

Git

Robot Programming Course
2015-07-29

Daniele E. Domenichelli
`daniele.domenichelli@iit.it`

Version Control

- System that records changes to files.
- Allows you to restore a specific versions later.
- A better option than copying files with a different name or into another directory.

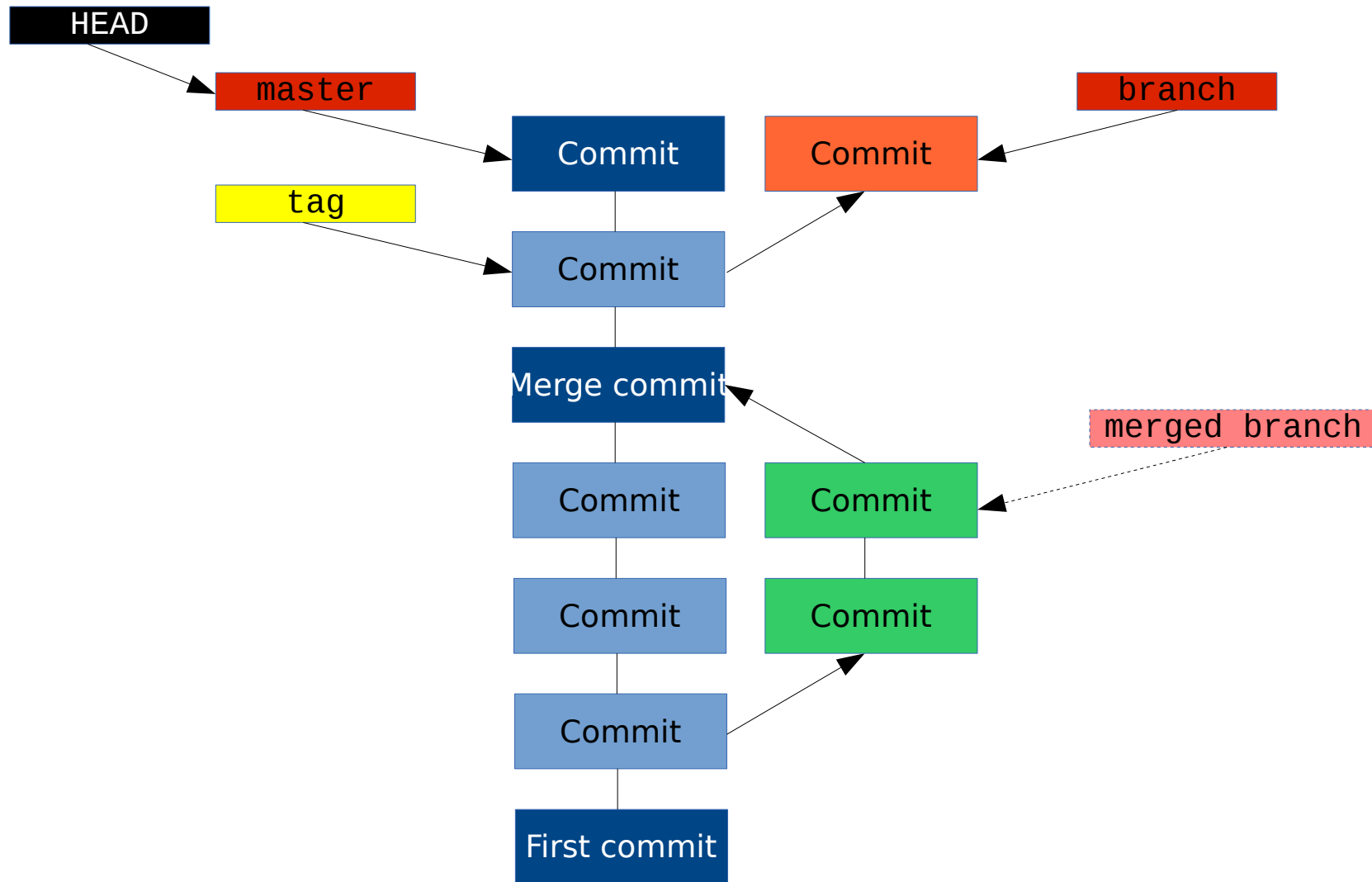
Git Brief History

- Linux Kernel
 - 1991–2002 Patches
 - 2002 BitKeeper
 - 2005 Linus Torvalds starts developing Git to host the kernel

Git Design

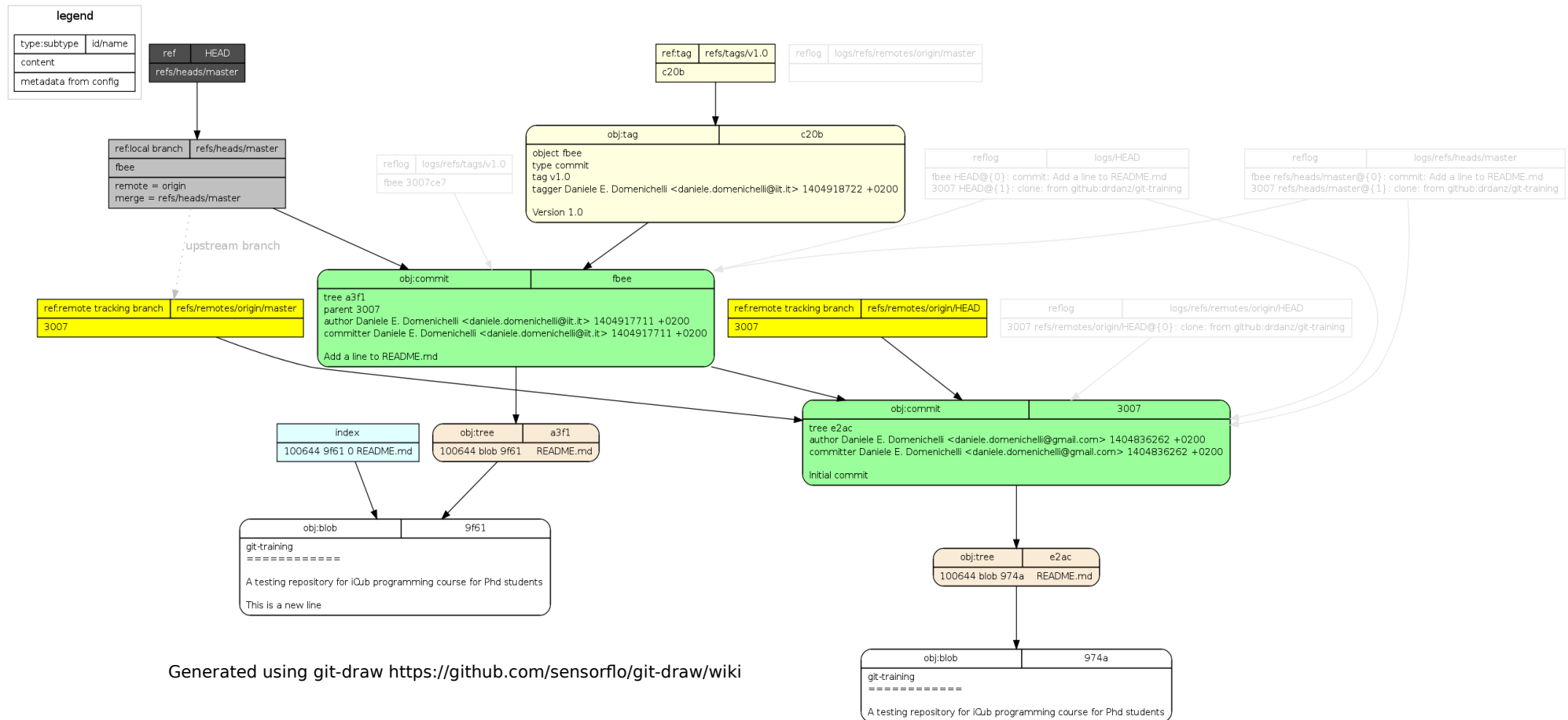
- Fully distributed
- Fast
- Strong support for non-linear development
- Able to handle large projects
- Cryptographic authentication of history
- Save “files” and not “patches”
- Is not optimized to handle binary files (but there are extensions for this).

Git Repository History Example



- “HEAD” is a reference to the commit that you checked out in your working tree
- A “merge commit” has two parents, the “first commit” has none.

Git Objects



- “Rounded corners” boxes are objects, the other boxes are references (names that you can use to point to one object).
- White objects are **blobs** (files)
- Pink objects are **trees** (files that describe the content of a directory)
- Green objects are **commits**
- Light Yellow objects are **tags**

LOCAL WORKFLOW

Git Repository Structure

Repository
Database

The History

Index

Staging area

Working Tree

Files you can edit

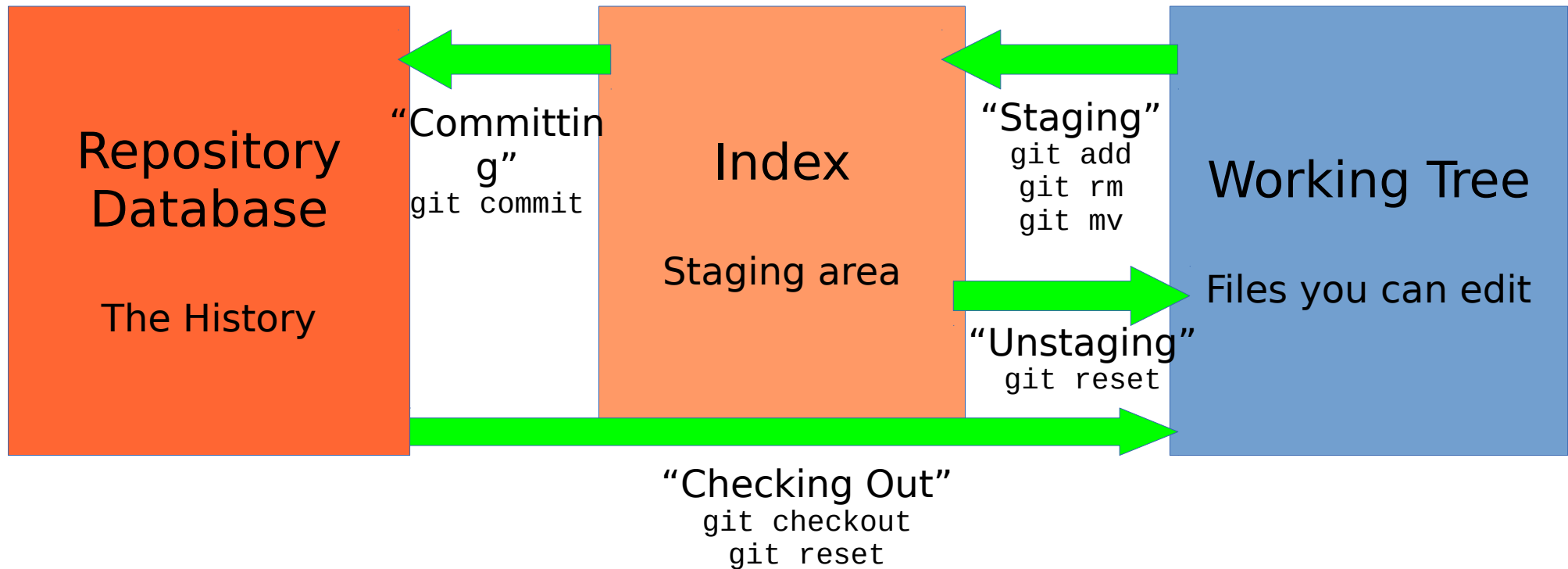
Git “Bare” Repository Structure



Repository
Database

The History

Local Workflow



Install Git

- Debian/Ubuntu

- `sudo apt-get install git`

- RHEL/Fedora/Centos

- `yum install git`

- Windows

- Git for Windows: <https://git-for-windows.github.io/>

- MacOS

- Installed with Xcode Command Line Tools

- GUIs and other tools

- TortoiseGIT: <https://tortoisegit.org/>

- Atlassian SourceTree: <https://www.atlassian.com/software/sourcetree/>

- GitHub Desktop: <https://desktop.github.com/>

- A lot more can be found here:

- <https://git.wiki.kernel.org/index.php/InterfacesFrontendsAndTools>

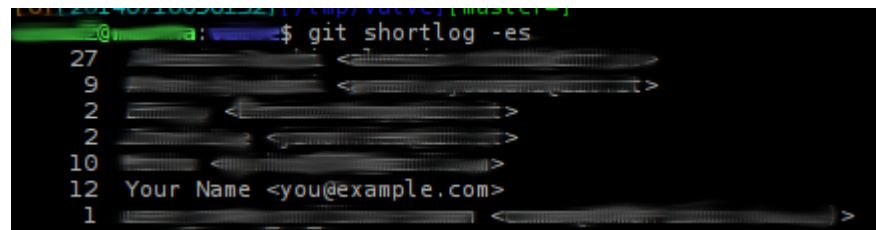
Identity

- The first thing you should do is to set your user name and e-mail address.

```
git config --global user.name "John Doe"
```

```
git config --global user.email "john.doe@example.com"
```

- Please, please, please replace “John Doe” and “john.doe@example.com” with **your real name and email!!**

A terminal window showing the output of the command 'git shortlog -es'. The output lists commit counts for various authors, with the last entry being '12 Your Name <you@example.com>'. The terminal has a dark background with light-colored text.

```
27  
9  
2  
2  
10  
12 Your Name <you@example.com>  
1
```

Creating a git repository

- `mkdir <folder>` (on Linux)
- `cd <folder>`
- `git init`
 - Now you have a fully working git repository.
 - A lot easier than creating an SVN repository.
 - Don't forget that for now it's just on your computer.

Status

- `git status`
 - Shows staged, unstaged and untracked files
 - Do not be afraid to use it after every command
 - Better with colours enabled
- Now try to create a file and see what changes in the output of the command.
 - `touch README.md`
 - `git status`

Some Color

- If you use git mostly from the command line, colors are very useful, you can enable them by running:

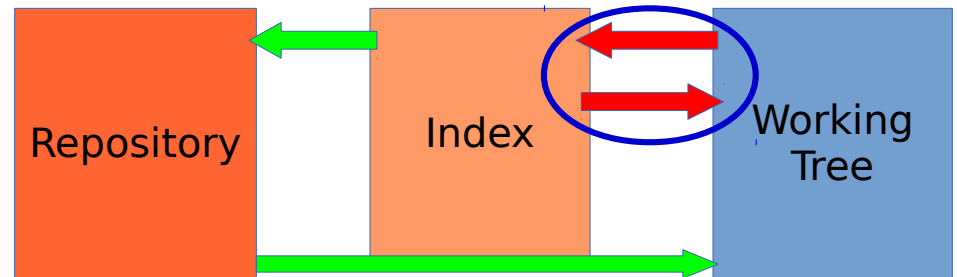
```
git config --global color.pager true
```

```
git config --global color.ui auto
```

- Status and diff are a lot more readable with colors.

Staging and Unstaging

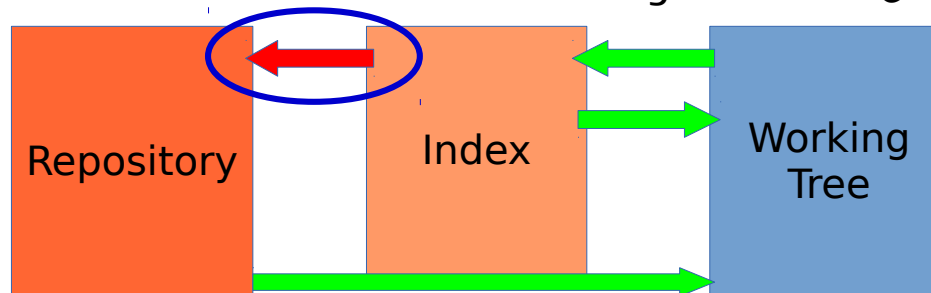
- `git add <file>`
- `git add .`
- `git reset <file>`
- `git reset`



- Nothing was committed yet!
- The “index” is updated
- Check what changes after each command using “`git status`”
- These commands work on the stage area, but also on the working tree:
 - `git rm <file>`
 - `git mv <old> <new>`

Committing

- Svn style commits (bad practice)
 - `git commit -a -m "Log message"`
 - -a = all modified and trackable files in the working directory
 - `git commit [--] <file>...`
- Staged files only
 - `git commit -m "Log message"`
- Commit on a shared machine, where your name and email address are not configured (e.g. on the iCub computers)
 - `git commit --author="John Doe <john.doe@example.com>"`

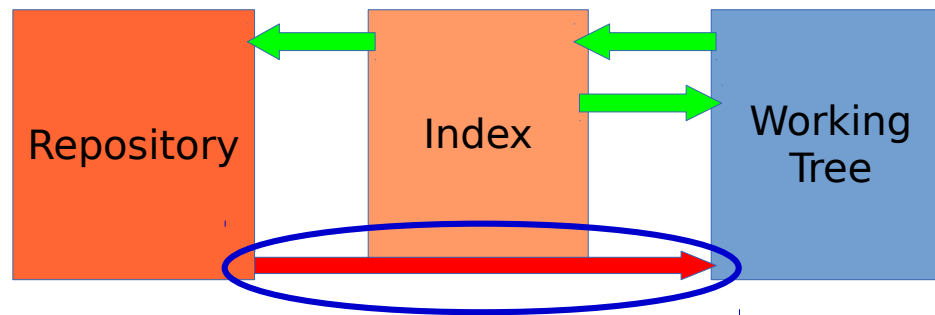


Log

- `git log`
 - Shows the log, starting from HEAD
- `git log <revision>`
 - Shows the log, starting from <revision>. Commit can be a hash or a name associated to a commit (for example the name of a branch, a tag or HEAD)
- `git log [--] <path>`
 - Limit the log to the changes to a file or directory
- `git log <commit1>..<commit2>`
 - Limit the log to the changes since <commit1> and until <commit2>
 - Can be any “revision range” (see `man 7 gitrevisions`)
- `git log --since="2 weeks ago" --until="yesterday"`
- `git log --oneline --graph --decorate --all`

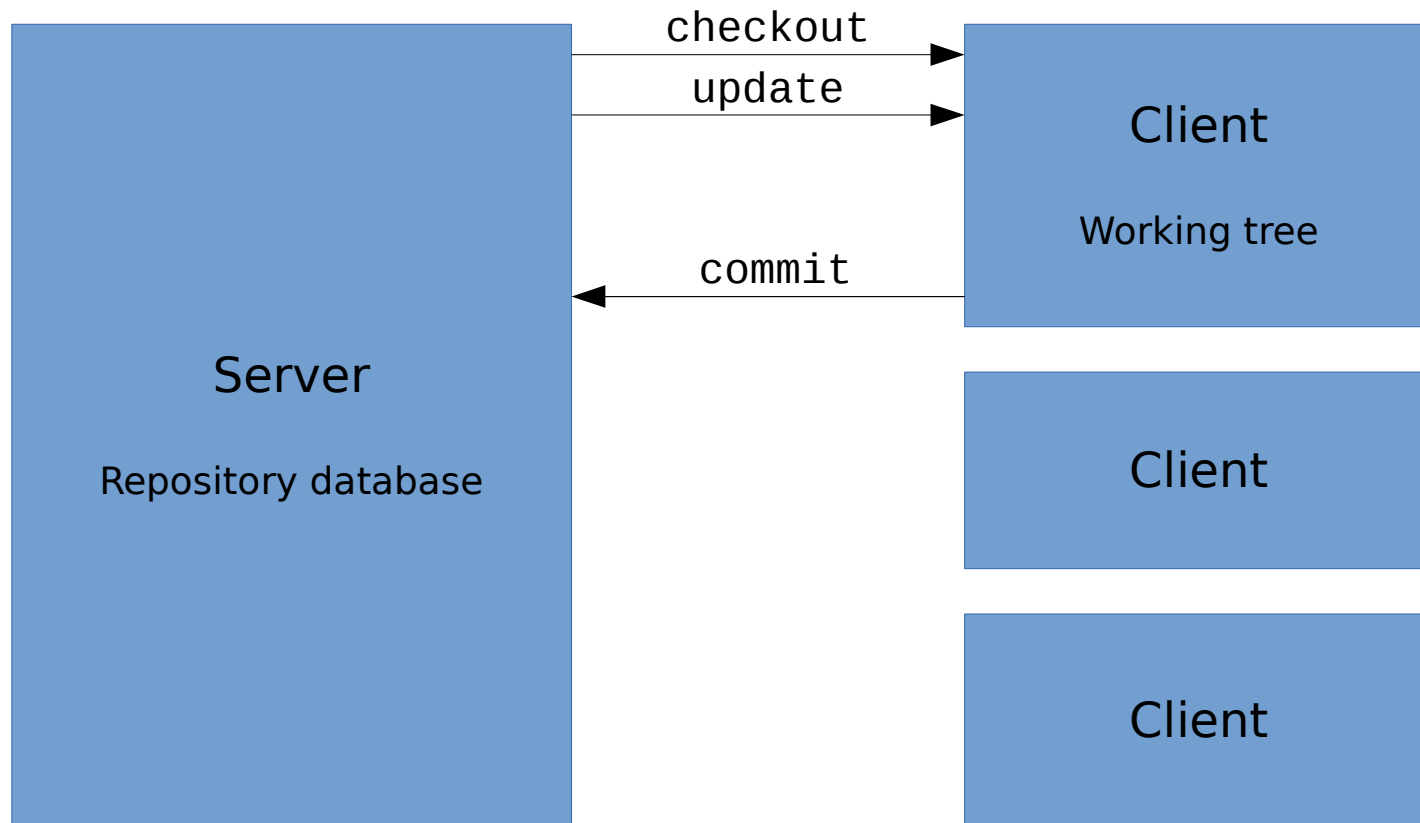
Checking Out a Commit or a Branch

- `git checkout <commit>`
 - If you have some changes to files that would be overwritten when switching branch, the operation fails, and does not try to do anything



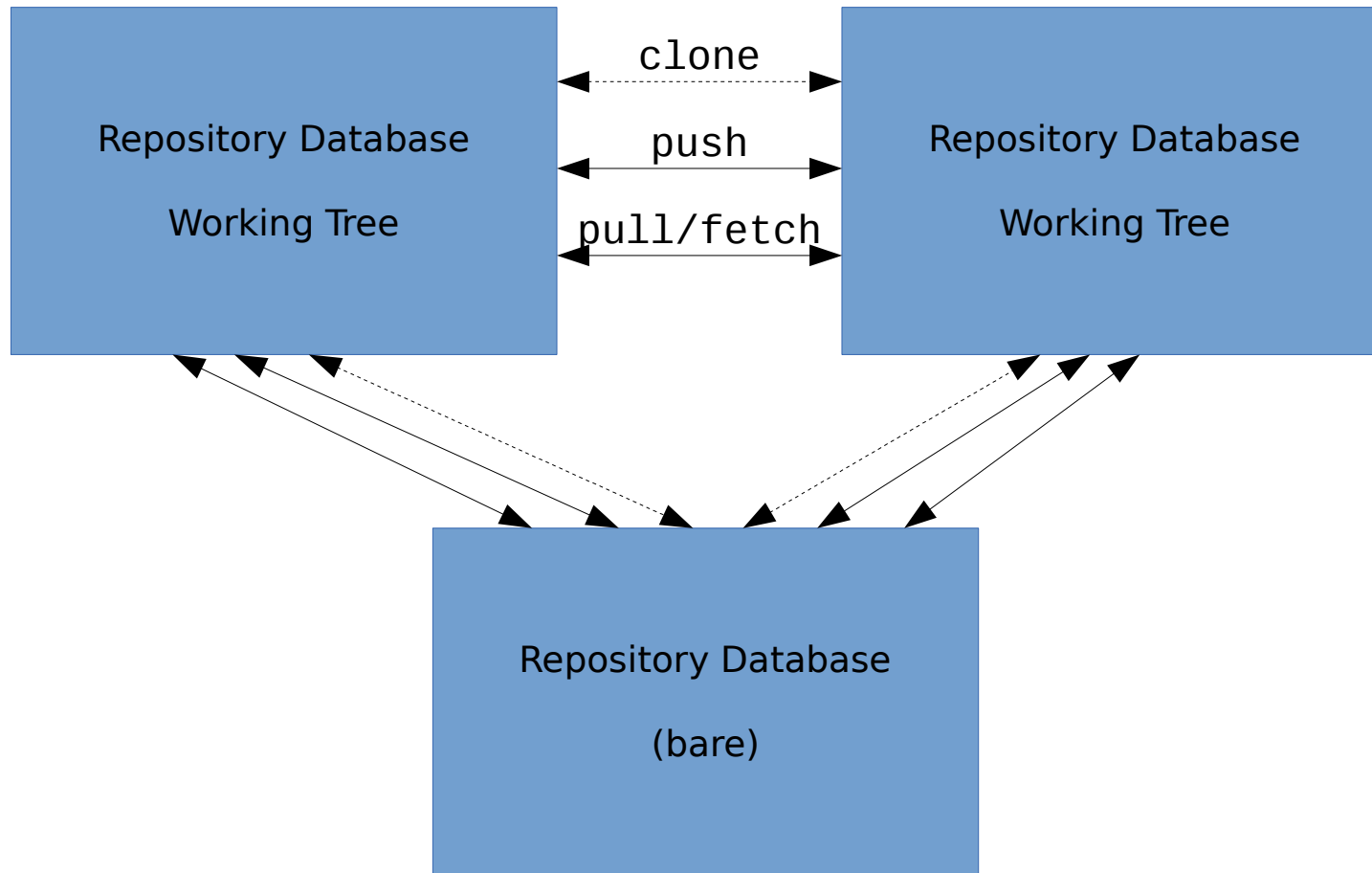
REMOTE WORKFLOW (with GitHub)

Centralized SCM (Subversion, CVS)



- Most commands require a server, including “informational” commands (log, diff, blame)

Decentralized SCM (Git)



- Any “clone” can be a remote, i.e. a server. for another clone.
- Most of the operations are performed locally.
- All the “clones” contain all the history of the repository.

Let's start...

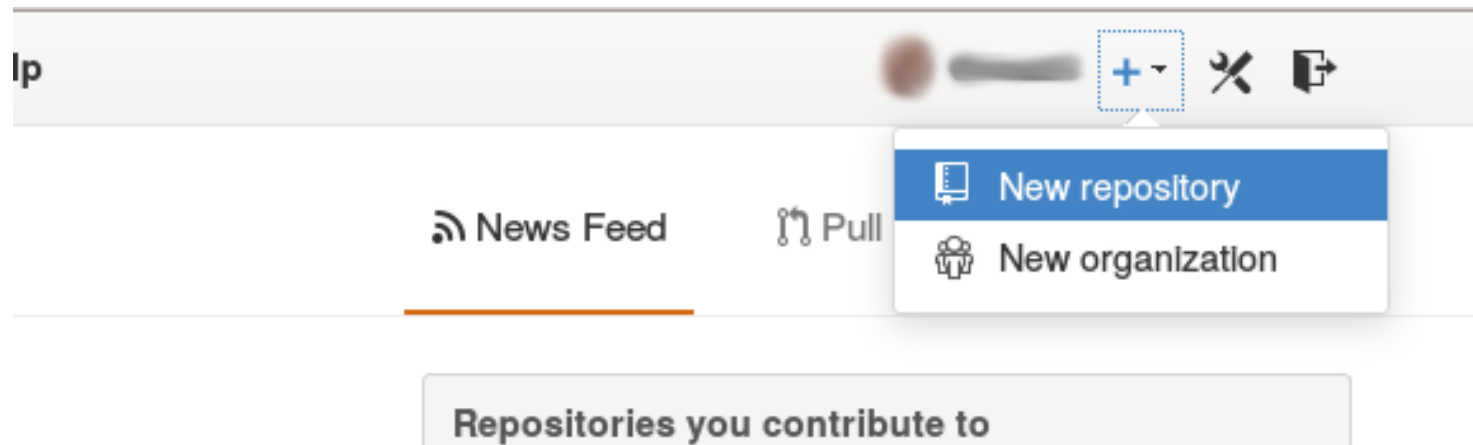
- For now we have just our local repository on our computer.
- Now we'll share this repository with the rest of the world using GitHub as a “central repository”



Working with remotes

- All the commands mentioned until now can be used locally, you don't need network or a server
- A remote can be
 - Another local repository
 - `file:///home/user/repo.git`
 - A remote repository (on a server, but also on your friend's computer)
 - `https://github.com/robotology/yarp.git`
 - `ssh://git@github.com:robotology/yarp.git`
 - `git://github.com/robotology/yarp.git`
- Remotes can be read/write or read only

Creating a Git Repository on GitHub



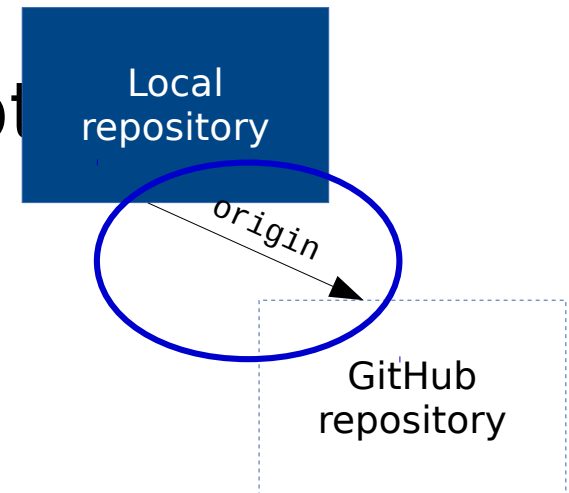
- Get an account if you don't have one
- Log in
- Click on “New repository”
- Follow the instructions
- Congratulations, you just created an empty repository on GitHub.

Local
repository

GitHub
repository

Configure Your Local Repository

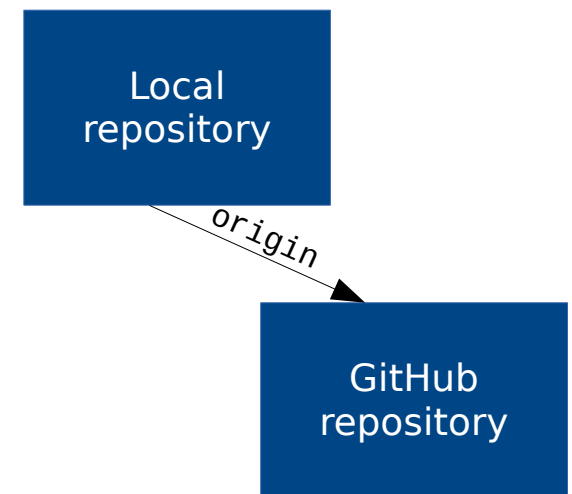
- Our local git repository needs to know that the new “remote” exists
 - `git remote add origin https://github.com/drdanz/git-training.git`
- Our local git repository now knows “origin”.
- This is the name that we assigned to the remote on GitHub, and we can refer to that name from now on.
- The “origin” remote is still empty



Now “Push” our Commits

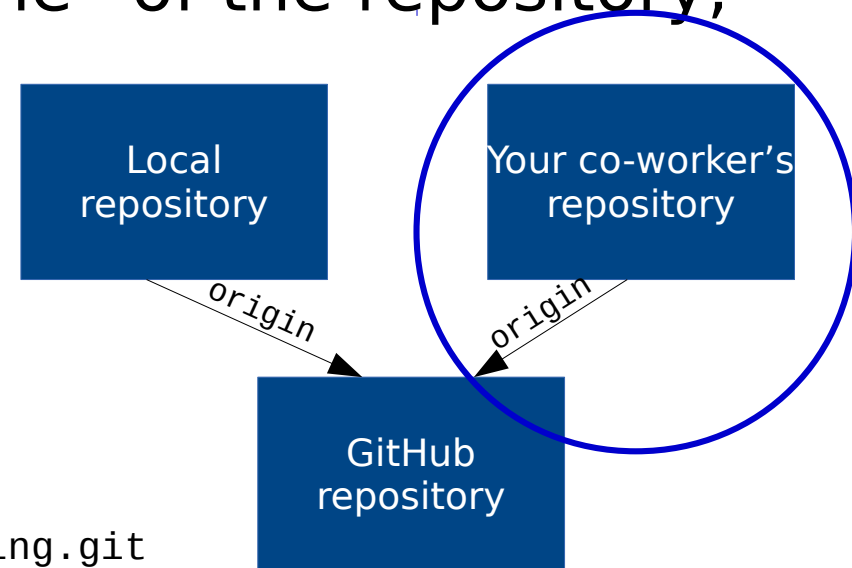
- `git push origin master`

- origin is the name of the remote where we want to push our changes.
- master is the name of the branch that we want to push.
- The master branch on GitHub repository now contains all the commits that we had locally in our local master branch.
- Now refresh the page on GitHub and see what happened.
- “master” is your local master branch.
- “origin/master” is the master branch on the origin remote (it can be different from the local one if you have some commits or if someone else made some commits on the remote)
- The server can reject the push for different reasons.



Let's Do Some Team Work

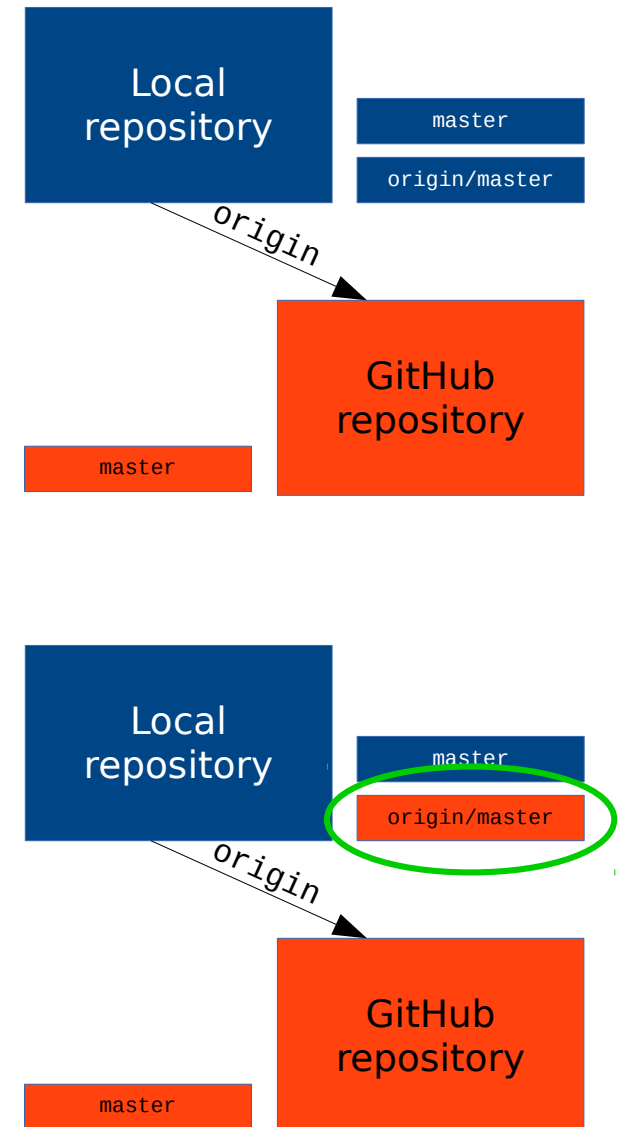
- We are ready to start working in team. Our co-worker wants to grab your files and do some changes
- First of all, he will make a “clone” of the repository, i.e. will download it locally.



- `git clone https://github.com/drduan/git-training.git`
- Your co-worker now has a full copy of the repository configured with an “origin” remote pointing to the GitHub repository.
- He can now push new commits on this repository.

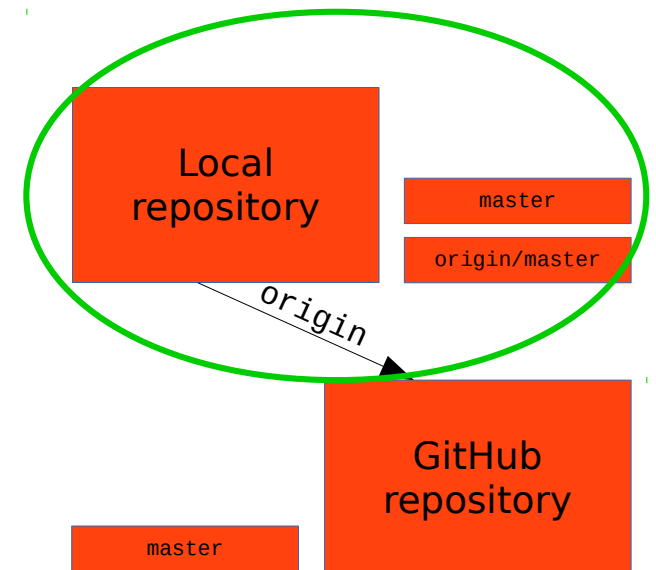
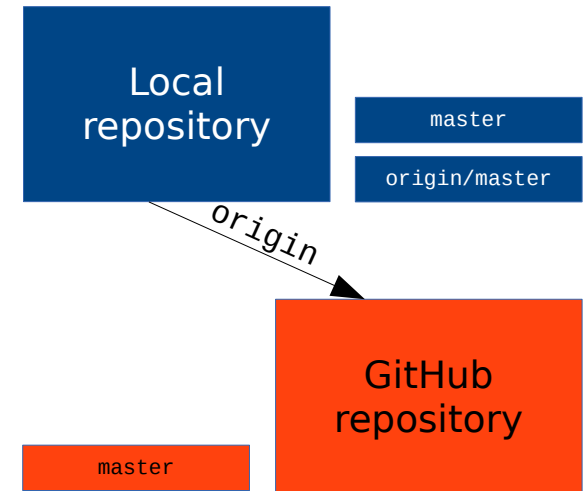
Check Changes on a Remote

- Someone pushed some new commit on the GitHub repository.
- `git fetch origin` (or just `git fetch`)
 - Retrieves all the new objects (commits, branches, tags) in the `origin` remote and saves them in your local repository, but does not apply anything to your working directory.

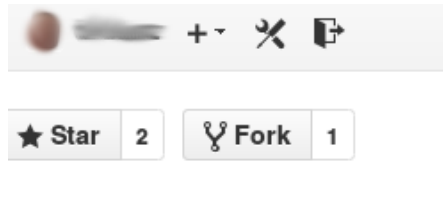


Pull Other People's Changes

- `git pull origin master`
 - Retrieves all the new commits in the master branch from the origin remote and merge them into your current branch (if you have local commits you get a new local merge commit)
 - Might result in a conflict
- `git pull --rebase origin master`
 - Same as previous command, but does not create an extra merge commit. Instead it rebases your local changes on top of the remote branch. (i.e. applies your local commits on top of origin/master)
 - Usually this is the recommended way to do, because the extra merge commits will make the history confused.

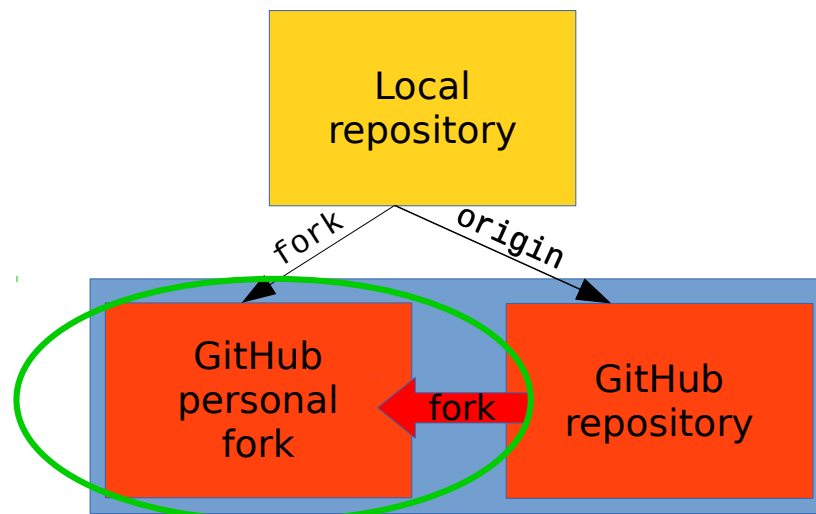


Forking a Repository on GitHub

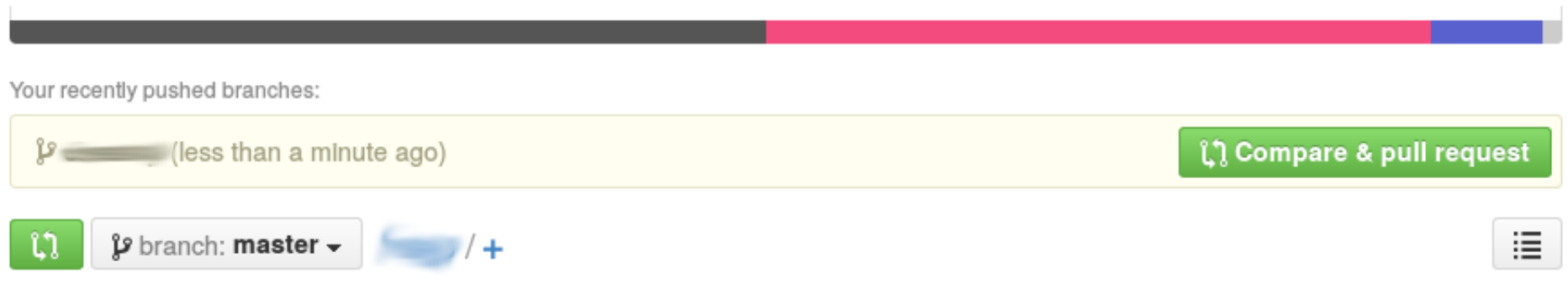


- Forking a repository means create another repository on GitHub that you can use to push your changes (for testing or for sharing with someone else, but avoiding to make them available to all the users of the main repository)
- Open the page of the repository you want to fork
- Click the fork button
- Add the new remote to your local clone

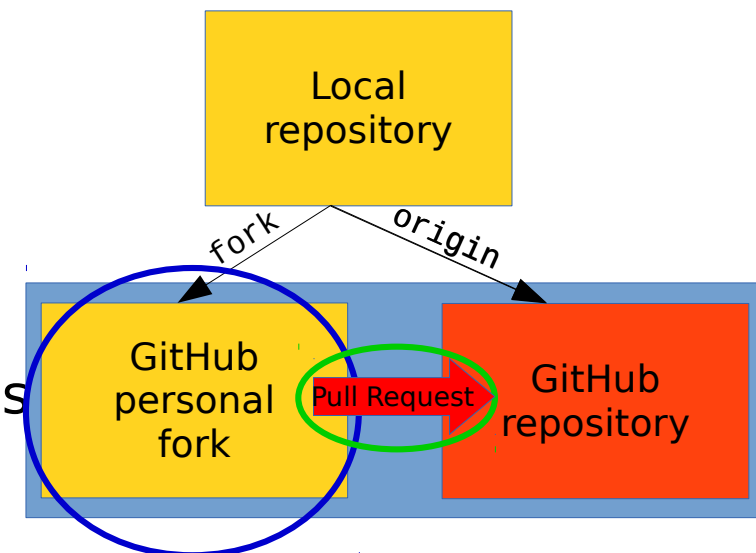
```
git remote add <remote name> git@github.com:<username>/<repository name>.git
```



Creating a Pull Request on GitHub

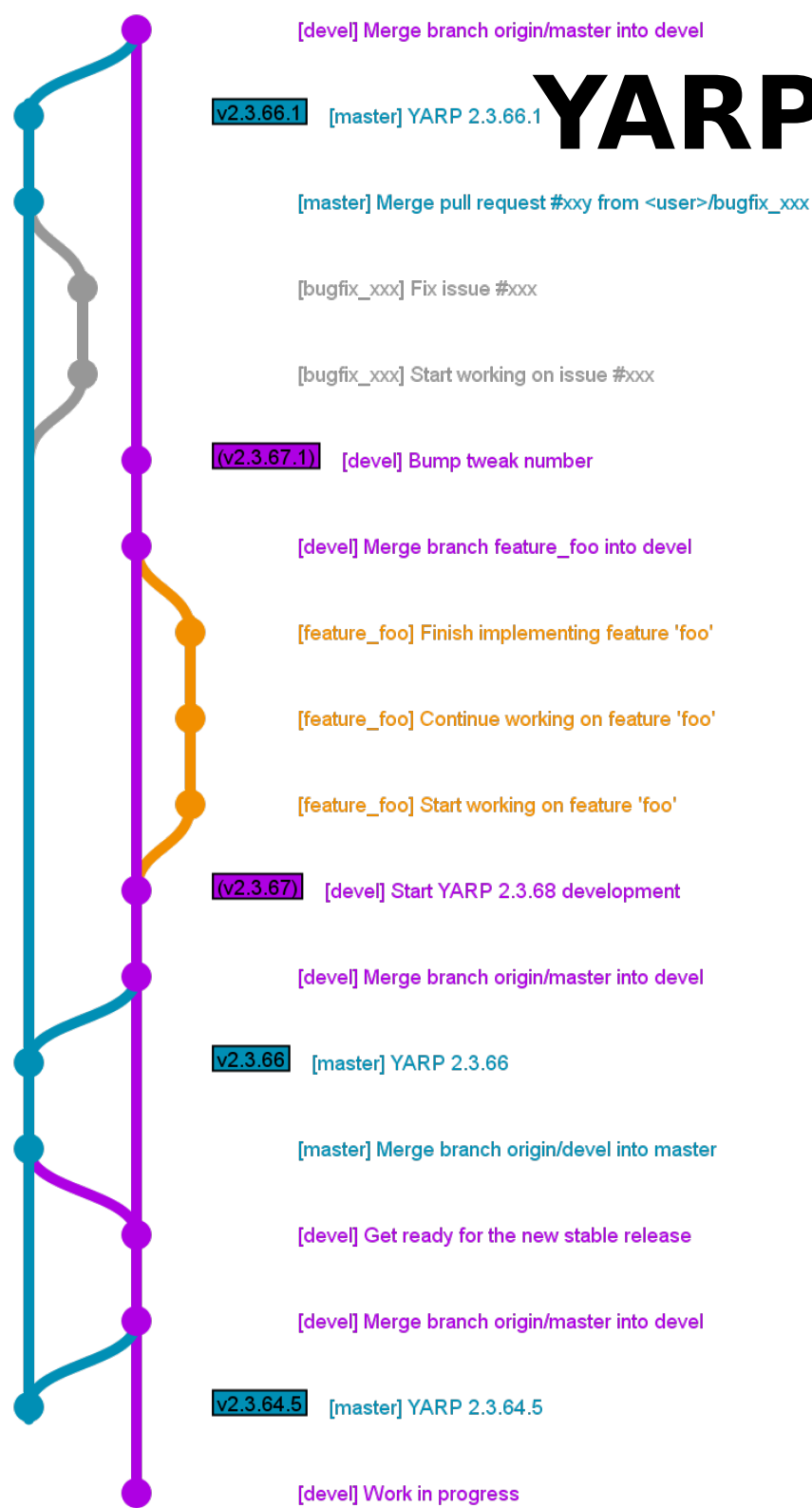


- Push your changes to a branch in your fork
 - `git push <remote name> <branch>`
- Open the main page of your fork
- Click the “Compare & pull request” button
- Write a comment and click “Create pull request”
- Now someone else can review your changes and merge the pull request, reject it or ask you to modify something.



OUR WORKFLOW

YARP Workflow



- 2 main branches
 - master (stable)
 - devel (unstable)
- master is merged into devel whenever we make a commit.
- devel is merged into master whenever we make a new “feature” release.
- We use branches and merge requests for bug fixes and new features.

GOOD PRACTICES

Good practices 1:

What am I pushing?

- Always ask yourself this question before pushing on a shared repository
- Use `git log (<remote>/<branch>..
branch>)` and gui tools (`git gui`, `gitk`, `gitg`, `qgit`, `git cola`) to ensure you are not pushing useless stuff
- Get rid of useless commits and merges using `git rebase`
- Review your own commits before pushing (`git show` and `git log --patch`), follow guidelines (for YARP avoid tabs and trailing white spaces)
- Read the manuals, search the internet, if you are still not sure ask someone before pushing

Good Practice 1: Example

●BAD

- `git commit -a -m "Fix bug"`
- `git pull`
- `git push origin master`

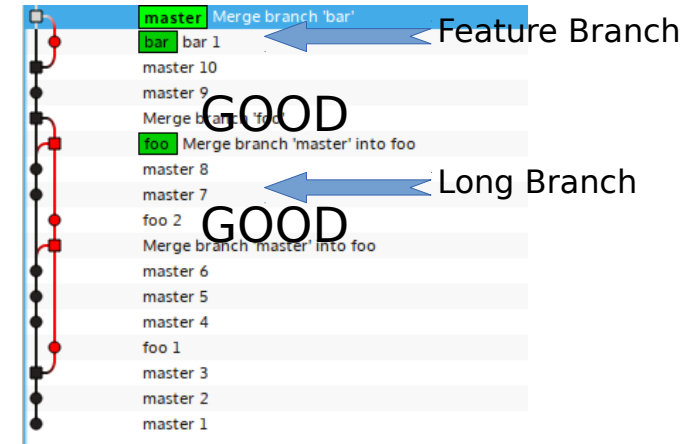
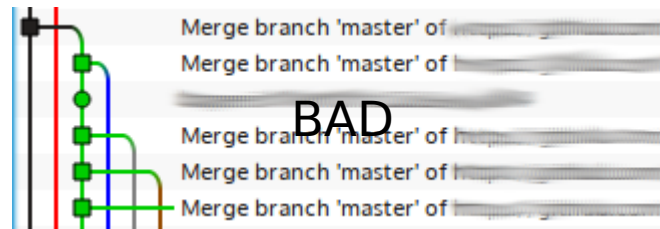
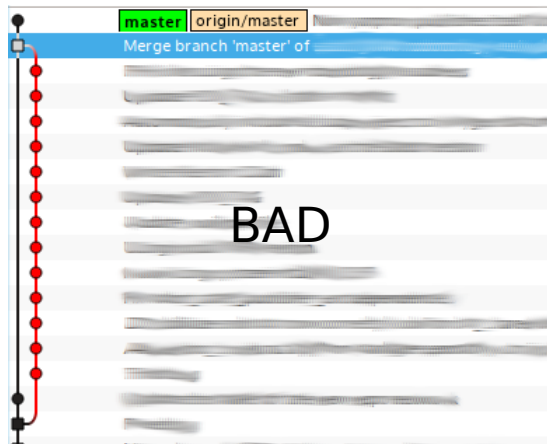
●GOOD

- `git add <file>` (even better stage the changes one by one using `git add -p <file>`)
- `git diff --cached` (always check what you are pushing)
- `git commit` (enter a proper commit log)
- `git pull --rebase origin master` (rebase your changes over the remote branch instead of creating an useless merge commit)
- `git log --oneline --graph --decorate origin/master..master` (always check which commits you are pushing)
- `git log -p origin/master..master` (always check the content of the commits that you are pushing)

Now that you are finally really sure about what you are pushing on the server

- `git push origin master`

Good Practice 1: Example



- “Merge branch 'master' of ...” usually means that you performed a “git pull” without rebasing your changes. One commit like this is bad enough, several developers doing this make a huge mess
- Use “git pull --rebase”.
- master history should always be on the left, this kind of commits makes the history dirty and hard to understand.
- If you cannot rebase, (for example for a shared branch that is developed for a long time) then use “git pull”, and either rebase the whole branch when it is ready to be merged, or merge back your changes into master before pushing to master, and not the opposite

Good practices 2:

Write good log messages

- Git log is very powerful, but with poor log messages, it's useless
- Do not commit with empty or useless log messages
 - Brief explanation of the commit
 - Leave an empty line
 - Detailed information about the commit
 - Use “`git commit --amend`” to edit the commit message of the last commit or use “`git rebase -i`” to edit the commit message of a commit that is not the last (do not edit published commits)

Good Practice 2: Example

- BAD

- `git commit -a -m "Fix bug"`

- GOOD

- `git commit`
(enter a proper commit log in the editor)

Summarize changes in around 50 characters or less

More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of the commit and the rest of the text as the body. The blank line separating the summary from the body is critical (unless you omit the body entirely); various tools like `'log'`, `'shortlog'` and `'rebase'` can get confused if you run the two together.

Explain the problem that this commit is solving. Focus on why you are making this change as opposed to how (the code explains that). Are there side effects or other unintuitive consequences of this change? Here's the place to explain them.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, preceded by a single space, with blank lines in between, but conventions vary here

If you use an issue tracker, put references to them at the bottom, like this:

Resolves: #123
See also: #456, #789

<http://chris.beams.io/posts/git-commit/>

Good practices 3:

Work in branches

- Create a branch for each bug you start fixing, or for each new feature you start to work on
- Switching branch with git is very easy and fast
- Merge into master only when you are ready, and you are sure that you will not break the build
- master should be always in a stable and releasable state.
- Feature branches are usually merged with “`git merge --no-ff <branch>`”. The `--no-ff` options forces git to create a merge commit even if the history could be linear.

Good practices 4:

Let someone else review your code

- Nobody writes bug-free code and knows all the best practices for writing code. Having someone else review your code helps reducing them.
- YARP and iCub use GitHub as main repository. GitHub has “pull requests”. If you are writing a non-trivial feature, push you commits in a branch on your private fork and open a “pull request” to ask to someone else to review your code
- Other projects use other web based tools (GitLab, Gerrit, Reviewboard, Phabricator) or a hierarchical access to repositories.

Good practices 5:

Never rewrite published history

- There are very few reasons why published history should be rewritten. Some projects enforce fast forward commits only. Unfortunately GitHub allows to do disable force pushes only if you pay for your account.
- Never push --force on a public repository.
- Never ever push --force on a public repository.
- If your push fails, and you are tempted to push --force, you are doing it wrong. pull or rebase, and try again.
- Recent git versions added a --force-with-lease flag that endures that there have been no changes since you last check. Do not use them, but if you really have some good reason to, use --force-with-lease.

QUESTIONS?

Getting Help

- `git help`
 - List of common commands
- `git <command> --help`
- `man git-command`
 - man page for <command>
- `git <command> -h`
 - brief help for <command>
- Search the internet (**stackoverflow** has all the answers, you just have to figure out the right question)
- Pro Git (Scott Chacon)
 - <http://www.git-scm.com/book>
- Some more useful links can be found here:
 - http://wiki.icub.org/wiki/Learning_more_about_git

Interactive Tutorials

- You should try them, even if you already know how to use git, they can teach you some new tricks.
- <http://try.github.com/>
- <http://pcottle.github.io/learnGitBranching/>

TIPS AND TRICKS WORKFLOW

Amending Commits

- Amend latest commit
 - `git commit --amend`
 - **WARNING**: Rewrites the history! Do not do it if you already pushed the commit
- Discards all your uncommitted changes and resets the current branch to the previous commit
 - `git reset --hard HEAD~`
 - **WARNING**: Deletes all your local changes!
- `git reset HEAD~`
 - Resets the current branch to the previous commit, but does not touch your working tree, your files are safe.

Stash

- You have some code that you don't want to commit yet, but you want to switch branch to work on something else
 - `git stash [save]`
 - “Stashes” your changes
 - `git stash pop`
 - Applies your stashed change to your current directory

Restoring Files

- **WARNING:** These commands will delete your local changes and there is no way to recover them.
- `git checkout [--] <file>`
 - Discards all uncommitted changes to a file (the “- -” is required in some cases, when git might be confused by the name of a branch)
- `git checkout <commit> [--] <file>`
 - Discards all uncommitted changes to a file and restore the file at <ref>
- `git reset --hard`
 - Discard all your uncommitted changes
- `git reset --hard <commit>`
 - Discards all your uncommitted changes and resets the current branch head to <commit>

Restore a Branch

- **WARNING:** These commands will delete your local changes and there is no way to recover them.
- Discard all the local changes and restore a branch to the status on the remote
 - `git fetch origin`
`git reset --hard origin/<branch>`

Referring to a Commit

- Full hash (85b19f74786eac3b43b8887e8529c9ac0537cf34)
- Short hash (85b19f74)
- Current checkout (HEAD)
- Branch (local) (master or refs/heads/master)
- Branch (remote) (origin/master or refs/remotes/origin/master)
- Previous branch (-)
- Tag (v2.3.21 or refs/tags/v2.3.21)
- One commit before HEAD (HEAD~ or HEAD~1)
- Some commits before master (master~5)
- The first parent of the current commit (HEAD^ or HEAD^1)
- The second parent of the current commit (for a merge commit) (HEAD^2)
- Two commits before head, then the second parent at the merge, then again three commits before, and the first parent at the merge (HEAD~2^2~3^1)
- Date, commit message...

MORE COMMANDS

Diff

- `git diff`
 - Shows diff between index and working dir
- `git diff --staged`
 - Shows diff between HEAD and index
- `git diff <object>`
 - Shows diff between object and working dir
- `git diff <object> <object>`
 - Shows diff between two objects

Show

- `git show`
 - "Shows" HEAD (displays the differences between HEAD and the previous commit)
- `git show <object>`
 - "Shows" a specific commit
- `git show --stat <object>`
 - "Shows" a commit, but just the stats, not the full patch

Blame

- `git blame <file>`
 - Shows the file with annotations on each line about the commit that modified that line, and its author

Tag

- `git tag`
 - Shows the tags in the repository
- `git tag -a -m "Message" tag-name`
 - Creates a new annotated tag (in your local repository)

Branch

- `git branch (-l)`
 - Shows local branches only
- `git branch -r`
 - Shows remote branches only
- `git branch -a`
 - Shows all branches (local and remote)

Create local branches

- `git branch new-branch <commit>`
 - Create a branch “new-branch” on HEAD or on specified commit. Does not do a checkout of that branch
- `git checkout -b new-branch <commit>`
 - Create a branch “new-branch” on HEAD or on specified commit and do a checkout of that branch

Merge

- `git merge <branch> ...`
 - Create a merge commit with multiple parents.
 - If it results in a conflict, user intervention is required
- `git mergetool`
 - Shows the configured tool to resolve conflicts
- Or you can solve them manually and then fix the index using `git add`
- After fixing the conflict, commit using “`git commit`” (no parameters)

Rebase

- **WARNING:** Rewrites the history!
- `git rebase <commit>`
 - Takes all your commits and applies them onto `<commit>`
- `git rebase -i <commit>`
 - Let you choose what to do with each commit (you can move, modify, delete, squash commits)

Bisect

- Very powerful tool to find the commit that caused a bug
- Let you interactively test a commit, and marking it as good or bad, “bisecting” until the first bad commit is found.
- See `git bisect --help`

Revert, Cherry-Pick

- `git revert <commit>`
 - Applies the reversed patch from <commit> to the repository and creates a new commit
- `git cherry-pick <commit>`
 - Applies a patch from <commit> to the repository and creates a new commit
- `git cherry-pick -x <commit>`
 - Same, but add a line to the commit with the hash of the original commit (useful when cherry-picking from another published branch)

Reflog

- Reflog is a mechanism to record when the tip of branches are updated.
 - `git reflog [ref]`
- Where have I been?
 - `git reflog`
 - `git reflog HEAD`
- Where has my branch <branch> been?
 - `git reflog <branch>`
- Can be very useful to find a previous commit after an error or an unwanted operation
 - `git reset --hard HEAD{2}`
 - reset current branch to where HEAD used to be two moves ago
 - **WARNING**: deletes all local changes
- **WARNING**: Reflog expires

Grep

- `git grep`
 - Searches for a string in tracked files
 - Faster than system `grep` (that usually requires a `find`)
 - Searches only in tracked files
 - Does not search in the `.git` directory
 - Does not search in new files

TIPS AND TRICKS SETUP

Bash Prompt

- Bash prompt can be tweaked to show you in which branch you are by using
`$(__git_ps1)`
- You can add it to your PS1 variable (see http://wiki.icub.org/wiki/Learning_more_about_git for an example)
- This is very useful when working with more than one branch.

Generate an SSH Key

- Most of the remote git servers allow you to use several protocols to connect.
- Using SSH protocol with an SSH key is more secure than using username and password, and often faster.
- HTTPS has some issues with large repositories.
- To create an SSH key you can just run:
`ssh-keygen -t rsa`

Adding an SSH Key to GitHub

- Allows you to use SSH to push commits instead of HTTPS.
- Open `https://github.com/settings/ssh`
- Click “Add SSH key”
- Insert a “Title” and copy the content of the `~/.ssh/id_rsa.pub` file in the “Key” field.
- Click Add key.

End of Lines

- This avoids issues with Unix-style and Windows-style end of lines

```
git config --global core.autocrlf input  
(Linux/Mac)
```

```
git config --global core.autocrlf true (Windows)
```

Editor, Diff Tool, and Pager

- `git config --global core.editor "vim"`
- `git config --global merge.tool "vimdiff"`
- `git config --global core.pager "less -FRSX"`

Https Authentication Helper

- Caches https username and passwords for <timeout> seconds.

```
git config --global credential.helper \  
"cache --timeout=7200"
```


Auto Setup Rebase

- Automatically configures your branches to perform a rebase instead of a merge when you run “git pull”
- `git config --global branch.autosetuprebase “always”`
- **WARNING:** this is a possibly dangerous operation; do not use it unless you understand the implications. We'll discuss about rebase later.