the review process. The project maintainers will then merge.

4.

Some tips on getting started with Git

IDEs, servers, workflow reminder

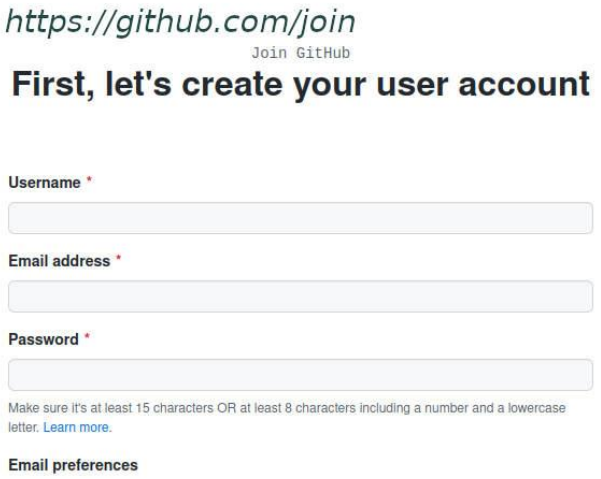
What to do next

Hint: *Maybe ignore branches for now.*

Get an account at a coding platform

like GitHub, SourceForge, GitLab, BitBucket, ...

- Free accounts typically have limits on team size for private repos or other restrictions
- Note: You can often get a professional account for free as a researcher at a non-profit institution or OSS dev
  - Requires verification of status.
  - Not needed to get started.



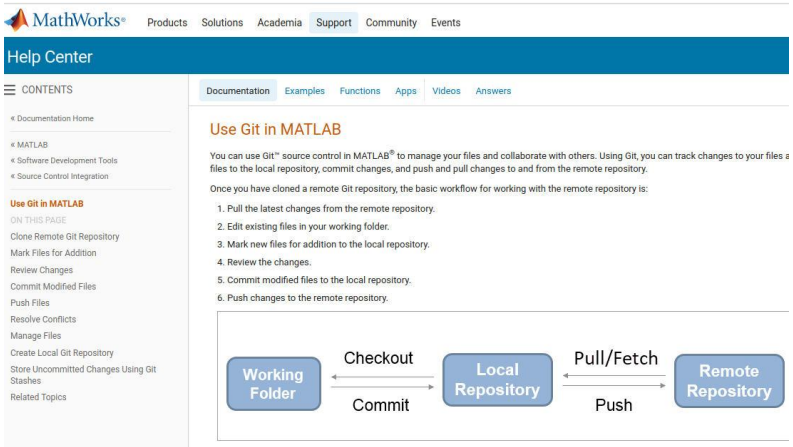
IDE Integration

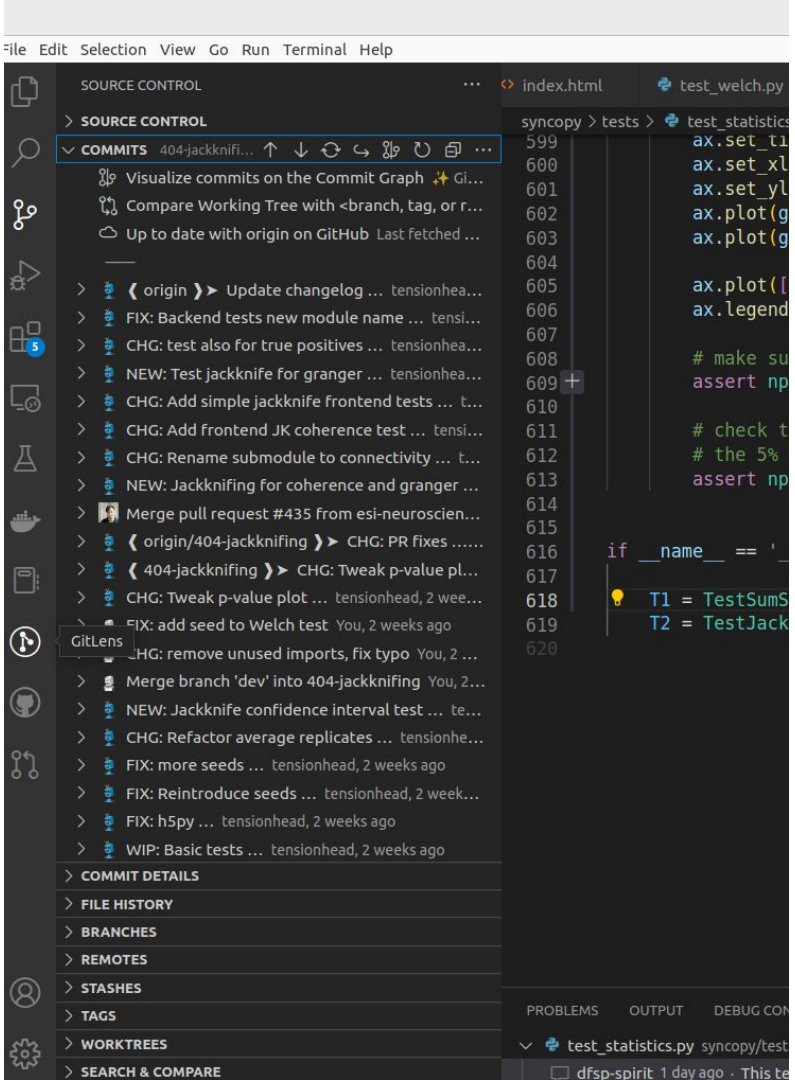
- Git is supported in all major IDEs
  - Matlab, PyCharm, VSCode (Source Control, Gitlens, GitHub, GitHub Copilot), …
- Check the Documentation and PLugins for your IDE

Source control

Git Lens

GitHub





Outlook: Git Features and Topics Not Discussed Yet

- Hooks
  - Automate things, e.g., run custom code after each commit
- Submodules
  - Linking to another repo within yours
- Git Servers
  - How to run your own, on-premises or in the cloud
- Interaction with other VCSs and Services
  - E.g., synchronization to SVN, Jenkins, …
- Large file support (LFS)
  - Store large files in git without slowing down your workflow
- …

THANKS!

Any questions?

CREDITS

- Examples and many figures from the Git Pro Book, License: Creative Commons **CC BY-NC-SA 3.0**
- Presentation made with Google Slides
- Presentation template "Technology Pixels" by SlidesCarnival, License: Creative Commons **CC BY 4.0**
  - https://www.slidescarnival.com/mowbray-free-presentation-template/1932

License

The content on these slides (my work, mostly a remix of the Git Pro book) is licensed under the following license:

Creative Commons **CC BY-NC-SA 3.0**

https://creativecommons.org/licenses/by-nc-sa/3.0/

Note that some figures are other people's work, and are under different (open!) licenses. See each slide / figure for the respective author and license. You can assume that figures without attribution were created by me, and are covered by the CC BY-NC-SA 3.0 license mentioned above.