Git Tutorial and Workshop

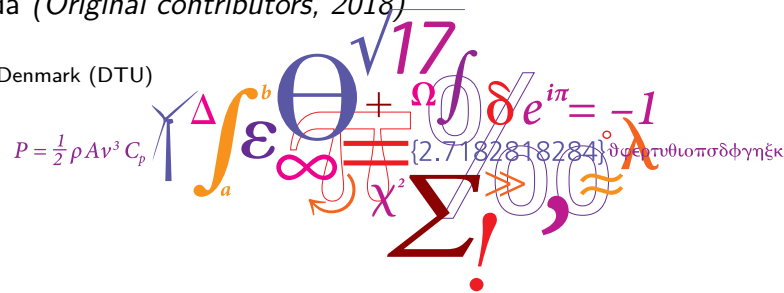
Introduction and hands-on excercises to version control systems

Hybrid, Lyngby, Risø, Tuesday 31st January, 2023

Kai Heussen *(recent mods)*

Alexander M. Prostejovsky, Lasse Orda *(Original contributors, 2018)*

Wind and Energy Systems, Technical University of Denmark (DTU)

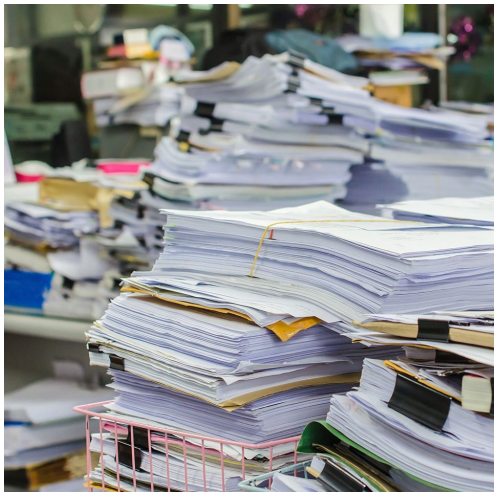


DTU Wind Energy
Department of Wind Energy

Agenda

- Motivation
- Git Introduction
- Prerequisites
- Basic Git Commands
- Advanced Git Commands
- Closing

Code is here, code is there, code is everywhere



[Caveman version control]

Coding for scalability

Linux Kernel project has 1000s of contributors.
No single person understands all lines of code.

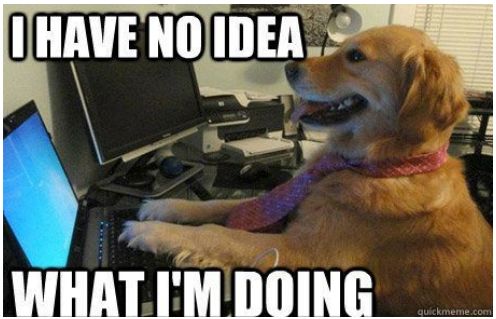
Why YOU want to use a Version Control System

"All nice and well for the Linux kernel, but why would I bother going the extra mile?"

Why YOU want to use a Version Control System

"All nice and well for the Linux kernel, but why would I bother going the extra mile?"

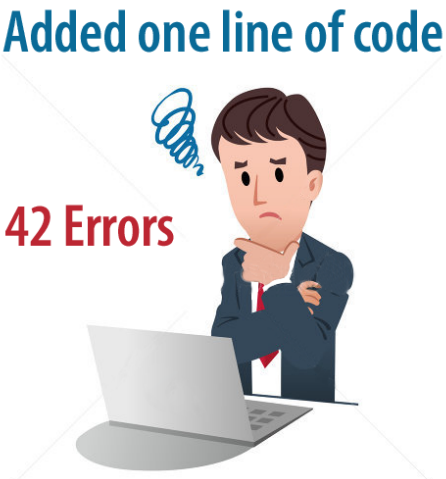
- Documentation of history doesn't only help others but also your future self
- Restore earlier stages of development in case you run into a dead end



Why YOU want to use a Version Control System

"All nice and well for the Linux kernel, but why would I bother going the extra mile?"

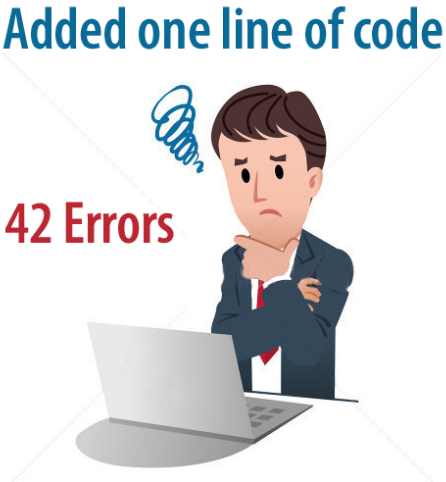
- Documentation of history doesn't only help others but also your future self
- Restore earlier stages of development in case you run into a dead end
- You can't find the reason behind that one damn bug when you're absolutely certain that you didn't change a single thing



Why YOU want to use a Version Control System

"All nice and well for the Linux kernel, but why would I bother going the extra mile?"

- Documentation of history doesn't only help others but also your future self
- Restore earlier stages of development in case you run into a dead end
- You can't find the reason behind that one damn bug when you're absolutely certain that you didn't change a single thing
- Transparent track record of your development achievements
- Easy to be compliant with *NDAs* and *DMPs* – dump Dropbox for code!



Version Control Systems

- What is a Version Control System (VCS)?
  - Tracks changes of documents
  - Revisions can be compared, restored, and merged
  - Multi-developer mechanisms
  - Popular VCS: *Git*, Mercurial, Subversion
- Benefits
  - Collaborative editing of code (merging, branching)
  - Detailed document editing history (who, when, what)
  - Backup in safe place
- Workflow integration
  - Command line tools
  - Plugins for IDEs (Eclipse, PyCharm, MATLAB, etc.)

Courtesy of the Git reference manual [2].

Why we want VCS in Wind and Energy Systems

Preserve group knowledge and keep it alive

Share, reuse and archive code and models to make developments available to current and future group members.

Enhance collaboration between researchers

Multiple people are enabled to work together on the same code, models, and documents, effectively enhancing our collaborative workflow.

Git is already used all over the Department

Ask anyone ;)

DTU Data Management Policy

Storage and IT of research data

Research data must be stored, so the data and the associated documentation (metadata) are findable and accessible by the staff at CEE. The data management plan (DMP) must include description of location of the data and metadata (storage system) and how these can be accessed (access control, legal restrictions etc.). Research data should be stored, so the data are interoperable and reusable. For instance, storage should as far as possible use standard formats.

DTU Data Management Policy

Storage and IT of research data

Research data must be stored, so the data and the associated documentation (metadata) are findable and accessible by the staff at CEE. The data management plan (DMP) must include description of location of the data and metadata (storage system) and how these can be accessed (access control, legal restrictions etc.). Research data should be stored, so the data are interoperable and reusable. For instance, storage should as far as possible use standard formats.

Storage system	Applications				Properties			
	Personal data	Open data	Restricted data (access control)	Long-term preser-vation	Multi-user access	External sharing	Versio ning	Logging
Energy Data DK	✓	✓	✓	✓	✓	✓	-	✓
Open access repositories	-	✓	-	[1]	✓	✓	[3]	[3]
share.dtu.dk	✓	-	✓	-	✓	✓	-	-
O-drive	-	-	[2]	✓	✓	-	-	-
M-drive	✓	-	✓	-	-	-	-	-
data.deic.dk	✓	-	✓	-	-	-	-	-
git.elektro.dtu.dk	✓	✓	✓	✓	✓	✓	✓	-

DTU Data Management Policy

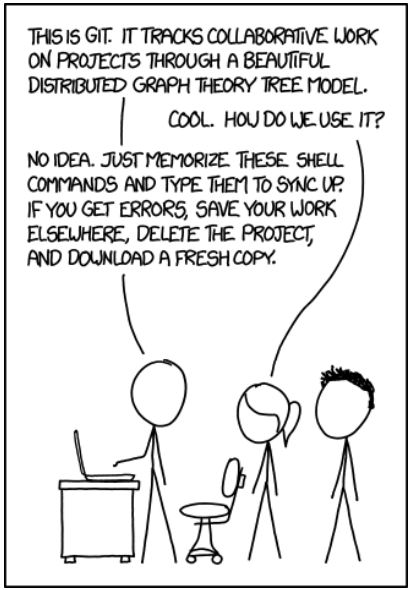
Storage and IT of research data

Research data must be stored, so the data and the associated documentation (metadata) are findable and accessible by the staff at CEE. The data management plan (DMP) must include description of location of the data and metadata (storage system) and how these can be accessed (access control, legal restrictions etc.). Research data should be stored, so the data are interoperable and reusable. For instance, storage should as far as possible use standard formats.

Storage system	Applications				Properties			
	Personal data	Open data	Restricted data (access control)	Long-term preser-vation	Multi-user access	External sharing	Versio ning	Logging
Energy Data DK	✓	✓	✓	✓	✓	✓	-	✓
Open access repositories	-	✓	-	[1]	✓	✓	[3]	[3]
share.dtu.dk	✓	-	✓	-	✓	✓	-	-
O-d[illegible]	-	-	[2]	✓	✓	-	-	-
[illegible]-drive	✓	-	✓	-	-	-	-	-
data.deic.dk	✓	-	✓	-	-	-	-	-
git.elektro.dtu.dk	✓	✓	✓	✓	✓	✓	✓	-

Introduction to Git

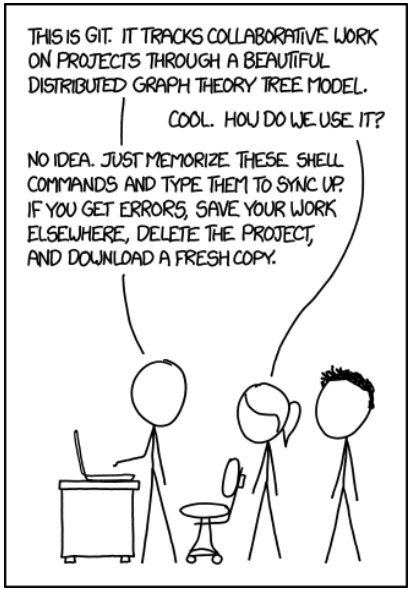
- Git is a fully distributed VCS *"even if your workflow isn't"*.
- Features & Advantages
  - Distributed backend prevents data loss
  - No connection required for working
  - Very fast because most operations are done locally
  - Data size on client side small due to compression



Courtesy of XKCD.com [5].

Introduction to Git

- Git is a fully distributed VCS *"even if your workflow isn't"*.
- Features & Advantages
  - Distributed backend prevents data loss
  - No connection required for working
  - Very fast because most operations are done locally
  - Data size on client side small due to compression
- Disadvantages
  - Poor handling of BLOBs (=Binary Large OBjects)
- Differences to other VCS
  - Distribution of repositories
  - Keeps full file system as opposed to file differences
  - Rewrite of history very difficult



Courtesy of XKCD.com [5].

Git Concepts - Overview

"Holy cow, do I need all this?"

Git Concepts - Overview

"Holy cow, do I need all this?"

- Git Architecture Overview
  - **Remote Repository:** Project database on a remote server
  - **Local Repository:** Local project database
  - **Index/Staging:** List of files selected for being committed to the local repository
  - **Workspace:** Working copy of local repository
  - **Stash:** Temporary "drawer", where changes can be stored for later

Git Concepts - Overview

"Holy cow, do I need all this?"

- Git Architecture Overview
  - **Remote Repository:** Project database on a remote server
  - **Local Repository:** Local project database
  - **Index/Staging:** List of files selected for being committed to the local repository
  - **Workspace:** Working copy of local repository
  - **Stash:** Temporary "drawer", where changes can be stored for later
- File stages
  - **Untracked:** File is not version controlled
  - **Unmodified:** Workspace and local repository file are identical
  - **Modified:** File has been changed but not committed
  - **Staged:** Modified file that is selected for the next commit

Git workflows

- Clone
- Add-Commit-Push workflow
- Pull-add-commit-push workflow
- Branching, merging and conflict resolution (later)

Add-commit-push workflow

"I just need my remote backup"

- Useful for single-person use
- Uses Git as backup and history
- Assumes no-one changes the remote repo
  e.g. If you change a file in the Gitlab web interface, you break this flow.

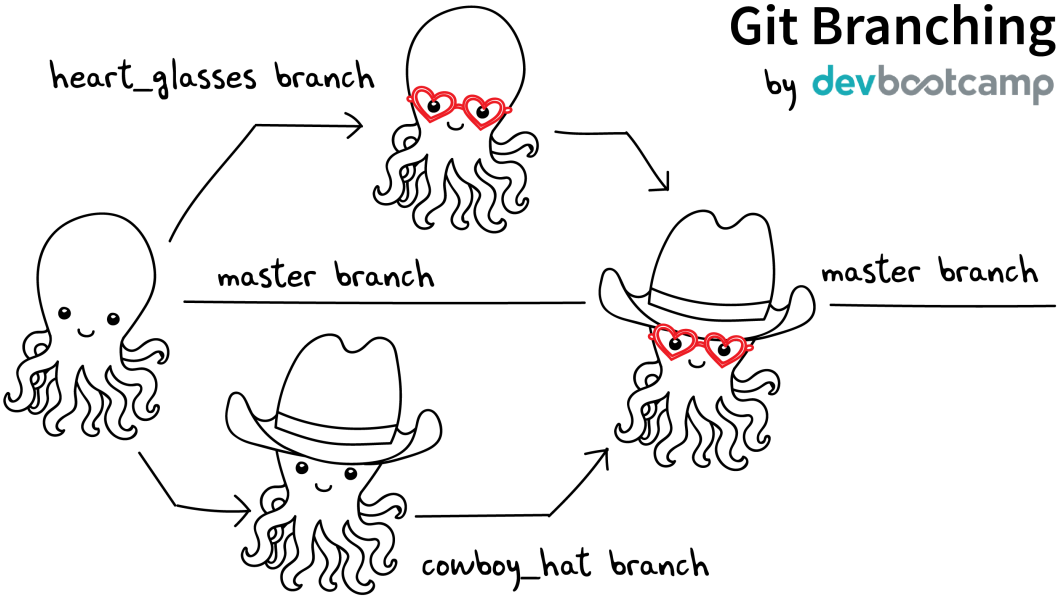
Add-commit-pull-push workflow

"I might not be alone in this."

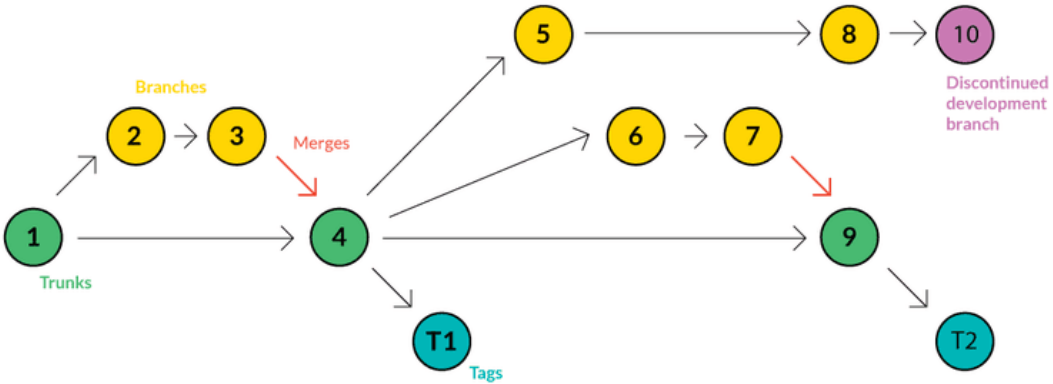
- Useful for collaboration
- Uses Git as common backup and file distributor
- allows for changes the remote repo

Live session: Git

Outlook: Git Advanced Workflows



Outlook: Git Advanced Workflows



Courtesy of The Webinerds [3].

Writing Meaningful Commit Messages

A well-crafted Git commit message is the best way to communicate *context* about a change to fellow developers (and indeed to your future self). A diff will tell you *what* changed, but only the commit message can properly tell you *why*.

- By convention (and tool support) the first line in a commit message is the subject of a commit. The subject is a short sentence describing the commit.
- The subject is followed by a blank line and then comes the body. The body is a detailed description of the commit used to explain *what* and *why* vs. *how*
- Not every commit requires both a subject and a body. Sometimes a single line is fine.

Courtesy of Chris Beams [11].

Git Dos and Don'ts

Respecting the following points will ease your and your collaborators' lives:

Git Dos and Don'ts

Respecting the following points will ease your and your collaborators' lives:

- Track only source code
  - Don't track auxiliary files created by the compiler or IDE
  - Use `.gitignore` to exclude unwanted files in project folder
  - Documentation may be added (preferably use GitLab's Wiki)
- BLOBs (Binary Large OBjects
  - Don't track BLOBs (Binary Large OBjects), use references/links
  - Use clear text file format if possible (e.g., Simulink's .mdl instead of .slx)

Git Dos and Don'ts

Respecting the following points will ease your and your collaborators' lives:

- Track only source code
  - Don't track auxiliary files created by the compiler or IDE
  - Use `.gitignore` to exclude unwanted files in project folder
  - Documentation may be added (preferably use GitLab's Wiki)
- BLOBs (Binary Large OBjects
  - Don't track BLOBs (Binary Large OBjects), use references/links
  - Use clear text file format if possible (e.g., Simulink's .mdl instead of .slx)
- Keep repositories up-to-date
  - Commit, push and pull often
  - However, commit only working code (stash otherwise)
- Have clear goals
  - Keep branches simple and with a specific purpose
  - Focus on one task at a time (e.g., a new feature, a bugfix)

Git Dos and Don'ts

Respecting the following points will ease your and your collaborators' lives:

- Track only source code
  - Don't track auxiliary files created by the compiler or IDE
  - Use `.gitignore` to exclude unwanted files in project folder
  - Documentation may be added (preferably use GitLab's Wiki)
- BLOBs (Binary Large OBjects
  - Don't track BLOBs (Binary Large OBjects), use references/links
  - Use clear text file format if possible (e.g., Simulink's .mdl instead of .slx)
- Keep repositories up-to-date
  - Commit, push and pull often
  - However, commit only working code (stash otherwise)
- Have clear goals
  - Keep branches simple and with a specific purpose
  - Focus on one task at a time (e.g., a new feature, a bugfix)

Git Dos and Don'ts

Respecting the following points will ease your and your collaborators' lives:

- Track only source code
  - Don't track auxiliary files created by the compiler or IDE
  - Use `.gitignore` to exclude unwanted files in project folder
  - Documentation may be added (preferably use GitLab's Wiki)
- BLOBs (Binary Large OBjects
  - Don't track BLOBs (Binary Large OBjects), use references/links
  - Use clear text file format if possible (e.g., Simulink's .mdl instead of .slx)
- Keep repositories up-to-date
  - Commit, push and pull often
  - However, commit only working code (stash otherwise)
- Have clear goals
  - Keep branches simple and with a specific purpose
  - Focus on one task at a time (e.g., a new feature, a bugfix)
- Be tidy right from the start
  - Clear code and folder structure
  - Don't reformat files if not necessary

DEFINITELY DON'T!!11!

DEFINITELY DON'T!!11!

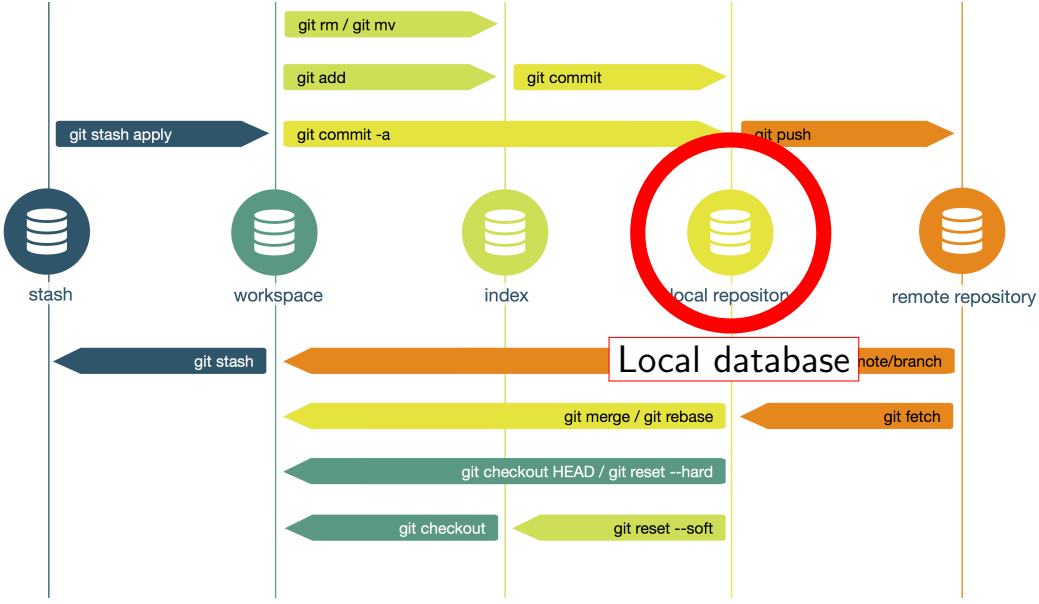
git commit --amend; git push -f

From here on: Homework and Lookup.

Git Data Transport

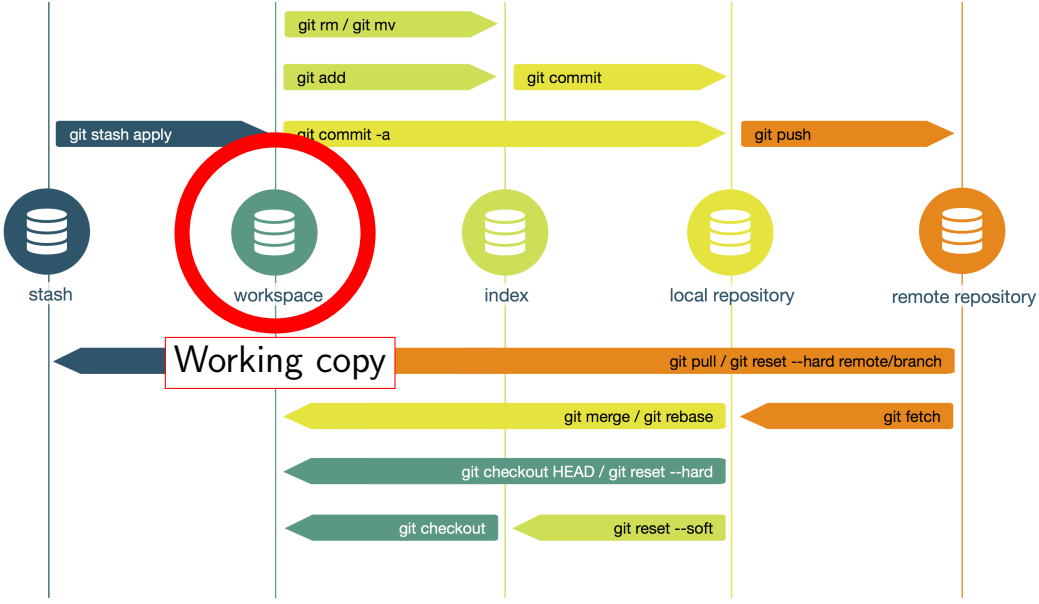
Courtesy of Patrick Zahnd [7].

Git Data Transport



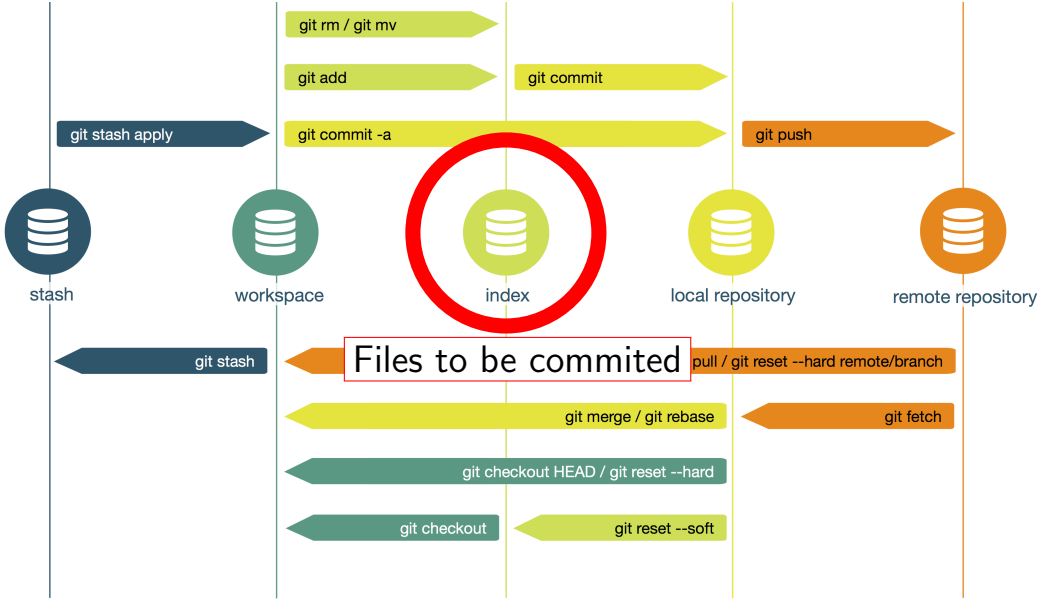
Courtesy of Patrick Zahnd [7].

Git Data Transport



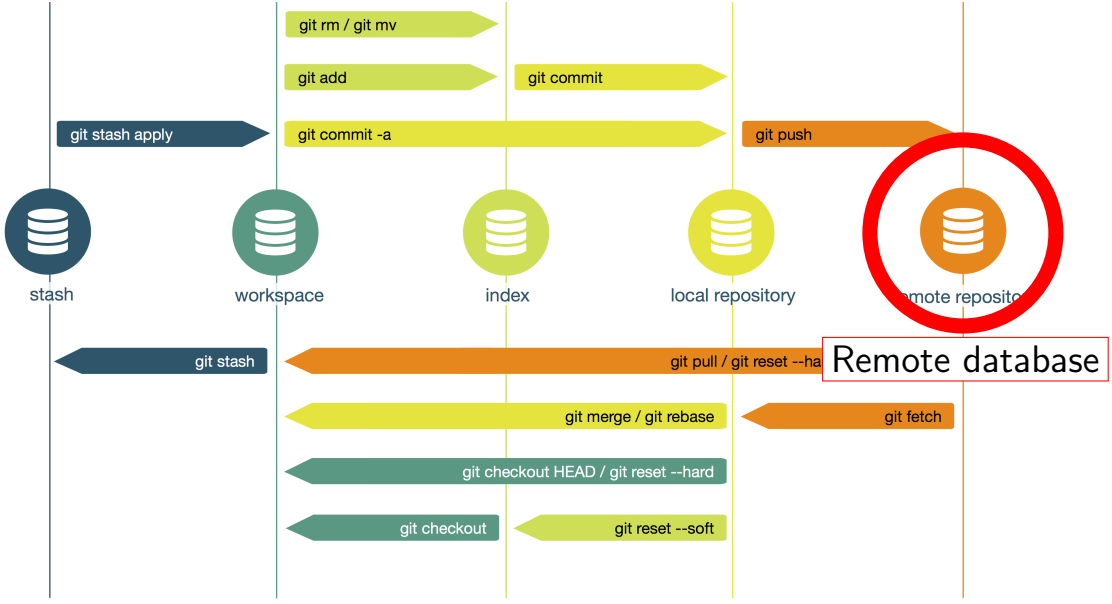
Courtesy of Patrick Zahnd [7].

Git Data Transport



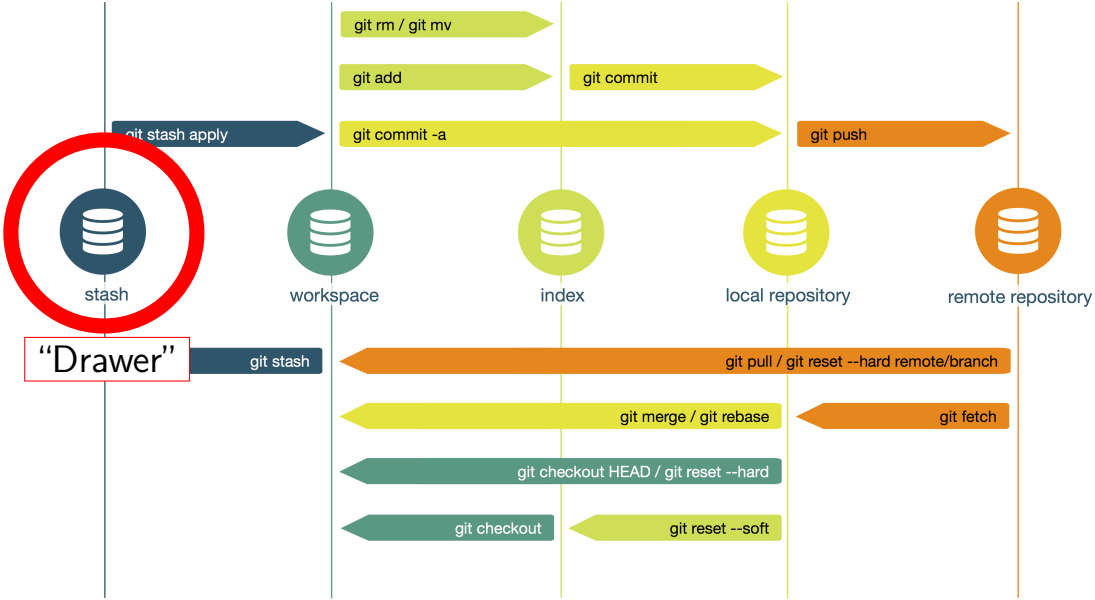
Courtesy of Patrick Zahnd [7].

Git Data Transport



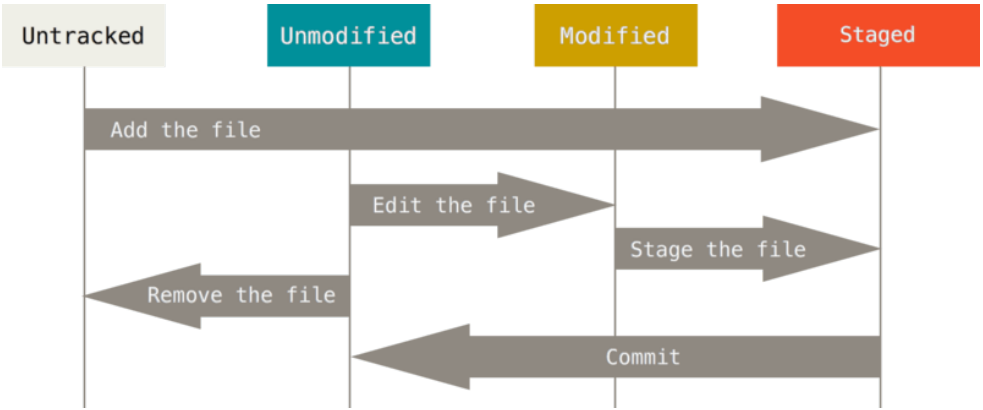
Courtesy of Patrick Zahnd [7].

Git Data Transport



Courtesy of Patrick Zahnd [7].

File Lifecycle



Courtesy of the Git reference manual [2].

Accessing the Git shell

Getting the Git client

- Linux/Mac
  - Git and graphical tools are included in the repositories of most distributions
  - Debian/Ubuntu/Mint: `$ sudo apt install git`
  - Arch/Manjaro: `$ sudo pacman -S git`
  - Mac: `$ git --version` (MacOS will prompt you to install it)
- Windows
  - Go to www.git-scm.com/download/win and download the client

Linux/Mac users

- Git is integrated into the bash shell

Windows users only

- Git is integrated into Windows CLI if `PATH` variable is set
- **Git Bash (supplied by Git installer) for Windows works always**

GitLab Server

The Department's GitLab server allows you or your workgroup to host your own projects.

- Log in to Wind GitLab server, accessing the course group on https://gitlab.windenergy.dtu.dk/python-at-risoe/spp-2023; use your DTU credentials
- Create new project:
  - On the main page, click *New Project*
  - Choose name and visibility level
  - Add collaborators by clicking on *Settings* → *Members*
- Contribute to existing project:
  - Select project on GitLab main page or follow invitation link
  - Clone the project on your computer
- Follow the command line instructions on the project overview page

Setting up Git

Fresh Git installations need to be configured with user information.

- Set the name and email that will be attached to commit actions

```
$ git config --global user.name "User Name"
$ git config --global user.email "user@email"
```

- Set default editor for commit messages and file diffs

```
$ git config --global core.editor editor_name
```

- Enable user interface colouring

```
$ git config --global color.ui auto # auto is default setting
```

Exercise 0: Setting up Git

1. Type `git version` to see if Git is working
2. Set user name
3. Set user email
4. Set default editor if you don't like the current one

Getting a Git Repository – Initialising new Project

Create a new repository when you start a new project or decide to have an existing project version controlled.

- Change into project directory (create one if it doesn't exist):
  - Linux:

```
$ cd /home/user/my_project
```

  - Mac:

```
$ cd /Users/user/my_project
```

  - Windows:

```
$ cd /c/user/my_project
```

- Initialise local repository in project folder:

```
$ git init
```

Adding Files and Ignore Files

Git needs to know which files are supposed to be version controlled.

- Adding (tracking/staging) specific file

```
$ git add README.TXT
```

- Use wildcard * to add all files that match pattern

```
$ git add *.TXT # Add all files ending on .TXT
$ git add *     # Add all files
```

- The file `.gitignore` in the project root folder allows to create file listing patters to prevent files from being tracked

```
# ignore all .a files
*.a
# but do track lib.a, even though you're ignoring .a
# files above
!lib.a
```

Remove and Move Files

Sometimes it is necessary to untrack or move tracked files.

- Remove files

```
$ git rm README.TXT
$ git rm *.TXT # Remove all files ending on .TXT
```

- Move files

```
$ git mv file_from file_to # Can also be used for renaming
```

Committing Files

When modifications to the project reached the desired outcome (e.g., a new feature or a bugfix), synchronise your working directory with the local repository by committing the changed files.

- Committing modified files to the local repository

```
$ git commit # Opens external editor to type commit message
```

- Commit messages can be added as an optional argument

```
$ git commit -m "This is what happened"
```

- The staging area can be skipped using the -a option

```
$ git commit -a -m "Commit all tracked and modified files."
```

Undoing Things

When the previous commit was faulty or incomplete, or the changes to a file need to be reverted, things can usually be undone.

- The `--amend` option replaces the last commit

```
$ git commit -m "Some message" # Commited too early
$ git add forgotten_file
$ git commit --amend
```

- Files staged for commit can be unstaged

```
$ git reset HEAD file_to_unstage
```

- Modified files can also be reset to their most recent version

```
$ git checkout -- file_to_reset # Replace modified file
```

Status Information

Information about the project and files can be obtained in several ways.

- Check the status of Your Files

```
$ git status     # Verbose status output
$ git status -s # --short gives compressed output
```

- Show difference between files (=*patches*)

```
$ git diff           # Changes that have not been staged
$ git diff --chached # What has been staged so far
```

- List commits made in repository in reverse chronological order

```
$ git log    # "-N" option limits output to last N entries
$ git log -p # --patch shows differences between commits
```

- Graphical history

```
$ git log --graph # Text-based viewer
$ gitk            # Graphical viewer
```

Getting a Git Repository – Cloning Exisiting Project

If you want to use to existing projects or even contribute to them, clone them locally on your computer.

- Change into directory where the project folder should be located (e.g., `user` in previous slide)
- Clone project from remote repository:

```
$ git clone https://github.com/libgit2/libgit2
```

- Specify targeted directory name if you don't want the default name (here: `libgit2`):

```
$ git clone https://github.com/libgit2/libgit2 my_project
```

Add and Remove Remote Repositories

Collaboration between developers is enabled by remote repositories, which are locally identified over acronyms.

- Show which remote servers are configured

```
$ git remote # Shows "origin" if retrieved with clone command
```

- New remotes can be added under the acronym `repo_name` using

```
$ # Firewall-friendly
$ git remote add repo_name https://url_to_repository
$ # Uses private key
$ git remote add repo_name git@url_to_repository
```

- Reference to remote repositories can of course also be removed

```
$ git remote remove repo_name
```

Fetch and Push to Remotes

Pushing and pulling data between local and remote repositories synchronises state and historical information and makes them visible to other developers.

- Fetch all remote information and add it to local repository

```
$ git fetch repo_name
$ git fetch https://url_to_repository
```

- Merge the updated information in the local repository with your working directory

```
$ git merge origin/master # If cloned from remote repository
```

- A shortcut to `fetch` and `merge` is `pull`

```
$ git pull origin/master # If cloned from remote repository
```

- Share local developments by pushing to remote repository

```
$ git push <remote> <branch>
$ git push origin master # If cloned from remote repository
```

Exercise 1: Clone project and make modifications

1. Gather in your teams
2. Decide on who is user A and who is B
3. Clone your own project set up in https://gitlab.windenergy.dtu.dk/python-at-risoe/spp-2023, where `N` is your team number
4. You get six files:
   - `main.py`: Main executable programme, needs 2 floating point numbers as arguments

     ```
     $ python main.py 1.23 4.56 # Example
     ```

   - `fun*.py`: Four additional modules
   - `README.md`: Documentation for GitLab
5. Run `main.py` – You will receive 'None' and/or exceptions as results for all tasks

Exercise 1: Clone project and make modifications (cont.)

7. Open file `main.py` and follow instructions for exercise 1
8. Run `main.py` when done – Task 1 should show expected results for the individual user functions (square/sum of inputs)
9. When functions are correctly implemented...
    1. Type `git status` to see changes
    2. Add changed files to index and commit
    3. Pull from remote – nothing should happen/change
    4. Push to remote repository
    5. Pull from remote – other user's changes will be pulled in
    6. User B: Confirm merge message
    7. Both user should arrive at same working state
10. Run `main.py` – Task 1 should show expected results for both user functions

Understanding `origin`, `master` and `HEAD`

- **HEAD** is a keyword that points to a specific commit in your local repository. It points to the "current" revision which is most often also the latest revision in that branch. Therefore, when you git reset or git revert to HEAD to you (usually) change to the "latest revision". E.g. doing a `$ git reset --hard` to HEAD reverts all changes to the latest revision.
  `$ git checkout branch_name` moves HEAD to another branch.
- **master** is a name that by convention is given to the main branch of your repository. What that means is defined by your project, but usually the master branch is the definitive one in which all finished changes goes into. When work in a branch is done, it is merged into master.
- **origin** refers to the address of the main remote repository. This address is the one used by default when you push and pull. You can change the address of origin anytime you want.

Resolve Merge Conflicts

- Merge fails because Git cannot resolve file (e.g., `index.html`)

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

- A more detailed explanation can be retrieved with `git status`

```
$ 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")
```

Resolve Merge Conflicts

- Open conflicting file(s) and look for sections marked with <<<<<<<|=======|>>>>>>>

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

- Correct issues manually and remove markers

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

Resolve Merge Conflicts

- Use `mergetool` to graphically resolve issues

```
$ git mergetool
```

- Stage files to mark issues as resolved for Git before committing again

```
$ git add index.html && git commit
$ git commit -a
```


BRANCH YOURSELVES
MERGE CONFLICTS ARE COMING
imgflip.com

Exercise 2: Collaborate on Same Code

1. Open file `main.py` and follow instructions for exercise 2
2. Both users implement function and commit
3. A pulls and pushes changes – Should work fine
4. B tries to pull and push – Will get a merge error during pull
5. B has to follow instructions given by git to resolve error locally
    1. Open conflicting file(s) in text editor and look for conflicting section(s)
    2. Repair code
    3. Remove <<<, === and >>> markers and save
6. Add, commit (confirm merge message) and push – Error should be gone
7. Both users need to pull to arrive at same state – Another automated merge may pop up
8. Run `main.py` – Task 2 should show expected results for both user functions
9. Repeat exercise with switched roles

Work With Branches

Branching is a powerful mechanism to keep the development of bigger project clean and traceable. See here for a good introduction: www.youtube.com/watch?v=8_mHSdCkv3s.

- Create a new feature branch

```
$ git branch new_branch # Creates but doesn't switch to branch
```

- List branches

```
$ git branch    # Lists local branches
$ git branch -r # --remote lists remote branches
```

- Switch to branch

```
$ git checkout branch_name # You now work on this branch
```

Push and Pull on Remotes Branches

Remote branches are handled exactly the same way as in the the simple `origin/master` case before. In most cases, `reponame` remains `origin`.

- Fetch remote information, add to local repository and merge with working directory

```
$ git pull # The quick way if local and remote are linked
$ git pull repo_name branch_name              # If not linked
$ git fetch r_name && git merge r_name b_name # Does the same
```

- Share local developments by pushing to remote repository

```
$ git push #
$ git push # The quick way if local and remote are linked
$ git push repo_name branch_name # If not linked
```

- Link new local branch with remote repository

```
$ git git branch --set-upstream-to=repo_name/branch_name
$ git push -u repo_name branch_name # Link on first push
```

Merge Branches

Once the desired goal of a branch is reached, or new developments of a related branch are of interest, merging them combines their functionality.

- Merge feature branch B into master branch A

```
$ git checkout branch_A # Only if you're on any other branch
$ git merge branch_B    # Merges into branch_A
```

- Delete feature branch when no longer needed

```
$ git branch -d branch_name # --delete deletes branch
```

- Alternatively, rebase branch for getting a cleaner history for the sake of traceability

```
$ git checkout branch_B # Only if you're on any other branch
$ git rebase branch_A
```

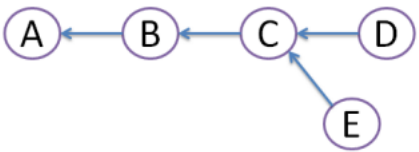
- Common use case for rebase: Clean up history
- Golden rule of rebasing: Never use on public branches!

Merge & Rebase Illustration

The following was adapted from user mvp's posting on Stackoverflow [9] to highlight differences.

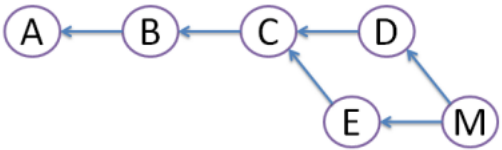
- Suppose originally there were 3 commits, A,B,C

- Then developer Dan created commit D, and developer Ed created commit E

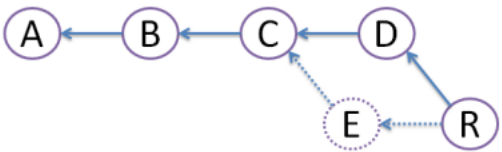


Merge & Rebase Illustration

- **Merge:** Both commits D and E are still here, but we create merge commit M that inherits changes from both D and E.



- **Rebase:** We create commit R, which actual file content is identical to that of merge commit M above. But, we get rid of commit E, like it never existed. Because of this obliteration, E should be local to developer Ed and should have never been pushed to any other repository. Diamond shape is avoided, and history stays nice straight line.



Stash Current Work

Switching between branches requires a clean branch without modifications, otherwise the modifications will be lost. This is typically achieved by conducting a commit, but sometimes you don't want to do that because the work is only half done. Instead, you can stash your changes and retrieve them once you want to continue the work.

- Push modifications since last commit to stack

```
$ git stash
```

- Last what's on the stack

```
$ git stash list
```

- Apply stashed modifications

```
$ git stash apply           # Applies last stash
$ git stash apply stash@{2} # Applies specific stash (here: 2)
```

Exercise 3: Add Feature Branches

1. Open file `main.py` and follow instructions for exercise 3
2. Get files from https://data.deic.dk/shared/GitWorkshopFiles if you haven't already done so
3. Both user create their own feature branches
4. Add `fun3A.py`/`fun3B.py` to respective branches
5. Put additional code snippets from text file `FunctionsToInclude` into `main.py`
6. Commit, pull, push, and pull again
7. Both user should be able to look at each others branches (no pun intended) – Try it out!
8. Each user should now merge their work into the master branch
9. Ideally, everything runs smoothly. Otherwise, fix merge errors like in previous exercise

Git Data Transport Recap

Courtesy of Patrick Zahnd [7].

References

1. Wikipedia on VCS: https://en.wikipedia.org/wiki/Version_control
2. Git homepage: https://git-scm.com
3. VCS overview: https://webinerds.com/version-control-systems-keep-your-code-in-order
4. Git tutorial: https://www.tutorialspoint.com/git/git_tutorial.pdf
5. XKCD on Git: https://xkcd.com/1597/
6. Git file stages: https://archaeogeek.github.io/foss4gukdontbeafraid/git/stages.html
7. Git data transport: http://patrickzahnd.ch/blog.html#gittransport
8. CEE/ELEKTRO GitLab: http://git.elektro.dtu.dk
9. Merge vs. rebase: https://stackoverflow.com/a/16666418/8362807
10. Cheat sheet: https://services.github.com/on-demand/downloads/github-git-cheat-sheet.pdf
11. Meaningful git commits: https://chris.beams.io/posts/git-commit

If you think all that stuff is complicated...

See for yourself what Git's inventor Linus Torvalds says:

If you think all that stuff is complicated...

See for yourself what Git's inventor Linus Torvalds says: