Version control using `git` & GitHub

Pablo Orviz <orviz@ifca.unican.es>
Instituto de Física de Cantabria

Máster en Data Science
2025/26

Outline

1. Version Control Systems (VCS)
2. Git and GitHub
3. Hands-on (basics)

Outline

1. **Version Control Systems (VCS)**
2. Git and GitHub
3. Hands-on (basics)

Version Control

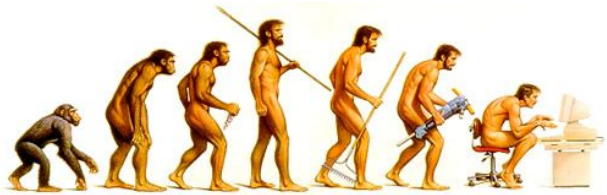
> *“Systems that record changes to a/set of file/s over time so that you can recall specific versions later”* (Chacon&Straub, *Pro Git*)

Version Control

> *"Systems that record changes to a/set of file/s over time so that you can recall specific versions later"* (Chacon&Straub, *Pro Git*)

Why use version control?

1. You noticed that your code is doing odd things now and didn't use to → ***Compare changes over time***
2. You deleted some code and want to get it back →***Revert files back to a previous state***
3. You want to show your supervisor what you did last week and/or (collaborative environments) see what your collaborators wrote last week → ***Move to a specific version (timestamp)***
4. You want to experiment with your code which may break the project and/or try different strategies to solve a problem→ ***Branch off from a specific version (content)***
5. You want to check when & who & last modified something that is causing a problem/issue →**Audit project history (person)**



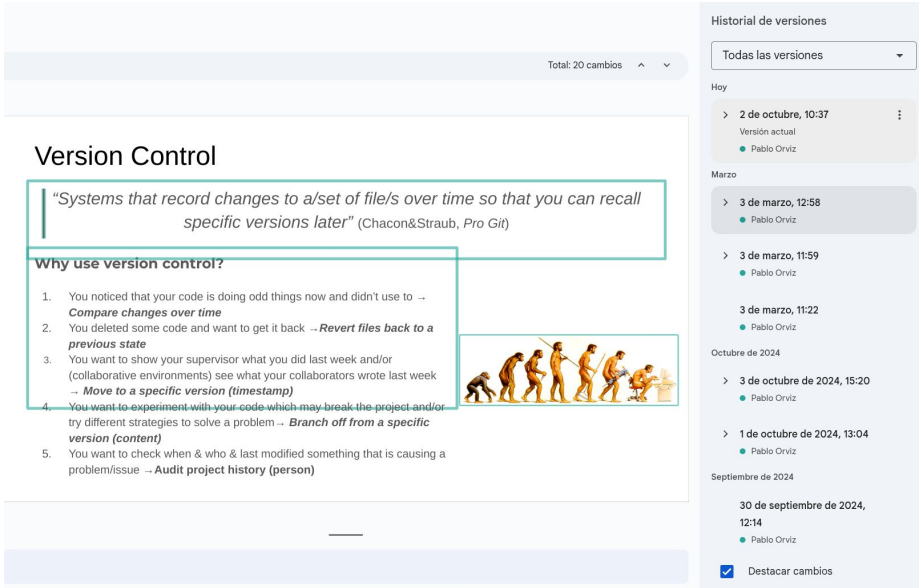
Version Control

> *“Systems that record changes to a/set of file/s over time so that you can recall specific versions later”* (Chacon&Straub, *Pro Git*)

Why use version control?

1. **Essential tool for code management** in collaborative environments of software development
2. **Best choice for Document version control**: outperforms built-in VCS of popular document (sharing) solutions, e.g. Microsoft Office, Google Docs, ..

> *“The revisions for your file may occasionally be merged to save storage space”* (Google Docs disclaimer)



Version Control Systems (VCS)

Pre-VCS solutions

Filesystem-based

- Duplicate files/directories
  - Hopefully time-stamped
  - Usually with prefix/suffix such as ‘v1’, ..

Version Control Systems (VCS)

Pre-VCS solutions

Filesystem-based

- Duplicate files/directories
  - Hopefully time-stamped
  - Usually with prefix/suffix such as ‘v1’, ..

Error-prone approach!

Version Control Systems (VCS)

Pre-VCS solutions

Database-centric

- *First approach to a real VCS*
- Each new version is stored in a database as a patch set (only *diffs*)
- Capable of *re-creating any file* by adding up the appropriate patches

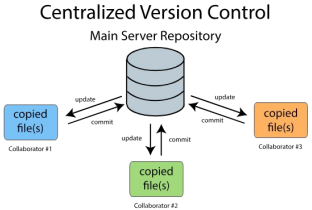
6 cloud_info_provider/providers/static.py

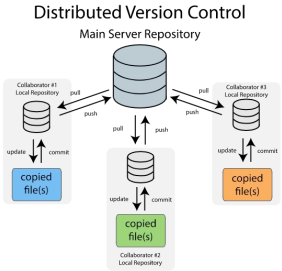
```
@@ -86,9 +86,11 @@ def get_site_info(self, **kwargs):
86  86          data = self.yaml.get('site', {'name': None})
87  87          site_info = self._get_fields_and_prefix(
88  88              ('name', 'id',
89      -            'country', 'country_code', 'roc', 'subgrid', 'giis_url'),
        89  +        'country', 'country_code', 'roc', 'subgrid', 'giis_url',
        90  +        'is_public'),
90  91              'site_',
91      -            data)
        92  +        data,
        93  +        defaults={'is_public': False})
92  94
93  95          # Resolve site name from BDII configuration
94  96          if site_info['site_name'] is None:
```

Version Control Systems (VCS)

Fortunately.. The advent of ad-hoc tools for version control simplified this task

- **Evolution/timeline** (2 types): *Centralized* (CVCS) ⇉ *Distributed* (DVCS)



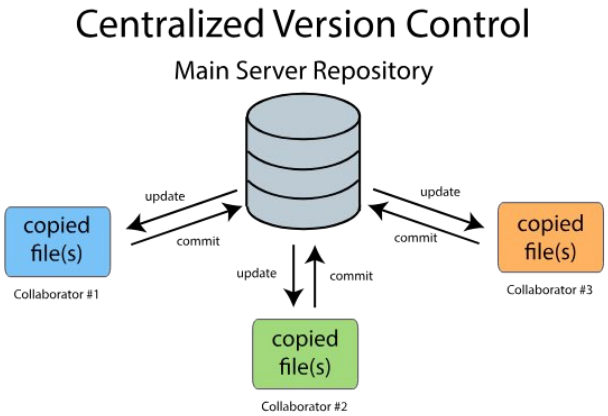


- Basic **terminology**:
  - *Repository* == database of changes
  - *Working copy* == personal copy of all the files in the repository
  - *Commit* (checkin) == creates a new version from the selected changes
  - *Pull* (checkout, update) == makes last version available from a remote/local repository
  - *Push* == stores current version in a remote/local repository
  - *Conflict* == 2 (or more) different versions of the same file/s (needs action: *resolve*)
  - *Resolve* == manual editing of conflicts (cannot be automatically resolved) in order to have 1 version

Version Control Systems (VCS)

Centralized Version Control Systems (*aka CVCS*)

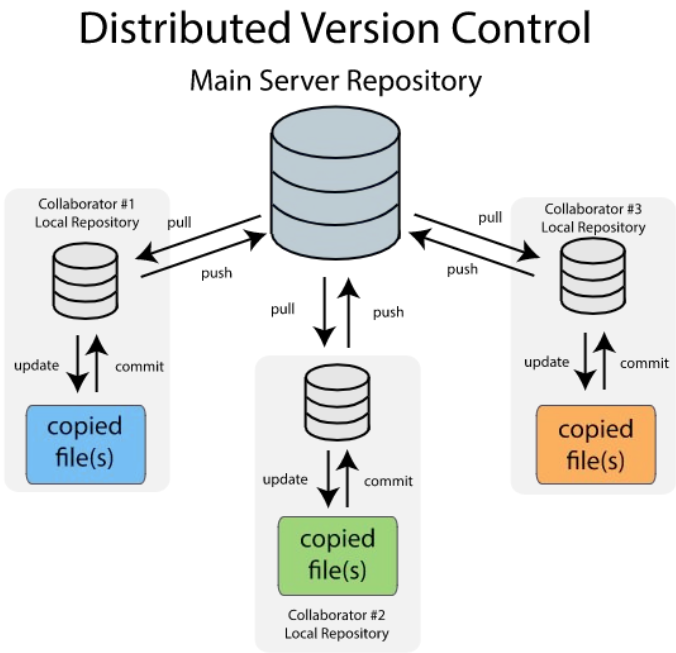
- Problem statement: Collaboration was needed, geographically distributed teams
- Centralized solution (server-client model):
  - **1 server** with all the versioned files
  - **N clients** that check out files from the server
    - ***Lock*** mechanism == *one developer at a time*
- Pros:
  - **Allows collaboration**
  - Everyone knows what the others do (to a certain degree)
  - Huge step ahead when compared with pre-VCS DBs
- Cons:
  - **Single point of failure**
    - Network outage (temporary loss), disk failure/corrupted (~permanent loss)
  - **Slower**: all operations are remote
- *Examples: CVS, Subversion*



Version Control Systems (VCS)

Distributed Version Control Systems (*aka DVCS*)

- Problem statement: Improve availability & reliability
- Distributed solution:
  - Removes the need of a central repository
  - Each client **clones the entire repository**
    - Full backup
    - Fast operations (all work done locally)
- Pros:
  - **High Availability**
    - Restore content from any client repository
  - **Multiple collaborations** through multiple remotes
    - No need of locking: Simultaneous operations
- Cons:
  - None, they are perfect! (almost)
- *Examples: **`git`**, mercurial, **fossil***



Version Control Systems (VCS)

Distributed Version Control Systems (*aka DVCS*)

Git vs Fossil

- *Fossil's view: https://fossil-scm.org/home/doc/trunk/www/fossil-v-git.wiki*
  - More similarities than actual differences
  - File versioning (Git) vs wiki/tickets/forums/chats/.. (Fossil)
  - key-value store (Git) vs relational SQL database (Fossil)
  - History can be modified (Git) vs History as it was (Fossil)
  - Commit-first (Git) vs Test-first (Fossil)
  - Manual Push (Git) vs Autosync (Fossil)

git

VCS in a nutshell

If all you remember is..

- *A VCS offers a great added value when it comes to **manage changes over time on** whatever type of **plain-text format files***

VCS in a nutshell

If all you remember is..

- *A VCS offers a great added value when it comes to **manage changes over time on** whatever type of **plain-text format files***
  - Compare changes over time
  - Revert files back to a previous state
  - Move to a specific version (timestamp)
  - Branch off from a specific version (content)
  - Audit project history (person)

VCS in a nutshell

If all you remember is..

- *A VCS offers a great added value when it comes to **manage changes over time on** whatever type of **plain-text format files***
  - Compare changes over time
  - Revert files back to a previous state
  - Move to a specific version (timestamp)
  - Branch off from a specific version (content)
  - Audit project history (person)
- ***Distributed VCS are the best choice** for VC*

Why Are Software Projects Moving From Centralized to Decentralized Version Control Systems?

Brian de Alwis
Dept of Computer Science
University of Saskatchewan
Saskatoon, SK, Canada
brian.de.alwis@usask.ca

Jonathan Sillito
Dept of Computer Science
University of Calgary
Calgary, AB, Canada
sillito@ucalgary.ca

Abstract

Version control systems are essential for co-ordinating work on a software project. A number of open- and closed-source projects are proposing to move, or have already moved, their source code repositories from a centralized version control system (CVCS) to a decentralized version control system (DVCS). In this paper we summarize the differences between a CVCS and a DVCS, and describe some of the rationales and perceived benefits offered by projects to justify the transition.

BZR,[3] and BITKEEPER,[4] have become sufficiently mature that many open- and closed-source projects are proposing to move, or have already moved, their source code repositories to a DVCS.

As part of a larger research project to explore practices and tool support around version management, we have begun a qualitative study to answer two research questions. First, what do these projects see as the benefits of a DVCS? Transitioning a source code repository to a new VCS requires significant effort,[5] and so we suppose that there must be compelling reasons for switching. Second, what changes have these projects made to their develop-

De Alwis, B., & Sillito, J. (2009, May). *Why are software projects moving from centralized to decentralized version control systems?*. In *2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering* (pp. 36-39). IEEE.

VCS in a nutshell

If all you remember is..

- *A VCS offers a great added value when it comes to **manage changes over time on** whatever type of **plain-text format files***
  - Compare changes over time
  - Revert files back to a previous state
  - Move to a specific version (timestamp)
  - Branch off from a specific version (content)
  - Audit project history (person)
- ***Distributed VCS are the best choice** for VCS*
  - High availability: they do not depended on a central server
  - Improve collaboration: handling multiple remotes
  - Speed: common operations are done faster as metadata is stored locally

Why Are Software Projects Moving From Centralized to Decentralized Version Control Systems?

Brian de Alwis
Dept of Computer Science
University of Saskatchewan
Saskatoon, SK, Canada
brian.de.alwis@usask.ca

Jonathan Sillito
Dept of Computer Science
University of Calgary
Calgary, AB, Canada
sillito@ucalgary.ca

Abstract

Version control systems are essential for co-ordinating work on a software project. A number of open- and closed-source projects are proposing to move, or have already moved, their source code repositories from a centralized version control system (CVCS) to a decentralized version control system (DVCS). In this paper we summarize the differences between a CVCS and a DVCS, and describe some of the rationales and perceived benefits offered by projects to justify the transition.

BZR,[3] and BITKEEPER,[4] have become sufficiently mature that many open- and closed-source projects are proposing to move, or have already moved, their source code repositories to a DVCS.

As part of a larger research project to explore practices and tool support around version management, we have begun a qualitative study to answer two research questions. First, what do these projects see as the benefits of a DVCS? Transitioning a source code repository to a new VCS requires significant effort,[5] and so we suppose that there must be compelling reasons for switching. Second, what changes have these projects made to their develop-

De Alwis, B., & Sillito, J. (2009, May). *Why are software projects moving from centralized to decentralized version control systems?*. In *2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering* (pp. 36-39). IEEE.

VCS in a nutshell

If all you remember is..

- *A VCS offers a great added value when it comes to **manage changes over time on** whatever type of **plain-text format files***
  - Compare changes over time
  - Revert files back to a previous state
  - Move to a specific version (timestamp)
  - Branch off from a specific version (content)
  - Audit project history (person)
- ***Distributed VCS are the best choice** for VCS*
  - High availability: they do not depended on a central server
  - Improve collaboration: handling multiple remotes
  - Speed: common operations are done faster as metadata is stored locally
- `git` ***is most widely-used DVCS tool***
  - ..although alternatives exist such as Fossil

Why Are Software Projects Moving From Centralized to Decentralized Version Control Systems?

Brian de Alwis
Dept of Computer Science
University of Saskatchewan
Saskatoon, SK, Canada
brian.de.alwis@usask.ca

Jonathan Sillito
Dept of Computer Science
University of Calgary
Calgary, AB, Canada
sillito@ucalgary.ca

Abstract

Version control systems are essential for co-ordinating work on a software project. A number of open- and closed-source projects are proposing to move, or have already moved, their source code repositories from a centralized version control system (CVCS) to a decentralized version control system (DVCS). In this paper we summarize the differences between a CVCS and a DVCS, and describe some of the rationales and perceived benefits offered by projects to justify the transition.

BZR,[3] and BITKEEPER,[4] have become sufficiently mature that many open- and closed-source projects are proposing to move, or have already moved, their source code repositories to a DVCS.

As part of a larger research project to explore practices and tool support around version management, we have begun a qualitative study to answer two research questions. First, what do these projects see as the benefits of a DVCS? Transitioning a source code repository to a new VCS requires significant effort,[5] and so we suppose that there must be compelling reasons for switching. Second, what changes have these projects made to their develop-

De Alwis, B., & Sillito, J. (2009, May). *Why are software projects moving from centralized to decentralized version control systems?*. In *2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering* (pp. 36-39). IEEE.

Outline

1. Version Control Systems (VCS)
2. **`git` and GitHub**
3. Hands-on (basics)

Social coding

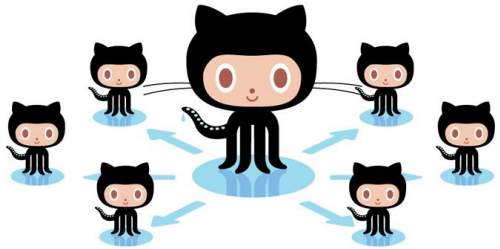
Re-shaping the Culture of Open Source: "everyone building things together for everyone else to use" (@wired)

Before social coding platforms..

- Open source maintained by ***professional developers***
- The bigger & more complex the sw project, the ***harder to contribute***
- Rigid framework (roles): ***producers vs consumers***

With the emergence of social coding platforms..

- Put the focus on ***individual developers*** (de-centralization)
- Very ***easy to contribute***, broadened peer production to individual users (democratization)
- Flexible framework (roles): ***producers are consumers and viceversa***



Social coding

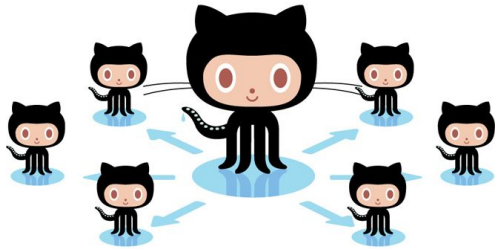
Re-shaping the Culture of Open Source: "everyone building things together for everyone else to use" (@wired)

Before social coding platforms..

- Open source maintained by ***professional developers***
- The bigger & more complex the sw project, the ***harder to contribute***
- Rigid framework (roles): ***producers vs consumers***

With the emergence of social coding platforms..

- Put the focus on ***individual developers*** (de-centralization)
- Very ***easy to contribute***, broadened peer production to individual users (democratization)
- Flexible framework (roles): ***producers are consumers and vice versa***



Tools for social coding

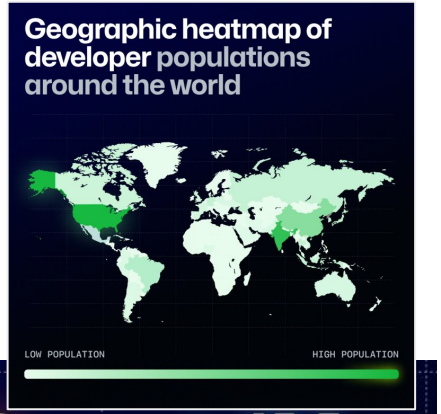
GitHub

GitLab

Bitbucket

GitHub

Git-repository hosting service: @github.com

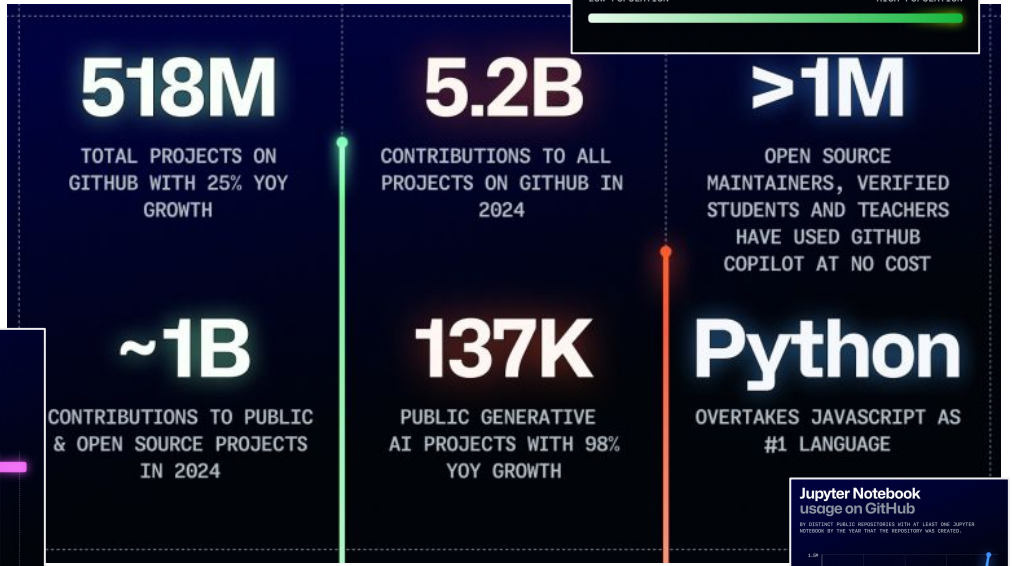


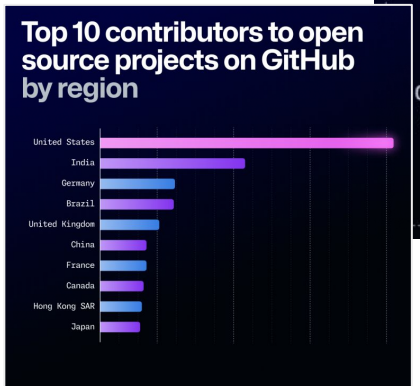
Social coding: account-based

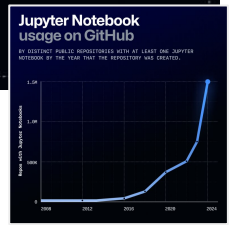
- *Individual:* https://github.com/**orviz**
- *Organizations*: https://github.com/**masterdatascience-UIMP-UC**

Extra functionality (on top of `git`):

- Collaborative features of social coding
  - Forks
  - Pull requests
- Private and public repositories
- Actions/workflows
- Dependabot
- Integrations with other services: CI, monitoring, ..
- Tools (wiki, issue tracking)
- Project webpage: GitHub Pages
- Reporting: graphs/stats ...



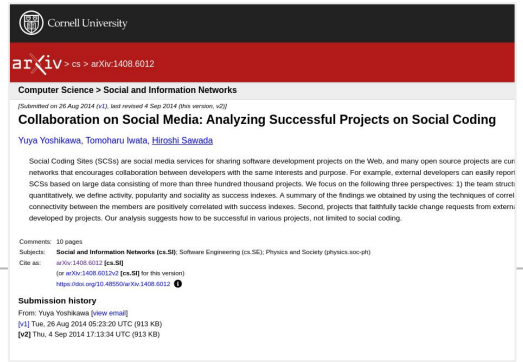




GitHub

Science in GitHub.com: democratic databases (Oct 2016)

1. Ebola outbreak in West Africa, *July 2014*
2. PhD student wanted to model the outbreak spread
3. Every day
   - Downloaded PDF updates from the ministries of health of the affected countries
   - Converted the numbers into computer readable tables
   - Uploaded the files to a GitHub.com repository, thought may be useful for someone
4. Other researchers started to contribute in the project:
   - On some days, files were uploaded before hers
   - Created programming scripts for error-checks on the data


Cornell University

arXiv > cs > arXiv:1408.6012

Computer Science > Social and Information Networks

[Submitted on 26 Aug 2014 (v1), last revised 4 Sep 2014 (this version, v2)]

Collaboration on Social Media: Analyzing Successful Projects on Social Coding

Yuya Yoshikawa, Tomoharu Iwata, Hiroshi Sawada

Social Coding Sites (SCSs) are social media services for sharing software development projects on the Web, and many open source projects are cur networks that encourages collaboration between developers with the same interests and purpose. For example, external developers can easily report SCSs based on large data consisting of more than three hundred thousand projects. We focus on the following three perspectives: 1) the team structi quantitatively, we define activity, popularity and sociality as success indexes. A summary of the findings we obtained by using the techniques of correl connectivity between the members are positively correlated with success indexes. Second, projects that faithfully tackle change requests from extern developed by projects. Our analysis suggests how to be successful in various projects, not limited to social coding.

Comments: 10 pages
Subjects: **Social and Information Networks (cs.SI)**; Software Engineering (cs.SE); Physics and Society (physics.soc-ph)
Cite as: arXiv:1408.6012 [cs.SI]
(or arXiv:1408.6012v2 [cs.SI] for this version)
https://doi.org/10.48550/arXiv.1408.6012

Submission history
From: Yuya Yoshikawa [view email]
[v1] Tue, 26 Aug 2014 05:23:20 UTC (913 KB)
[v2] Thu, 4 Sep 2014 17:13:34 UTC (913 KB)


BRITISH ECOLOGICAL SOCIETY

Methods in Ecology and Evolution

REVIEW | Open Access

Not just for programmers: How GitHub can accelerate collaborative and reproducible research in ecology and evolution

Pedro Henrique Pereira Braga, Katherine Hébert, Emma J. Hudgins, Eric R. Scott, Brandon P. M. Edwards, Luna L. Sánchez Reyes, Matthew J. Grainger ... See all authors

First published: 21 April 2023 | https://doi.org/10.1111/2041-210X.14108 | Citations: 22


nature > humanities and social sciences communications > articles > article

Article | Open access | Published: 01 March 2025

Citizen participation and technology: lessons from the fields of deliberative democracy and science and technology studies

Julian "Iñaki" Goñi

Humanities and Social Sciences Communications **12**, Article number: 287 (2025) | Cite this article


PLOS Biology

► PLoS Biol. 2025 Feb 14;23(2):e3003029. doi: 10.1371/journal.pbio.3003029

GitHub enables collaborative and reproducible laboratory research

Katharine Y Chen [1], Maria Toro-Moreno [1,*], Arvind Rasi Subramaniam [1,*]

► Author information ► Article notes ► Copyright and License information

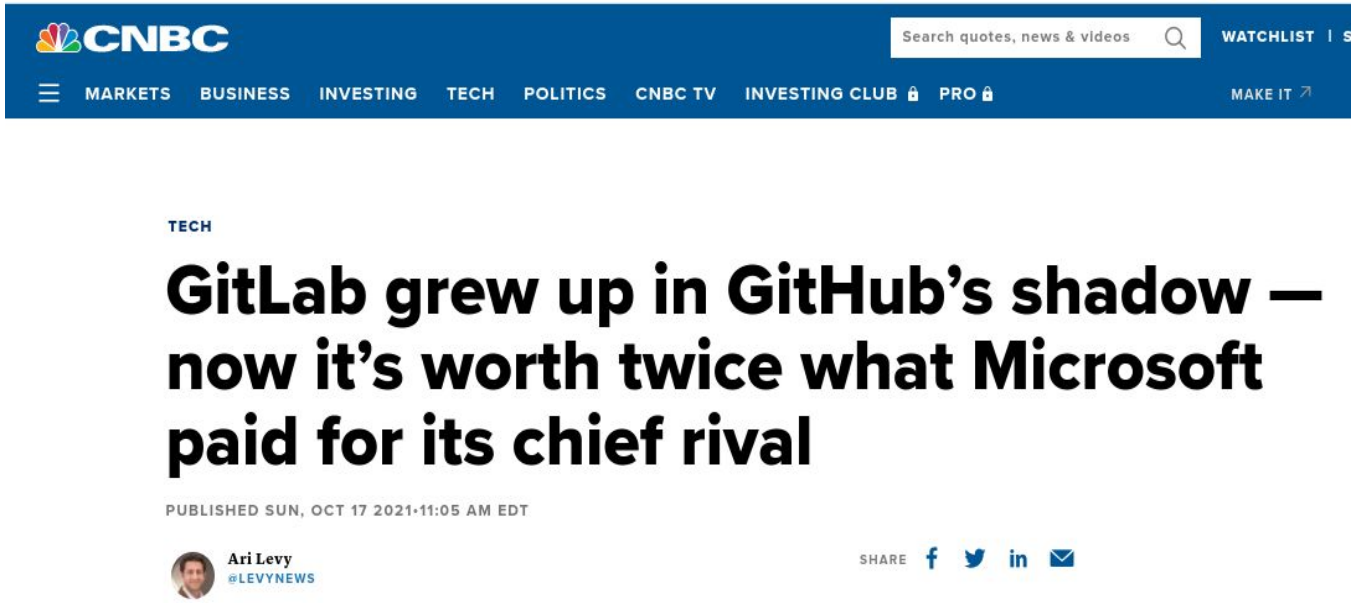
PMCID: PMC11828340 PMID: 39951468

NEWS · 15 JUNE 2018

Microsoft's purchase of GitHub leaves some scientists uneasy

They fear the online platform will become less open, but other researchers say the buyout could make GitHub more useful.

GitLab



01 Software faster

02 AI-powered

03 Privacy-first AI

04 Security

05 Compliance

06 Deployment

07 Insights

08 Planning

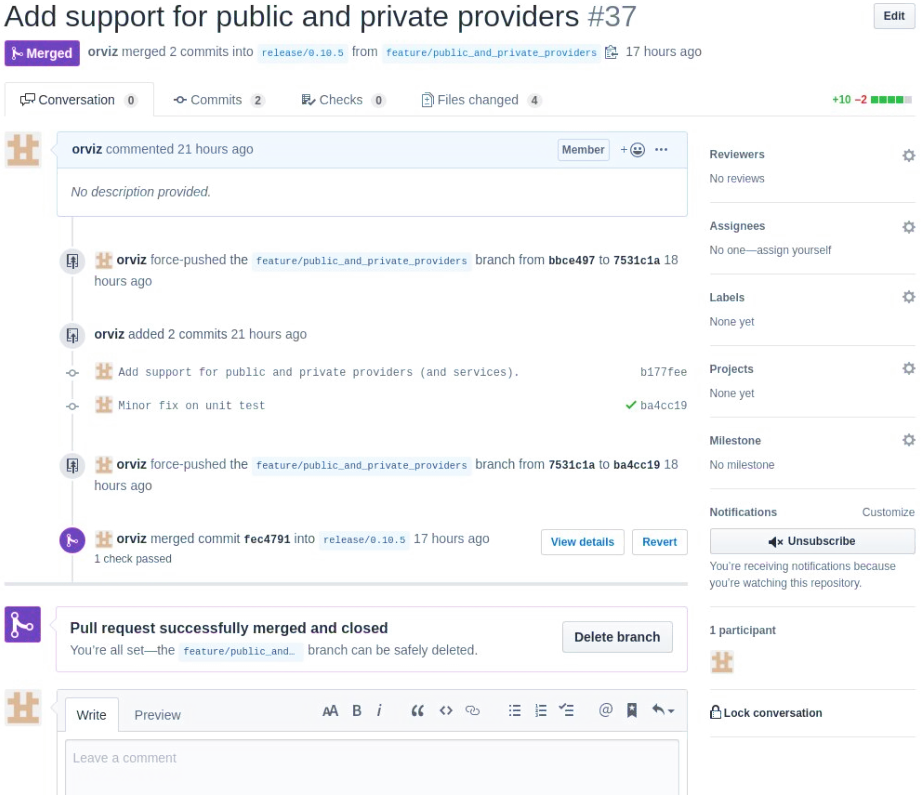
09 No vendor lock-in

10 Open core

Collaborative features

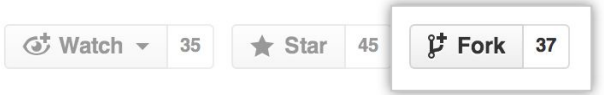
GitHub

Pull Request (PR): primary means of collaboration in GitHub



2 main ways to PR..

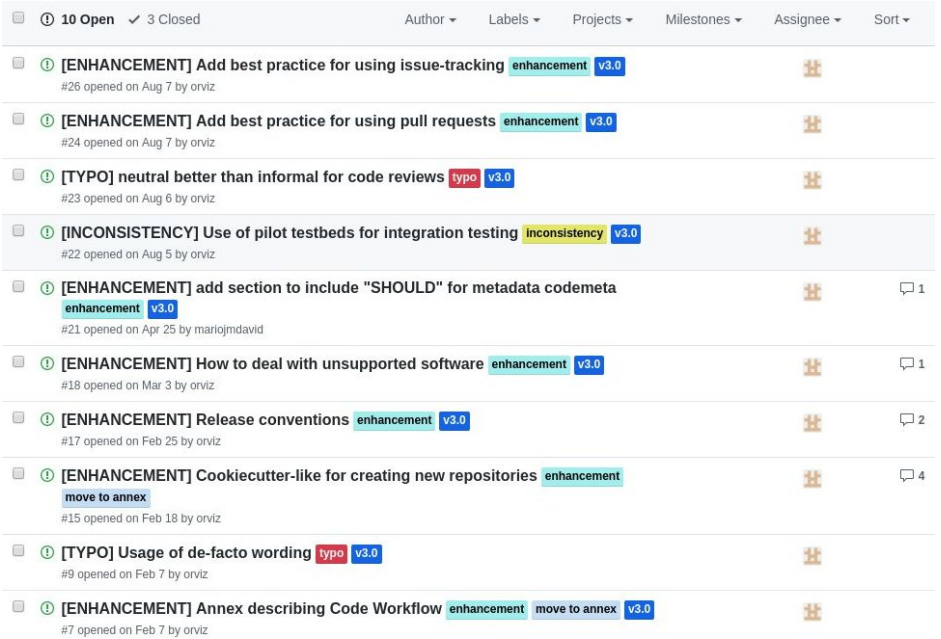
- **Cloning**: requires you to be main collaborator of the repo
  - *Collaboration*: PR done through *branches*
- **Forking**: own repo copy, suitable for contributions to external repos
  - Full permissions
  - *Collaboration*: fork changes submitted to upstream repo through PR
  - Usually *involves cloning*



Issues

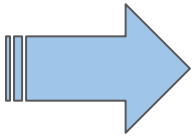
Useful feature to improve your development workflow..

- Track new ideas, enhancements, bugs..
- Receive feedback from the outside world
- Prioritize & Distribute (assignments, labels) your work
- Can be closed automatically through PRs

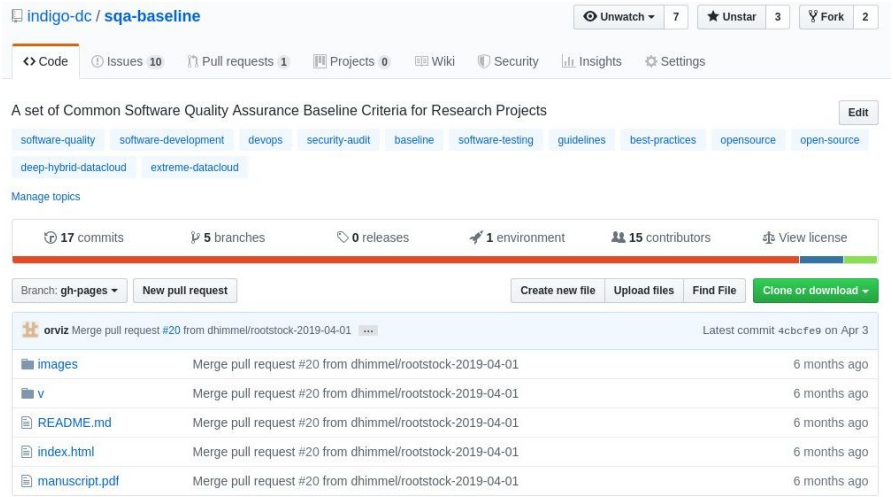


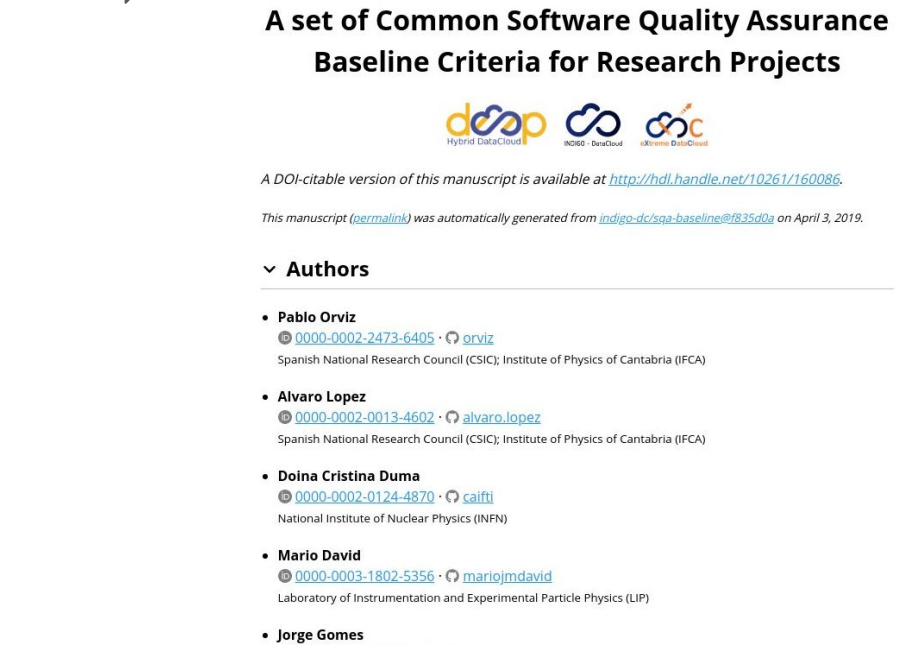
GitHub Pages: project website

https://github.com/indigo-dc/sqa-baseline/



https://indigo-dc.github.io/sqa-baseline/



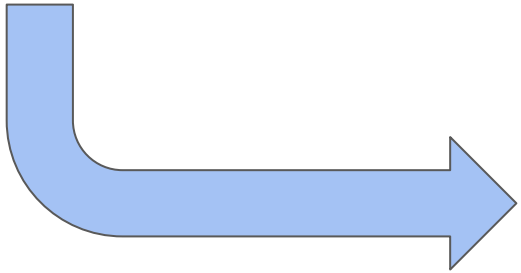


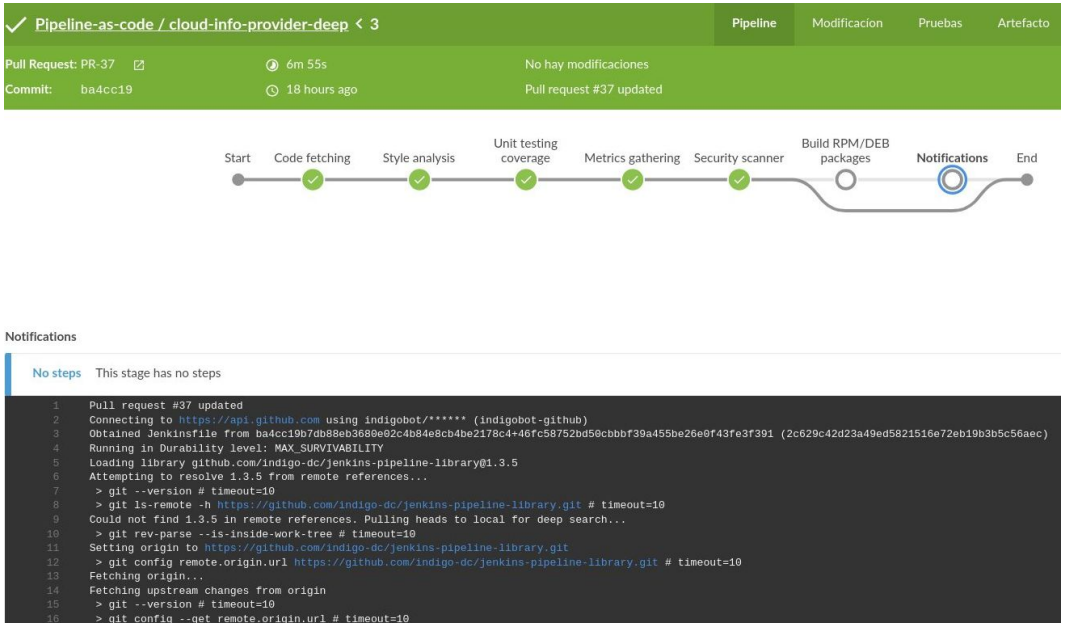
Integrations

Continuous integration, Code quality, Monitoring, Project Management..

https://github.com/marketplace

- Very useful for adding checks/tests to our source code
- Tests are executed for each change → PRs





git

- Uses **snapshots**, not diffs
  - Every change (commit, save state, ..) triggers a snapshot of all the (meta)data and adds a reference to it
    - Different from the *delta-based*: just stores the files changed and the changes themselves
- Most **operations are local**
  - Operations are *instant*
    - Checks local database, no network needed
    - Unlike CVCS, which adds network latency overhead
- The **Three States**
  1. *Modified*: the file has changed
  2. *Staged*: mark a modified file to go into your next version
  3. *Committed*: establishes/creates the new version in your local database

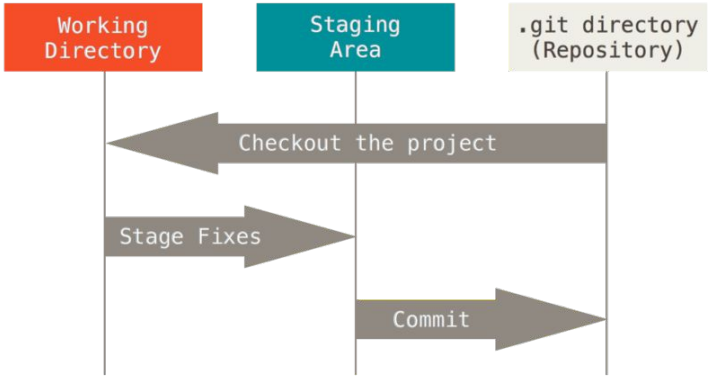
`git` basics

The Three Sections of a Git project

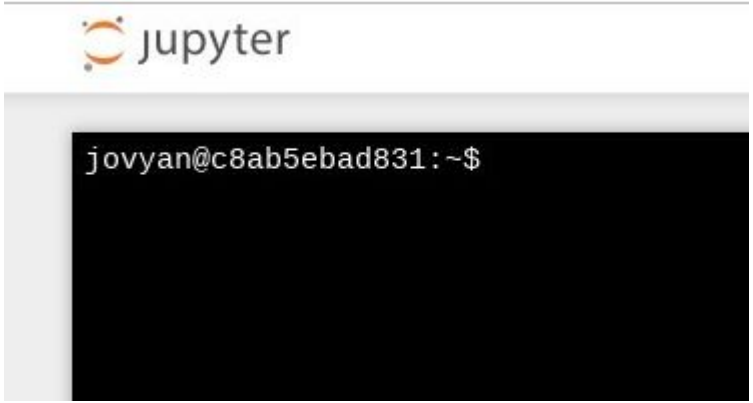
- *Git directory (.git)*: metadata and DB
- *Working directory*: local copy (from remote, checkout) in a given version
- *Staging area*: changes that will go into the next commit/version

Workflow:

1. **M**odify files in your *working dir*ectory
2. **S**tage the changes in files meant to go into the next commit (**`add`** operation)
   - Changes are moved to the *staging area*.
3. **C**ommit the staged changes (**`commit`** operation)
   - Creates a snapshot of the files in the staging area to place them in the `.git` *directory*.



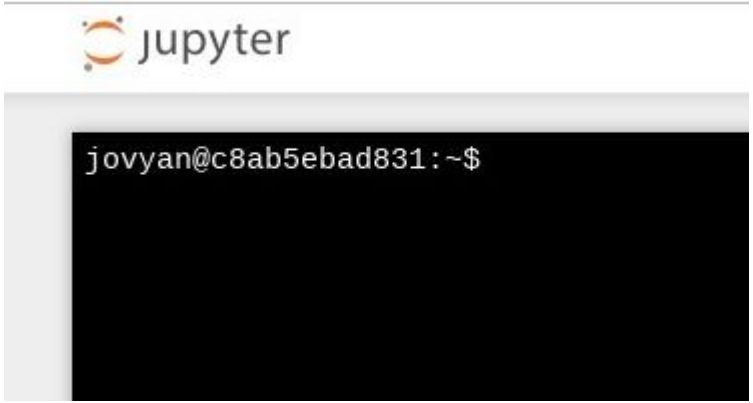
Getting started with `git`



Command-line (CLI) vs GUI

- ***We will be using `git` on the command line***
  - GUIs implement only a partial subset, e.g. https://desktop.github.com/
    - Alternative solutions: https://git-scm.com/downloads/guis
  - Hard (*and best*) way: if you master the command-line, GUI won't be any secret.
    - The opposite is not necessarily true.
- `git` *CLI is available through https://datasciencehub.ifca.es*
  - Web-based, no need to install the Git client locally
  - But **preferably install `git` locally** →how to install git
- A GitHub account is required
  - If not already, create one at https://github.com

Getting started with `git`



Access to the terminal

1. Go to *https://datasciencehub.ifca.es*
2. Log in with your GitHub credentials
3. Press `Start server` if you haven't done it before
4. Open a terminal: on Files tab > New > Terminal
    a. Reuse an already open terminal: Running tab > Terminals

Getting started with `git`

Installing git on your local computer

Installing on Linux

If you want to install the basic Git tools on Linux via a binary installer, you can generally do so through the package management tool that comes with your distribution. If you're on Fedora (or any closely-related RPM-based distribution, such as RHEL or CentOS), you can use `dnf`:

```
$ sudo dnf install git-all
```

If you're on a Debian-based distribution, such as Ubuntu, try `apt`:

```
$ sudo apt install git-all
```

For more options, there are instructions for installing on several different Unix distributions on the Git website, at https://git-scm.com/download/linux.

Installing on macOS

There are several ways to install Git on macOS. The easiest is probably to install the Xcode Command Line Tools. On Mavericks (10.9) or above you can do this simply by trying to run `git` from the Terminal the very first time.

```
$ git --version
```

If you don't have it installed already, it will prompt you to install it.

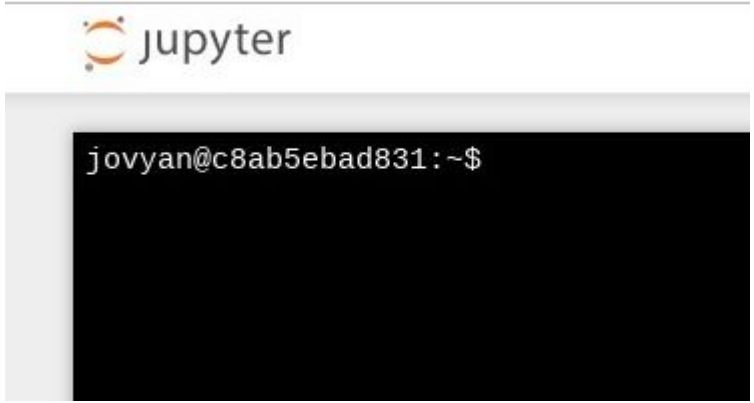
If you want a more up to date version, you can also install it via a binary installer. A macOS Git installer is maintained and available for download at the Git website, at https://git-scm.com/download/mac.

Installing on Windows

There are also a few ways to install Git on Windows. The most official build is available for download on the Git website. Just go to https://git-scm.com/download/win and the download will start automatically. Note that this is a project called Git for Windows, which is separate from Git itself; for more information on it, go to https://gitforwindows.org.

To get an automated installation you can use the Git Chocolatey package. Note that the Chocolatey package is community maintained.

Getting started with `git`



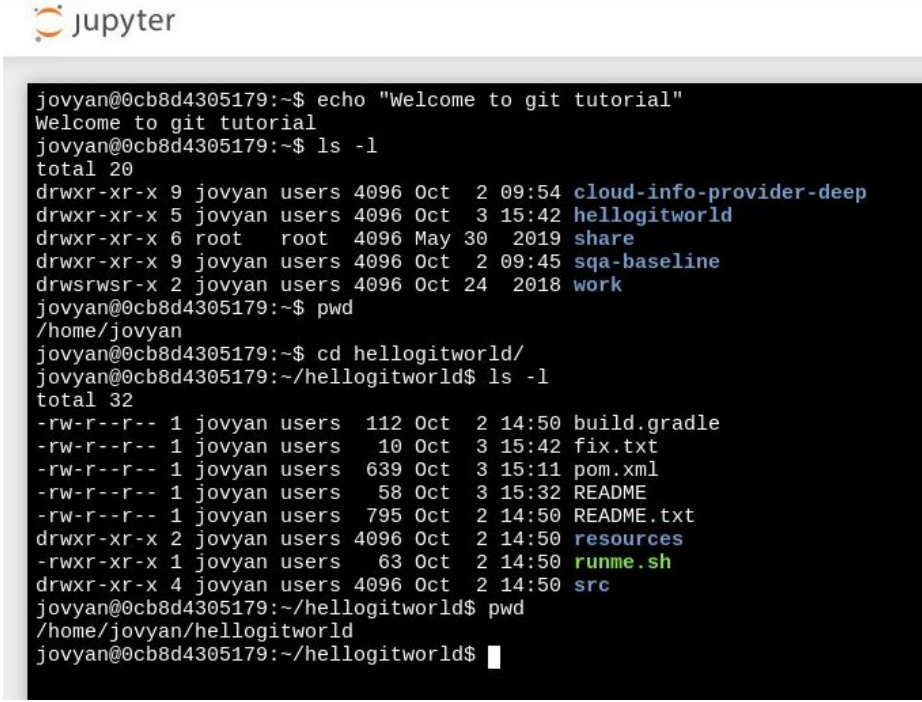
Shell basics

- Shell is the CLI for running programs on UNIX systems
  - Runs the programs/commands and returns the output
- Shell types: **bash**, sh, csh, zsh, tcsh, ..
- Terminal: interacts with the shell
  - Just like a web browser with websites
  - Shell prompt ($): ready to accept input commands

Getting started with `git`

Shell basics: commands for git tutorial

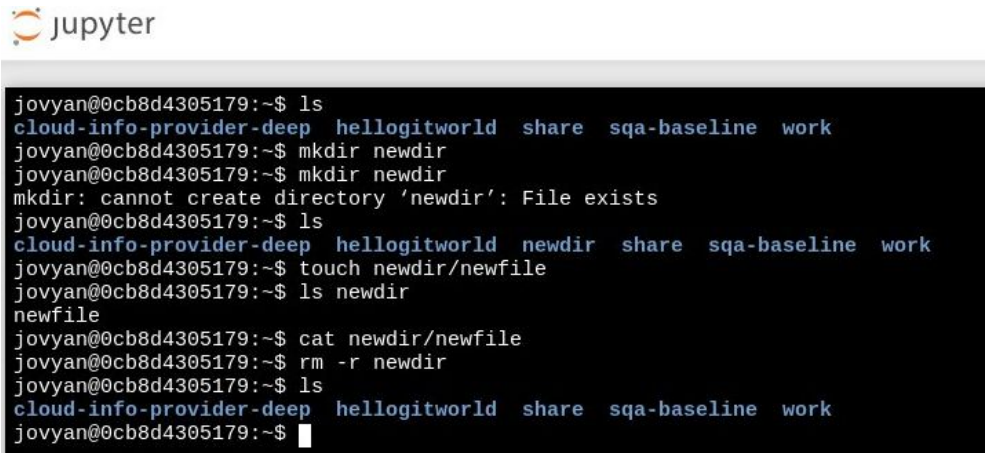
- `echo`: returns whatever you type
  - Much like print in programming languages
- Navigating:
  - `ls`: list the contents of a directory
    - `ls -l`: provides additional information such as file permissions
  - `cd`: **c**hanges **d**irectory
  - `pwd`: tell us what is our current directory



Getting started with `git`

Shell basics: commands for git tutorial

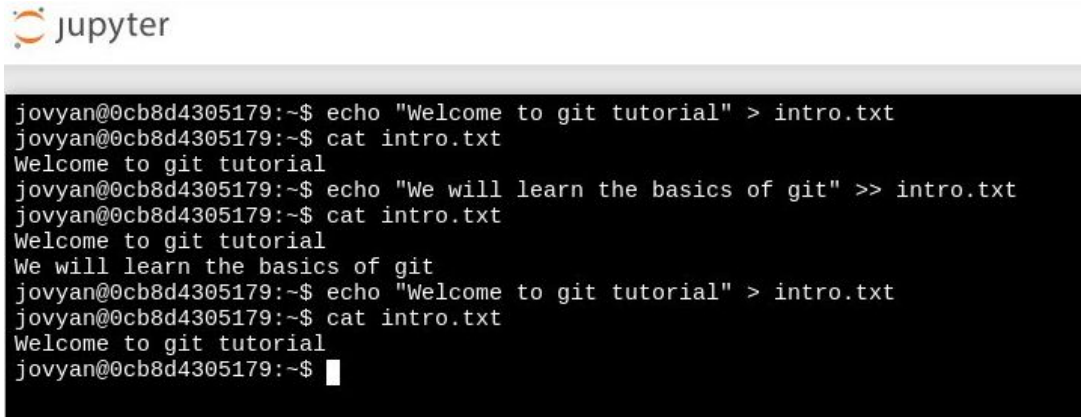
- Organizing directories/files:
  - `mkdir`: creates a directory
  - `rmdir`: removes a directory
  - `mv`: moves files/dirs from one location to another (good to rename files/dirs)
  - `touch`: creates an empty file
  - `rm`: removes a file
    - In order to remove a dir, use `rm -r`
- Viewing files:
  - `cat`: reads a file and outputs its contents
- Help:
  - `man`: help for commands



Getting started with `git`

Shell basics: commands for git tutorial

- Redirecting:
  - Overwrite: `>`
  - Append: `>>`
- With `echo`:



Getting started with `git`

First-time `git` setup

- 3 `git` configuration files on your system (from lowest to highest priority):
  1. **`/etc/gitconfig`** (system-wide)
  2. **`$HOME/.gitconfig`** or **`$HOME/.config/git/config`** (user space)
  3. **`.git/config`** in your `git` directory (single repository)
- ***`git config`*** command line will modify the values in the files above for you
  - With ***--system*** option, reads/writes from **`/etc/gitconfig`**
  - With ***--global*** option, reads/writes from user space, aka **`$HOME/.gitconfig`**
  - With ***--local*** option, reads/writes from current repository, aka **`.git/config`**

Getting started with `git`

First-time `git` setup

- Set your `git` identity (globally)
  - **Important!** Name & Email will be used in each commit

```
$ git config --global user.name “John Doe”

$ git config --global user.email johndoe@example.com
```

- Additional configurations..
  - Set your preferred editor (`nano`, `vim` ..)
  - Disclaimer: only choose `vim` if you are familiar

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

Getting started with `git`

Check your Settings

- List all of them:

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
...
```

- Print specific configuration parameter:

```
$ git config user.name
John Doe
```

Ask `git` for Help

- Quick reference of a command

```
# git <verb> -h
$ git config -h
```

- Manpage (full documentation)

```
# git help <verb>
$ git help config

# git <verb> --help
$ git config --help

# man git-<verb>
$ man git-config
```

Setting `git`'s default branch name..

Branching is more advanced topic that we will cover in a dedicated section of this tutorial

- Curious fact: in 2020, GitHub moved from previous `master` to `main`
  - Reasoning: https://sfconservancy.org/news/2020/jun/23/gitbranchname/
  - Implemented in:
    - Git >= 2.28.0 (July 2020)
    - Github.com (Oct 2020)
    - Gitlab.com (May 2021)

```
$ git --version
```

Ensure that `main` is always being used as the default branch name:

```
$ git config --global init.defaultBranch main
```

Git & GitHub in a nutshell

If all you remember is..

- **GitHub**
  - *Not a VCS solution by itself*, hosts repositories that are managed with `git`

Git & GitHub in a nutshell

If all you remember is..

- **GitHub**
  - *Not a VCS solution by itself*, hosts repositories that are managed with `git`
  - Offers *additional functionality* on top of `git` repositories, to foster:
    - Collaboration: Pull Requests through forking or cloning
    - Project Management: Issue tracking, GitHub Pages (web), wiki

Git & GitHub in a nutshell

If all you remember is..

- **GitHub**
  - *Not a VCS solution by itself*, hosts repositories that are managed with `git`
  - Offers *additional functionality* on top of `git` repositories, to foster:
    - Collaboration: Pull Requests through forking or cloning
    - Project Management: Issue tracking, GitHub Pages (web), wiki
- **`git`**
  - The Three States: only cares about *tracked* files (modified, staged, committed)

Git & GitHub in a nutshell

If all you remember is..

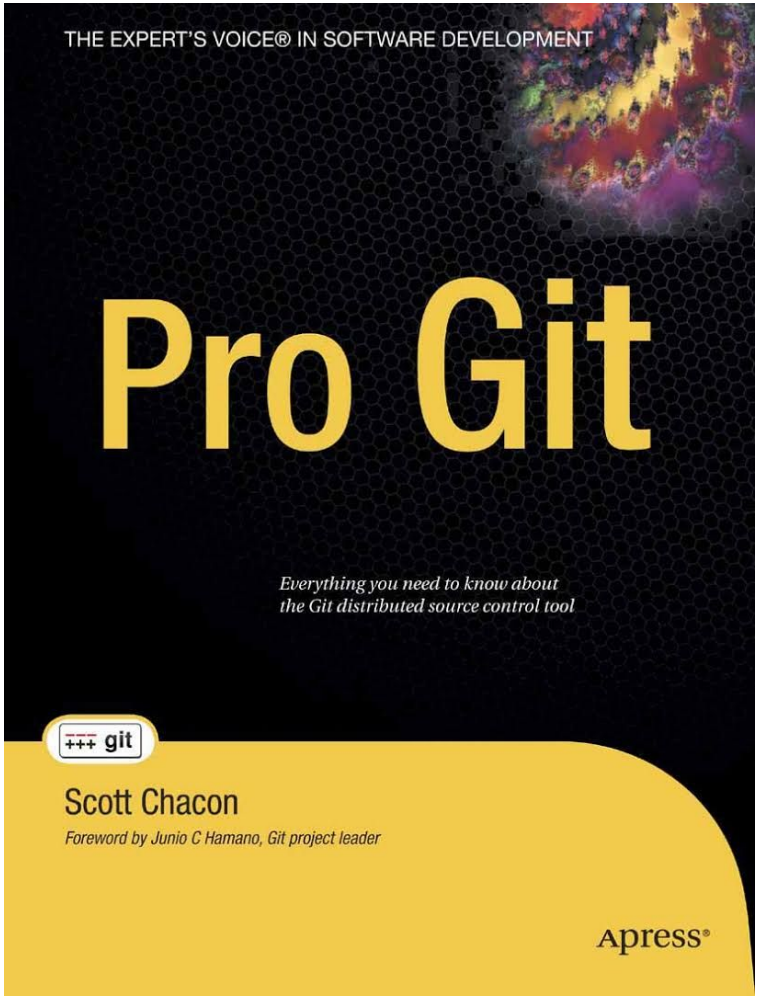
- **GitHub**
  - *Not a VCS solution by itself*, hosts repositories that are managed with `git`
  - Offers *additional functionality* on top of `git` repositories, to foster:
    - Collaboration: Pull Requests through forking or cloning
    - Project Management: Issue tracking, GitHub Pages (web), wiki
- **`git`**
  - The Three States: only cares about *tracked* files (modified, staged, committed)
  - The Three Sections:
    - *Working directory*: current working version
    - *Staging area*: contains the file/s that will go in the next commit
    - `.git` *directory*: contains the data (local DB) for `git` usage

Outline

1. Version Control Systems (VCS)
2. Git and GitHub
3. **Hands-on (basics)**

*This tutorial is based on **Pro Git** by Scott Chacon (2nd edition, 2014)*

Free ebook:
https://git-scm.com/book/en/v2

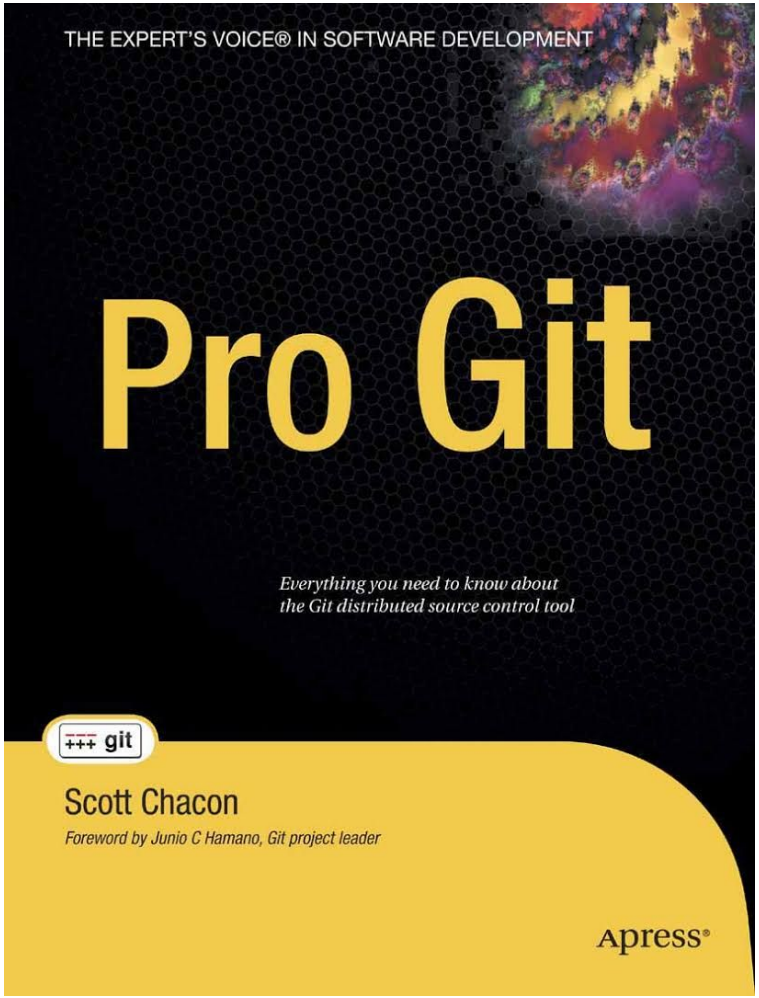


*This tutorial is based on **Pro Git** by Scott Chacon (2nd edition, 2014)*

Free ebook:
https://git-scm.com/book/en/v2

Tons of git tutorials & forums out there:

- https://stackoverflow.com
- https://ohshitgit.com/
- ...



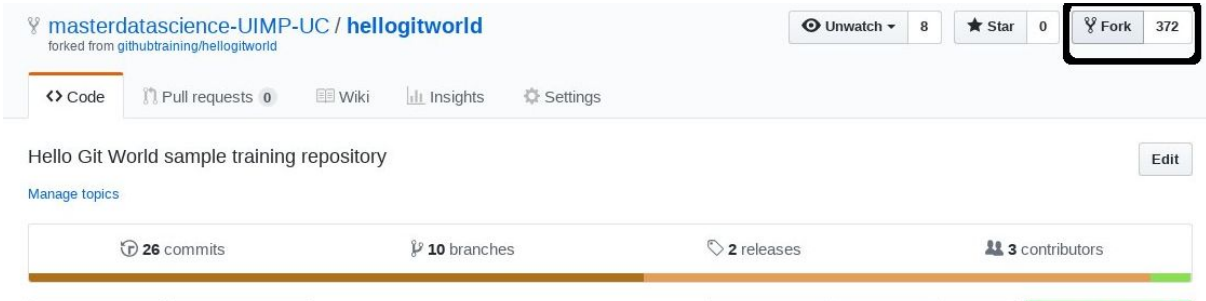
Fork & Clone from GitHub

Sample workflow for delivering exercises..

1. You will be given a GitHub repository URL with the exercises
2. You will fork it to your GitHub personal account
3. You will clone your fork to your system
4. You will work on the exercises, committing & pushing changes
5. You will provide the fork URL to your teacher

Fork a repository in GitHub

- A fork is a **copy of a repository**
- Freely *experiment with changes without affecting the original project*
- Go to the URL of the project to fork from and click on Fork



Fork & Clone from GitHub

A possible workflow for the exercises in the master..

1. You will be given a GitHub repository URL with the exercises
2. You will fork it to your GitHub personal account
3. You will clone your fork to your system
4. You will work on the exercises, committing & pushing changes
5. You will provide the fork URL to your teacher

Let's do an example..

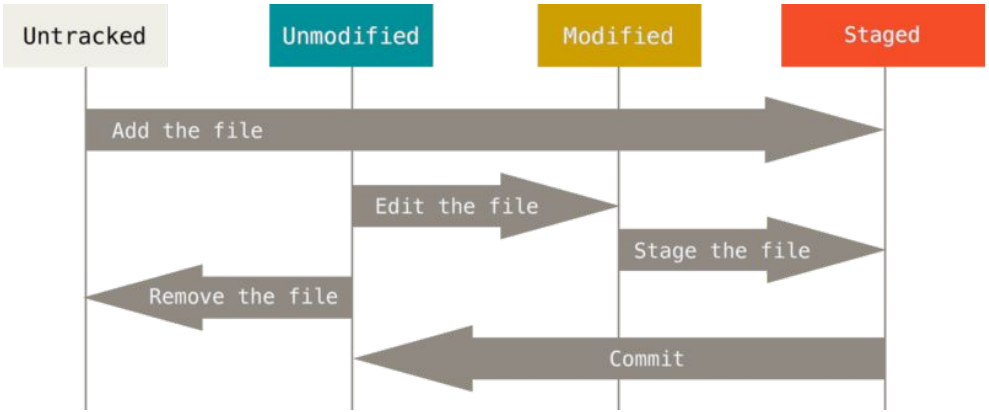
1. (Web) Login to https://github.com
2. (Web) Fork this repository: https://github.com/masterdatascience-UIMP-UC/hellogitworld
   - Use your personal account as the target space
3. (Terminal) Clone your fork with ***git clone*** command

```
# Full copy (files, Git directory, ..) to 'hellogitworld' dir in current path
$ git clone https://github.com/<your-account-name>/hellogitworld

# Enter in the 'hellogitworld' directory just created and list the files included
$ cd hellogitworld
$ ls -l
```

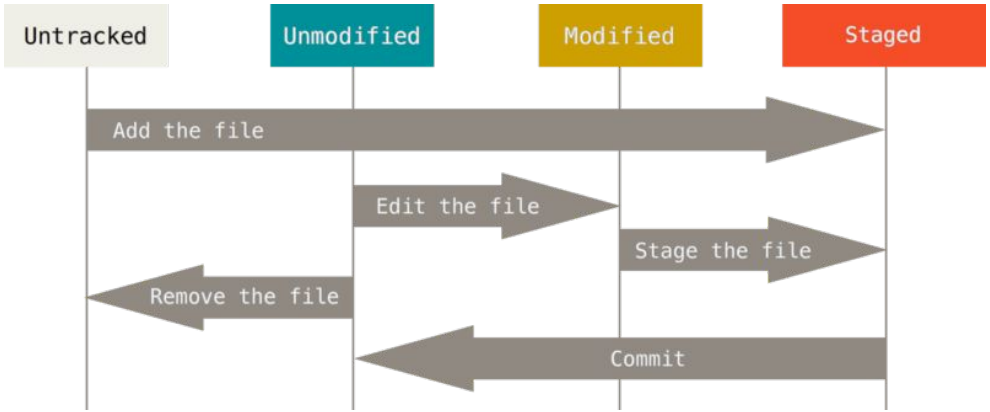
Checking the status of your files

- Status of files
  a. *Untracked*: `git` does not know about these files
  b. *Tracked*: `git` knows about
     i. *Unmodified*
     ii. *Modified*
     iii. *Staged*
- Workflow
  a. Edit files -> *Modified*
  b. Stage the required files -> *Staged*
  c. Commit them -> *Unmodified*



Checking the status of your files

- Status of files
  a. *Untracked*: `git` does not know about these files
  b. *Tracked*: `git` knows about
     i. *Unmodified*
     ii. *Modified*
     iii. *Staged*
- Workflow
  a. Edit files -> *Modified*
  b. Stage the required files -> *Staged*
  c. Commit them -> *Unmodified*
- ***git status*** command




```
# Inside 'hellogitworld' directory ==
$ git status

On branch main
Your branch is up-to-date with 'origin/main'.
nothing to commit, working directory clean
```

A note on `git status` output..

```
$ git status
On branch main
Your branch is up-to-date with 'origin/main'.
nothing to commit, working directory clean
```

Remotes will be also introduced later on in this tutorial

- For this introductory part, we will consider the default remote as `origin`

A note on `git status` output..

```
$ git status
On branch main
Your branch is up-to-date with 'origin/main'.
nothing to commit, working directory clean
```

`git status` **cares about the working directory**, and thus, it will provide accurate:

- information about the status of the files in the current version
- hints on how to proceed with them

In the example above, no changes are registered yet, so all the files in the working directory match with the current version ⇒ working directory clean

Tracking new files

```
# Create a new file called README
$ echo "My own README file" > README
```

Tracking new files

```
# Create a new file called README
$ echo "My own README file" > README

# Check the repository status
$ git status
On branch main
Your branch is up-to-date with 'origin/main'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

      README

nothing added to commit but untracked files present (use "git add" to track)
```

Tracking new files

```
# Create a new file called README
$ echo "My own README file" > README

# Check the repository status
$ git status
On branch main
Your branch is up-to-date with 'origin/main'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

      README

nothing added to commit but untracked files present (use "git add" to track)
```

- README is ***untracked*** (under *Untracked files*)
  - README was not in the previous snapshot (commit)
- `git` just warns us, it won't do anything until we explicitly request so
  - Let's track the README file

Tracking new files

To track a new file we use ***git add***

```
# Start tracking README file
$ git add README
```

Tracking new files

To track a new file we use ***git add***

```
# Start tracking README file
$ git add README

# Check the repository status
$ git status
On branch main
Your branch is up-to-date with 'origin/main'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

      new file: README
```

Tracking new files

To track a new file we use ***git add***

```
# Start tracking README file
$ git add README

# Check the repository status
$ git status
On branch main
Your branch is up-to-date with 'origin/main'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

      new file: README
```

README is now ***staged*** (under *Changes to be committed*)

- Achieved by ***git add*** command [untracked ⇒ staged]
- At this point we could commit this change, resulting in the README file to be added to the repository content [staged ⇒ unmodified], **but first let's understand staging better..**

Staging modified files (I)

Understanding **staging**..

```
# Same situation as before: README file was added
# We now modify an already tracked/existing file --> fix.txt
$ echo “Fix #1 added -- You can safely remove this line --” >> fix.txt
```

Staging modified files (I)

Understanding **staging**..

```
# Same situation as before: README file was added
# We now modify an already tracked/existing file --> fix.txt
$ echo “Fix #1 added -- You can safely remove this line --” >> fix.txt
$ git status
On branch main
Your branch is up-to-date with 'origin/main'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   fix.txt
```

Staging modified files (I)

Understanding **staging**..

```
# Same situation as before: README file was added
# We now modify an already tracked file --> fix.txt
$ echo “Fix #1 added -- You can safely remove this line --” >> fix.txt
$ git status
On branch main
Your branch is up-to-date with 'origin/main'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   fix.txt
```

fix.txt is ***unstaged*** (under *Changes not staged for commit*)

- == modified in the working directory but not staged
- As before, let’s use ***git add*** command to stage it

Staging modified files (II)

Understanding **staging**..

```
# Stage fix.txt file
$ git add fix.txt
```

Staging modified files (II)

Understanding **staging**..

```
# Stage fix.txt file
$ git add fix.txt
# Check the status
$ git status
On branch main
Your branch is up-to-date with 'origin/main'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   fix.txt
```

Staging modified files (II)

Understanding **staging**..

```
# Stage fix.txt file
$ git add fix.txt
# Check the status
$ git status
On branch main
Your branch is up-to-date with 'origin/main'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   fix.txt
```

Both README and fix.txt are ***staged*** and will go into the next commit

Staging modified files (II)

Understanding **staging**..

```
# Stage fix.txt file
$ git add fix.txt
# Check the status
$ git status
On branch main
Your branch is up-to-date with 'origin/main'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   fix.txt
```

Both README and fix.txt are ***staged*** and will go into the next commit

..but you just realize about a last minute change required in fix.txt before you commit it..

Staging modified files (III)

Understanding **staging**..

```
# Add the last minute change to fix.txt
$ echo “Fix #2: Very important fix added -- You can safely remove this line --” >> fix.txt
```

Staging modified files (III)

Understanding **staging**..

```
# Add the last minute change to fix.txt
$ echo “Fix #2: Very important fix added -- You can safely remove this line --” >> fix.txt
# Check the status
$ git status
On branch main
Your branch is up-to-date with 'origin/main'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   fix.txt

Changes not staged for commit:
 (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   fix.txt
```

Staging modified files (III)

Understanding **staging**..

```
# Add the last minute change to fix.txt
$ echo “Fix #2: Very important fix added -- You can safely remove this line --” >> fix.txt
# Check the status
$ git status
On branch main
Your branch is up-to-date with 'origin/main'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   fix.txt

Changes not staged for commit:
 (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   fix.txt
```

fix.txt is ***unstaged*** and ***staged*** at the same time **?¿**

- If you commit now, only the first change in fix.txt (“Fix #1..”) will be considered
- You need to ***git add*** fix.txt to stage the new change, so it goes in the next commit

Staging modified files (IV)

Understanding **staging**..

```
# Stage the last change of fix.txt == ‘Fix #2..’
$ git add fix.txt
```

Staging modified files (IV)

Understanding **staging**..

```
# Stage the last change of fix.txt == ‘Fix #2..’
$ git add fix.txt
# Check the status
$ git status
On branch main
Your branch is up-to-date with 'origin/main'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   fix.txt
```

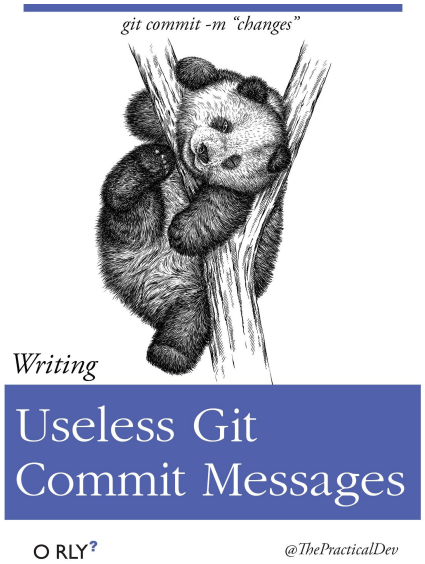
Committing your changes (I)

To commit a new change we use ***git commit***

Committing your changes (I)

To commit a new change we use ***git commit***:

- A commit **creates a new version** (creates a snapshot, remember?)
  - Anything not staged won't be part of the commit
  - Commits can be compared/reverted
  - A commit may involve changes to multiple files
    - SHOULD be **atomic** and **topical** → *"A commit should contain related changes and nothing but related changes" (codefoster.com)*

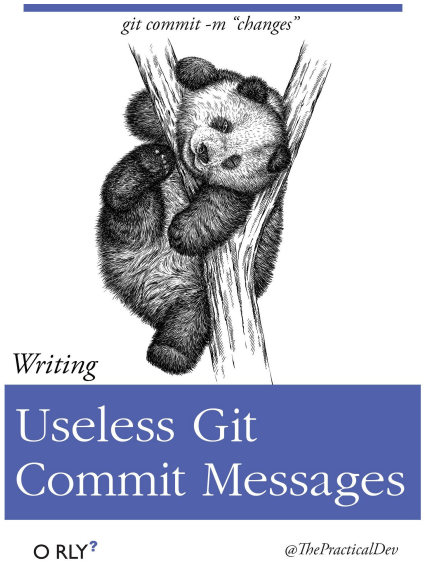


Committing your changes (I)

To commit a new change we use ***git commit***:

- A commit **creates a new version** (makes a snapshot of your project)
  - Anything not staged won't be part of the commit
  - Commits can be compared/reverted
  - A commit may involve changes to multiple files
    - SHOULD be **atomic** and **topical** → *"A commit should contain related changes and nothing but related changes" (codefoster.com)*

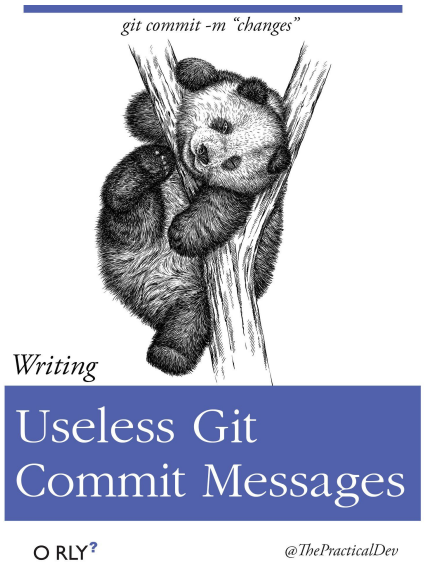
```
# Ou changes are all staged, so we commit..
$ git commit
```



Committing your changes (I)

To commit a new change we use ***git commit***:

- A commit **creates a new version** (makes a snapshot of your project)
  - Anything not staged won't be part of the commit
  - Commits can be compared/reverted
  - A commit may involve changes to multiple files
    - SHOULD be **atomic** and **topical** → *"A commit should contain related changes and nothing but related changes" (codefoster.com)*




```
# Ou changes are all staged, so we commit..
$ git commit
```

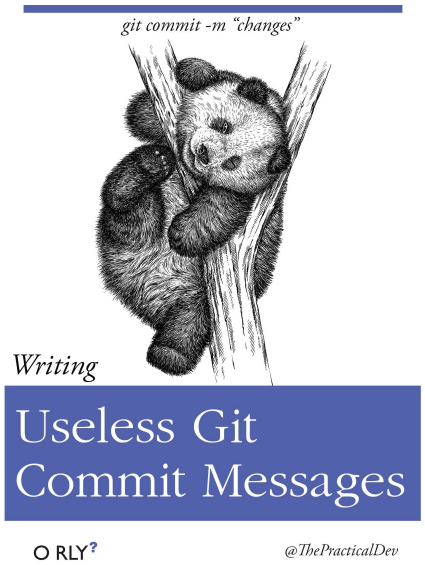
With no options, git commit opens the default editor (remember git config), wit

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch main
# Your branch is up-to-date with 'origin/main'.
#
# Changes to be committed:
#     new file:   README
#     modified:   fix.txt
```

Committing your changes (I)

To commit a new change we use ***git commit***:

- A commit **creates a new version** (makes a snapshot of your project)
  - Anything not staged won't be part of the commit
  - Commits can be compared/reverted
  - A commit may involve changes to multiple files
    - SHOULD be **atomic** and **topical** → *"A commit should contain related changes and nothing but related changes" (codefoster.com)*




```
# Ou changes are all staged, so we commit..
$ git commit
```

With no options, git commit opens the default editor (remember git config), wit

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch main
# Your branch is up-to-date with 'origin/main'.
#
# Changes to be committed:
#     new file:   README
#     modified:   fix.txt
```

- A **descriptive commit message** must be added to the blank line above the default text
- The default content can be left, just to remind in the future what files you have changed

Committing your changes (II)

A one-liner approach to git commit

```
# Our changes are all staged, so we commit..
$ git commit -m “New README and the two first fixes documented”
[main 1a90007] New README and the two first fixes documented
 2 files changed, 3 insertions(+)
 create mode 100644 README
```

Committing your changes (II)

A one-liner approach to git commit

```
# Our changes are all staged, so we commit..
$ git commit -m “New README and the two first fixes documented”
[main 1a90007] New README and the two first fixes documented
 2 files changed, 3 insertions(+)
 create mode 100644 README
```

Your first commit is done!

- To branch `main`
- With ID `1a90007`

Moving/renaming files

To move or rename tracked files we use ***git mv***:

```
# Let’s rename the README file..
$ git mv README README.to_delete
```

Moving/renaming files

To move or rename tracked files we use ***git mv***:

```
# Let’s rename the README file..
$ git mv README README.to_delete

$ git status
On branch main
Your branch is up-to-date with 'origin/main'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README -> README.to_delete
```

Removing files

To remove a tracked file we use ***git rm***:

```
# Let’s remove README.to_delete file
$ git rm README.to_delete
error: the following file has changes staged in the index:
    README.to_delete
(use --cached to keep the file, or -f to force removal)
```

Removing files

To remove a tracked file we use ***git rm***:

```
# Let’s remove README.to_delete file
$ git rm README.to_delete
error: the following file has changes staged in the index:
    README.to_delete
(use --cached to keep the file, or -f to force removal)

$ git rm -f README.to_delete
rm ‘README.to_delete’

$ git status
On branch main
Your branch is up-to-date with 'origin/main'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:    README
```

The **next commit will eliminate the file** and will be no longer tracked ⇒ **Commit this change with an appropriate message**

Viewing Commit History

To view the commit history of our project we use ***git log***:

Viewing Commit History

To view the commit history of our project we use ***git log***:

```
$ git log
commit 1a900077fdeffd73e0c7412a731949c540c78250 (HEAD -> main)
Author: Pablo Orviz <orviz@ifca.unican.es>
Date:   Mon Oct 1 15:33:36 2018 +0000

    New README and the two first fixes documented.

commit ef7bebf8bdb1919d947afe46ab4b2fb4278039b3 (origin/main, origin/HEAD)
Author: Jordan McCullough <jordan@github.com>
Date:   Fri Nov 7 11:27:19 2014 -0700

    Fix groupId after package refactor

commit ebbbf773431ba07510251bb03f9525c7bab2b13a
Author: Jordan McCullough <jordan@github.com>
Date:   Wed Nov 5 13:00:35 2014 -0700

    Update package name, directory

commit 45a30ea9afa413e226ca8614179c011d545ca883
Author: Jordan McCullough <jordan@github.com>
Date:   Wed Nov 5 12:59:55 2014 -0700

    Update package name, directory

commit 9805760644754c38d10a9f1522a54a4bdc00fa8a
Author: Jordan McCullough <jordan@github.com>
Date:   Wed Nov 5 12:19:02 2014 -0700

    Fix YAML name-value pair missing space
```

- Most recent commits show up first
- Info per commit:
  - commit ID
  - Author's name & email
  - Date
  - Author's commit message

Viewing Commit History

git log uses UNIX pager:

```
$ git log
commit 1a900077fdeffd73e0c7412a731949c540c78250 (HEAD -> main)
Author: Pablo Orviz <orviz@ifca.unican.es>
Date:   Mon Oct 1 15:33:36 2018 +0000

    New README and the two first fixes documented.

commit ef7bebf8bdb1919d947afe46ab4b2fb4278039b3 (origin/main, origin/HEAD)
Author: Jordan McCullough <jordan@github.com>
Date:   Fri Nov 7 11:27:19 2014 -0700

    Fix groupId after package refactor

commit ebbbf773431ba07510251bb03f9525c7bab2b13a
Author: Jordan McCullough <jordan@github.com>
Date:   Wed Nov 5 13:00:35 2014 -0700

    Update package name, directory
:
```

Pager prompt

Navigating `git` pager..

- Move upwards/downwards: **cursor keys**
- Quit: **press ‘q’ key**

How to disable the `git` pager?

- For a specific `git` command

```
$ git --no-pager log
```

- For _all_ `git` commands

```
$ export GIT_PAGER=cat
```

If we wanted to re-enable the pager:

```
$ unset GIT_PAGER
```

Viewing Commit History

git log command has several interesting options..

Viewing Commit History

git log command has several interesting options..

1. Filtering

```
$ git log -3
commit 1a900077fdeffd73e0c7412a731949c540c78250 (HEAD -> main)
Author: Pablo Orviz <orviz@ifca.unican.es>
Date:   Mon Oct 1 15:33:36 2018 +0000

    New README and the two first fixes documented.

commit ef7bebf8bdb1919d947afe46ab4b2fb4278039b3 (origin/main, origin/HEAD)
Author: Jordan McCullough <jordan@github.com>
Date:   Fri Nov 7 11:27:19 2014 -0700

    Fix groupId after package refactor

commit ebbbf773431ba07510251bb03f9525c7bab2b13a
Author: Jordan McCullough <jordan@github.com>
Date:   Wed Nov 5 13:00:35 2014 -0700

    Update package name, directory

$ git log --since="2 weeks ago"
commit 1a900077fdeffd73e0c7412a731949c540c78250 (HEAD -> main)
Author: Pablo Orviz <orviz@ifca.unican.es>
Date:   Mon Oct 1 15:33:36 2018 +0000

    New README and the two first fixes documented.
```

Viewing Commit History

git log command has several interesting options..

2. Formatting output

```
$ git log --pretty=oneline
1a900077fdeffd73e0c7412a731949c540c78250 (HEAD -> main) New README and the two
first fixes documented.
ef7bebf8bdb1919d947afe46ab4b2fb4278039b3 (origin/main, origin/HEAD) Fix groupId
after package refactor
ebbbf773431ba07510251bb03f9525c7bab2b13a Update package name, directory
```

Viewing Commit History

git log command has several interesting options..

3. Checking modifications

```
$ git log -1 --patch
commit 1a900077fdeffd73e0c7412a731949c540c78250 (HEAD -> main)
Author: Pablo Orviz <orviz@ifca.unican.es>
Date:   Mon Oct 1 15:33:36 2018 +0000

    New README and the two first fixes documented.

diff --git a/README b/README
new file mode 100644
index 0000000..29afdfa
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My own README file
diff --git a/fix.txt b/fix.txt
index e69de29..3e6b239 100644
--- a/fix.txt
+++ b/fix.txt
@@ -0,0 +1,2 @@
+Fix #1 added -- You can safely remove this line --
+Fix #2: Very important fix added -- You can safely remove this line --
```

Viewing Commit History

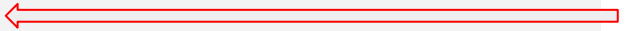
git log command has several interesting options..

3. Checking modifications

```
$ git log -1 --patch
commit 1a900077fdeffd73e0c7412a731949c540c78250 (HEAD -> main)
Author: Pablo Orviz <orviz@ifca.unican.es>
Date:   Mon Oct 1 15:33:36 2018 +0000

    New README and the two first fixes documented.

diff --git a/README b/README
new file mode 100644
index 0000000..29afdfa
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My own README file
diff --git a/fix.txt b/fix.txt
index e69de29..3e6b239 100644
--- a/fix.txt
+++ b/fix.txt
@@ -0,0 +1,2 @@
+Fix #1 added -- You can safely remove this line --
+Fix #2: Very important fix added -- You can safely remove this line --
```



Returns the diffs of changed files

Viewing Commit History

git log command has several interesting options..

3. Checking modifications

```
$ git log -1 --patch
commit 1a900077fdeffd73e0c7412a731949c540c78250 (HEAD -> main)
Author: Pablo Orviz <orviz@ifca.unican.es>
Date:   Mon Oct 1 15:33:36 2018 +0000

    New README and the two first fixes documented.

diff --git a/README b/README
new file mode 100644
index 0000000..29afdfa
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My own README file
diff --git a/fix.txt b/fix.txt
index e69de29..3e6b239 100644
--- a/fix.txt
+++ b/fix.txt
@@ -0,0 +1,2 @@
+Fix #1 added -- You can safely remove this line --
+Fix #2: Very important fix added -- You can safely remove this line --
```

git diff command is very useful to compare/show diffs between commits..

```
$ git diff <commit-id-1>..<commit-id-2>
```

Working with remotes

So far we have been working locally on our computer..

Working with remotes

So far we have been working locally on our computer..

Remote repositories are ..

- **Alternative copies** of the repository you are working with
- **Hosted on the Internet** (e.g. GitHub)
- Either copies where you have **write** (own repos, forks) **or read-only** (upstream) **permissions**
- Required to perform **contributions** (foster collaboration)
  1. Add & Remove remotes: with ***git remote***
  2. Push & Pull changes to remote repositories, with ***git push*** and ***git pull***

Listing your remotes

```
# Lists shortnames of each remote
$ git remote
origin
```

Listing your remotes

```
# Lists shortnames of each remote
$ git remote
origin

# Lists shortnames of each remote together with their URLs
$ git remote -v
origin  https://github.com/orviz/hellogitworld (fetch)
origin  https://github.com/orviz/hellogitworld (push)
```

- **`origin` is the default name** `git` gives to the remote repository you have cloned from
- ***`git clone`*** adds the `origin` remote for us

Adding remote repositories

For instance, we could add the upstream repository we forked from

https://github.com/masterdatascience-UIMP-UC/hellogitworld

```
# Adds a remote repository named ‘upstream’
$ git remote add upstream https://github.com/masterdatascience-UIMP-UC/hellogitworld
```

Adding remote repositories

For instance, we could add the upstream repository we forked from

https://github.com/masterdatascience-UIMP-UC/hellogitworld

```
# Adds a remote repository named ‘upstream’
$ git remote add upstream https://github.com/masterdatascience-UIMP-UC/hellogitworld

# List remotes
$ git remote -v
origin https://github.com/orviz/hellogitworld (fetch)
origin https://github.com/orviz/hellogitworld (push)
upstream  https://github.com/masterdatascience-UIMP-UC/hellogitworld (fetch)
upstream  https://github.com/masterdatascience-UIMP-UC/hellogitworld (push)
```

Now we can refer to upstream for any operation we may need to do with this remote repository

Fetch and pull from your remotes

git fetch allows us to retrieve data from remote repositories

- Format: ***git fetch <remote>***
- Use:
  - In collaborative scenarios, your local repository might be outdated
  - By fetching, `git` **gets any new work** committed since the last time you cloned or updated the remote content
- Updates .git database, ***git fetch*** **does not merge the remote content with your current version**
  - You would need to merge it manually with ***git merge*** command

Fetch and pull from your remotes

git fetch allows us to retrieve data from remote repositories

- Format: ***git fetch <remote>***
- Use:
  - In collaborative scenarios, your local repository might be outdated
  - By fetching, `git` **gets any new work** committed since the last time you cloned or updated the remote content
- Updates .git database, ***git fetch*** **does not merge the remote content with your current version**

You would need to merge it manually with *git merge* command or better use *git pull*

Fetch and pull from your remotes

git pull can be seen as a combination of ***git fetch*** + ***git merge***

- Format: ***git pull <remote> <branch>***
- Will try to **automatically fetch and merge** the remote version into the current version you are currently working on (locally)
- More convenient, although may not work in all the scenarios

Before pulling, check first if you have uncommitted (modified, staged) files

```
# Check first if you have uncommitted (modified, staged) files before pulling
$ git status
(..)

# Pulls from remote repository labelled as 'origin'
$ git pull origin main
From https://github.com/orviz/hellogitworld
 * branch            main     -> FETCH_HEAD
Already up to date.
```

Push to your remotes

git push allows us to send our committed changes to a remote repository

- Format: ***git push <remote> <branch>***

Push to your remotes

git push allows us to send our committed changes to a remote repository

- Format: ***git push <remote> <branch>***

```
# Pushes changes to the remote labelled as ‘origin’ (branch: ‘main’)
$ git push origin main
```

Keep in mind that..

- This command will only work if we have **write permissions** in the remote repository
- It is a good practice to **pull right before push**

The files to be pushed remote can be checked with ***git diff***

```
# List files that will be pushed to ‘origin’ remote & branch ‘main’
$ git diff --stat --cached origin/main
```

Push to your remotes

git push allows us to send our committed changes to a remote repository

- Format: ***git push <remote> <branch>***

```
# Pushes changes to the remote labelled as ‘origin’ (branch: ‘main’)
$ git push origin main
```

Keep in mind that..

- This command will only work if we have **write permissions** in the remote repository
- It is a good practice to **pull right before push**

The files to be pushed remote can be checked with ***git diff***

```
# List files that will be pushed to ‘origin’ remote & branch ‘main’
$ git diff --stat --cached origin/main
```

[Exercise] Understanding permissions

- Push committed changes to `origin` and `upstream` remotes
- Compare the output between them

Before pushing to your remote repository..

GitHub requires authentication, so we will be asked for ~~user/password~~ user/token for every remote operation (when using *https*)

Create a GitHub token

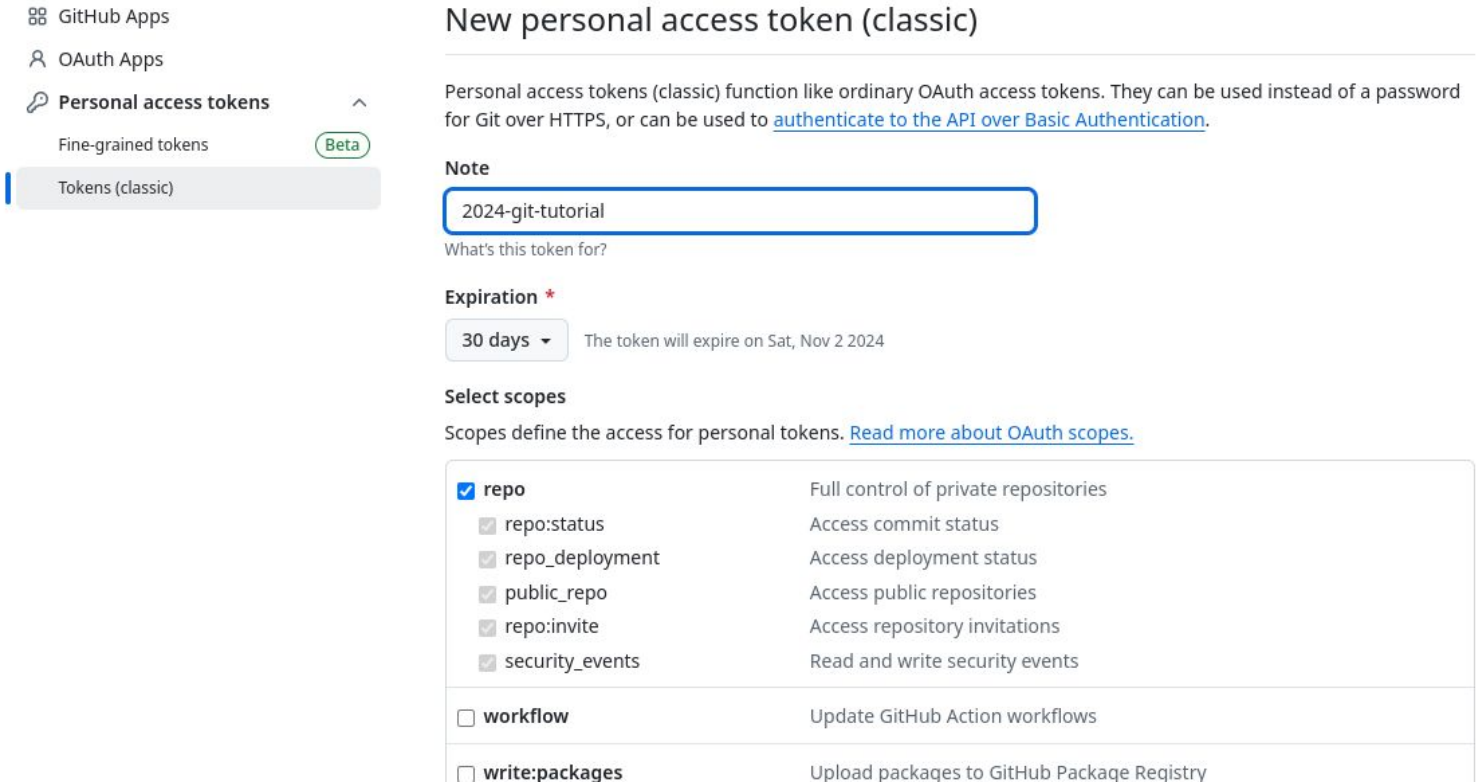
https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token

How? *From your GitHub account, go to*

Settings *=>* ***Developer Settings*** *=>* ***Personal Access Token*** *=>* ***Generate New Token*** *(classic) =>* ***Fill up the form*** *(mark "repo" check box) => click* ***Generate token*** *=>* ***Copy the generated Token****, it will be something like*
ghp_sFhFsSHhTzMDreGRLjmks4Tzuzgthdvfsrta

What to do with the token? You will be asked the first time you interact with the remote repository

Before pushing to your remote repository..



Before pushing to your remote repository..

GitHub requires authentication, so we will be asked for ~~user/password~~ user/token for every remote operation

Caching user credentials

- Enter ~~user/password~~ user/token once & tell `git` to cache them for a given timeframe
- Cache forever? → Not a good practice
  - Best approach → use a specific tool to handle creds, i.e. GCM Core (also GitHub CLI)

```
# Enable credential's memory cache (default: 15 minutes)
$ git config --global credential.helper cache

# Set a custom value for cache timeout (e.g. 1 hour)
$ git config --global credential.helper 'cache --timeout=3600'

# THIS COURSE's APPROACH: cache them for ~2 days
$ git config --global credential.helper 'cache --timeout=200000'
```

Push to your remotes

Check that the changes/commits have been uploaded into your remote repository @GitHub

https://github.com/<your-github-account>/hellogitworld

If you are unsure where you push it, double-check where does your `origin` remote is pointing to

Exercise with remotes (I)

Setting up a GitHub repository from a local directory

- So far we have worked with already-existing repositories (fork & clone)
  - Now we present an alternative way to create a `git` repository and upload/push it to GitHub
- With ***git init*** we can **initialize whatever folder** in our system **as a `git` repository**:

```
# Create the directory to host the repository
$ mkdir /home/user/my_project

# Move to the directory meant to be the repository
$ cd /home/user/my_project

# Convert directory to Git repository (creates .git directory)
$ git init
```

- `git` **does not track any file** (from the local folder) **by default**
  - Need to tell `git` about the files we want to start tracking
- There is **no remote repository** in GitHub
  - **Need to create one!**

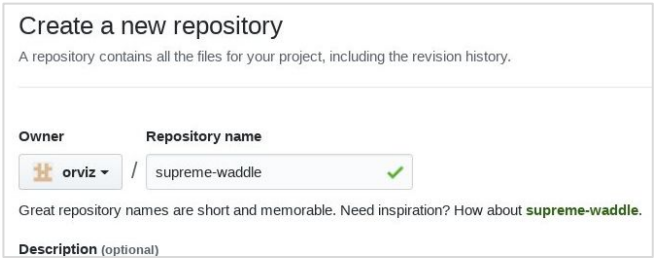
Remember!

```
$ git config --global init.defaultBranch main
```

Exercise with remotes (II)

Setting up a GitHub repository from a local directory

1. (GitHub) Create a new repository in your personal account
   - Repositories > New



2. (Terminal) Create a Git repository from a directory in your system, with git init

```
# Start in your $HOME directory
$ cd
# Directory name does not need to match the GitHub repository name
$ mkdir mygithubrepo
$ cd mygithubrepo
$ git init
Initialized empty Git repository in /home/jovyan/mygithubrepo/.git/
```

3.

```
# Create README.md file (or with `nano README.md`)
$ echo “Welcome to my GitHub repository! It was created from a local directory” > README.md
```

Exercise with remotes (III)

Setting up a GitHub repository from a local directory

4. (Terminal) Add it to be part of the commit (add to the staging area) and commit

```
# HACK: One-liner would be `git commit -a -m “The first commit in my GitHub repo”`
$ git add README.md
$ git commit -m “The first commit in my GitHub repo”
```

5. (Terminal) Add your GitHub repository as remote

```
# Copy the URL from GitHub (web browser)
$ git remote add origin https://github.com/orviz/supreme-waddle
$ git remote -v
origin  https://github.com/orviz/supreme-waddle (fetch)
origin  https://github.com/orviz/supreme-waddle (push)
```

6. (Terminal) Push the changes to GitHub

```
# Push the commit to the recently added ‘origin’ remote
$ git push origin main
```

7. (GitHub) Check in the repo that you actually have a new README.md file!

If all you remember is..

- A **fork** is a GitHub feature that copies a remote repository in your local account
  - Forks are the most common way to contribute to external repos (in combination with PRs)
- ***git clone*** creates a local copy of a remote repository
  - Sets the remote repository as origin
  - Sets the current branch as main
- ***git add*** is a multipurpose command:
  - Begin tracking new files
  - Stage files
- ***git commit*** gathers all the staged changes and creates a new version/snapshot
  - ***git commit -a*** is shortcut to ***git add*** + ***git commit***
    - Warning: **All modified files will be added** to the commit
- ***git pull*** gets last updates from a remote repository and merges them with the current version in the local working directory
- ***git push*** sends committed changes to a remote repository

Hands-on

Collaborative work

GitHub Pull requests

GitHub feature that allows you to propose changes to others

- Primary source of collaboration

GitHub Pull requests

GitHub feature that allows you to propose changes to others

- Primary source of collaboration
- *“Once a PR is opened, you can **discuss and review** the potential changes with collaborators and add follow-up commits before your changes are merged into the base branch”* (github.com)

GitHub Pull requests

GitHub feature that allows you to propose changes to others..

- Primary source of collaboration
- *“Once a PR is opened, you can **discuss and review** the potential changes with collaborators and add follow-up commits before your changes are merged into the base branch”* (github.com)
- Owner/s of the repository are the one/s that eventually accept/reject the change
    - If accepted, the change will be typically merged into the owner’s production branch (remember: `main` by default)

GitHub Pull requests

The importance of review..

- Owner/s, collaborator/s and (if public) any external user/expert can comment and suggest further modifications, e.g.:
  - Goal/scope of the change
  - If source code, any suggestion to optimize the execution (efficiency, efficacy) of the code
  - Commit/s message/s description

GitHub Pull requests

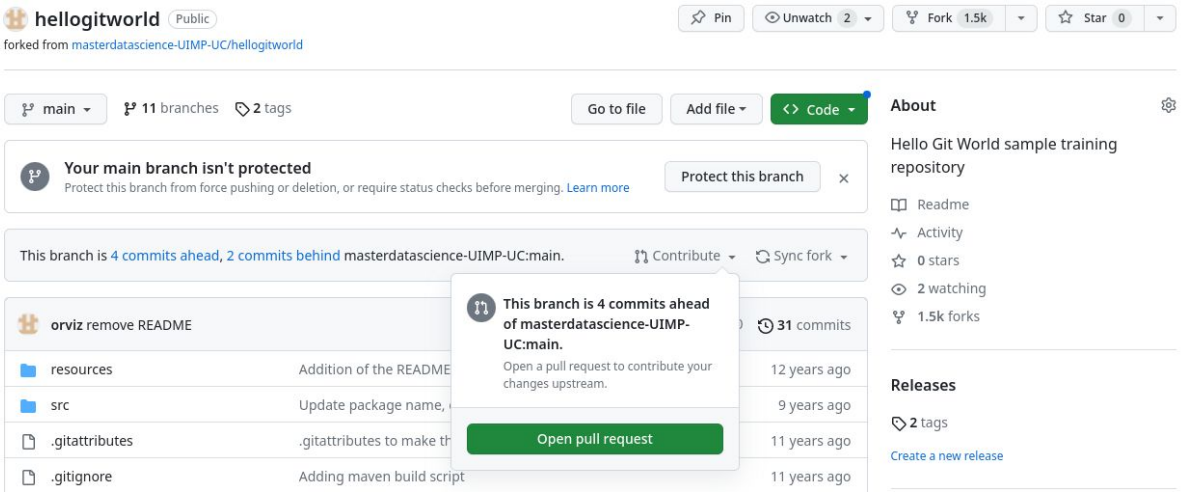
The importance of review..

- Owner/s, collaborator/s and (if public) any external user/expert can comment and suggest further modifications, e.g.:
  - Goal/scope of the change
  - If source code, any suggestion to optimize the execution (efficiency, efficacy) of the code
  - Commit/s message/s description
- Code review process results in the change being merged into the production version (commonly main branch) ⇒ should be addressed carefully!
  - Usually is complemented with the execution of a set of automatic tests => GitHub Integrations

Exercise: Create Pull Request from a fork (I)

[Exercise] Create a Pull Request from fork

- Use your **hellogitworld**'s fork repository
- Head to GitHub & Create a PR out of the fork

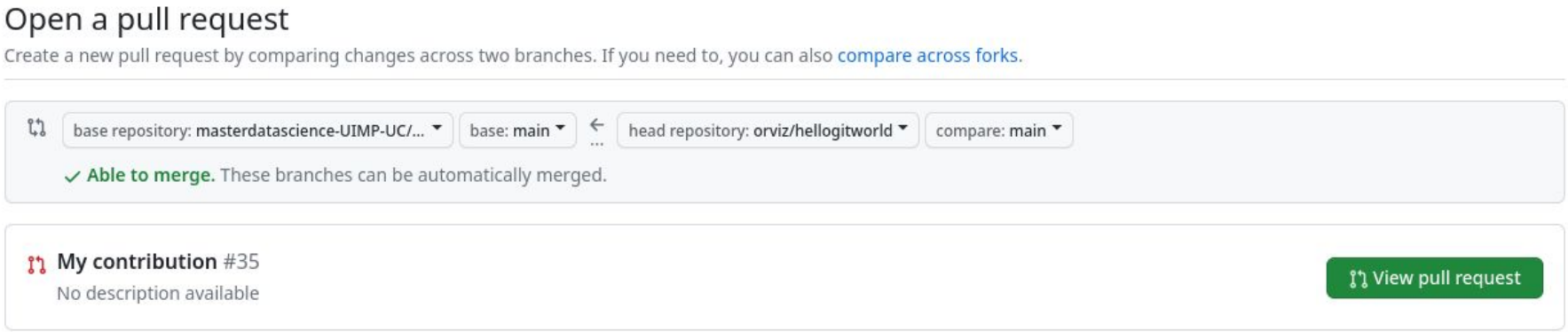


Click on 'Open pull request'

Exercise: Create Pull Request from a fork (II)

[Exercise] Create a Pull Request from fork

- Use your **hellogitworld**'s fork repository
- Head to GitHub & Create a PR out of the fork



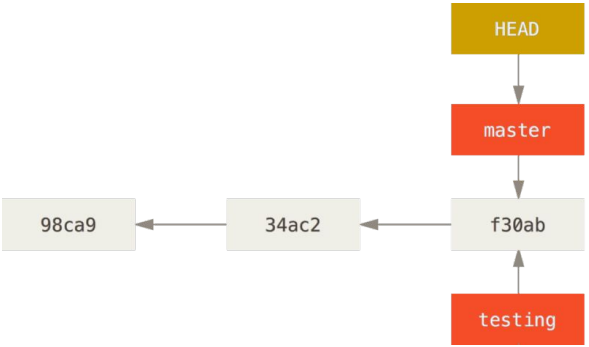
Before clicking "Create pull request"..double-check that the remote/branch combination is ok for both the target (to) and source (from)

Git Branching

"*Diverge from the main line of development (`main` branch) and continue to do the work without messing with that main line*" (Pro Git)

Default branch name in `git` is `main`

- *Not a special branch*, just like any other



Common operations with branches

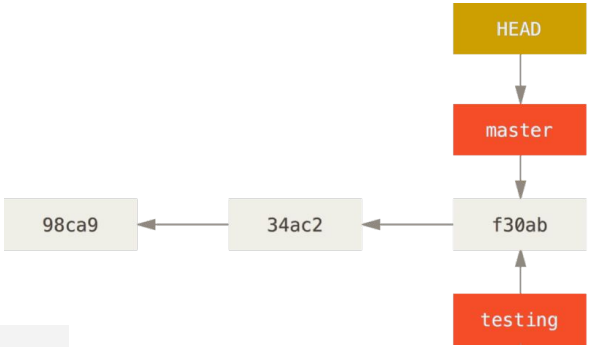
- Check what branch we currently are
  - Format: **git branch**
- Create a branch
  - Format: **git branch <branch-name>**
  - *By default*: the new branch *points to the current commit ID you are on*
    - State explicitly the source branch with **git branch <branch-name> <origin-branch>**

`git` Branching

"Diverge from the main line of development (`main` branch) and continue to do the work without messing with that main line" (Pro Git)

Common operations with branches

- Switching branches
  - Format: **`git checkout <branch-name>`**
- One-liner (create & checkout) with `-b` option




```
# Shorthand for "git branch testing + git checkout testing"
$ git checkout -b testing
```

Let's see how branches can help to carry out clean & safe contributions to repositories..

Clean & safe: `git` Branches + GitHub Pull Requests

When you have write access to the remote repository (forks, own repos)

- Workflow
  1. (Terminal) For each new change, create an individual branch `new-fix` [hint: **`git checkout -b <branch-name>`**]
  2. (Terminal) Make all the needed changes within the branch [hint: **`git add`, `git commit`**]
  3. (Terminal) Once done, push the branch to the remote repository [hint: **`git push`**]
  4. (GitHub) Head to the remote repository webpage
  5. (GitHub) Create a Pull Request from `new-fix` to be merged in `main` branch
  6. (GitHub) Review the Pull Request and merge it if ok

Clean & safe: `git` Branches + GitHub Pull Requests

When you have write access to the remote repository (forks, own repos)

- Workflow
  1. (Terminal) For each new change, create an individual branch `new-fix` [hint: **`git checkout -b <branch-name>`**]
  2. (Terminal) Make all the needed changes within the branch [hint: **`git add`, `git commit`**]
  3. (Terminal) Once done, push the branch to the remote repository [hint: **`git push`**]
  4. (GitHub) Head to the remote repository webpage
  5. (GitHub) Create a Pull Request from `new-fix` to be merged in `main` branch
  6. (GitHub) Review the Pull Request and merge it if ok
- Benefits of this workflow
  - **`main` branch is protected from direct pushes**
    - Via feature branches we isolate new additions/fixes from production/working version in `main`

Clean & safe: `git` Branches + GitHub Pull Requests

When you have write access to the remote repository (forks, own repos)

- Workflow
  1. (Terminal) For each new change, create an individual branch `new-fix` [hint: **`git checkout -b <branch-name>`**]
  2. (Terminal) Make all the needed changes within the branch [hint: **`git add`, `git commit`**]
  3. (Terminal) Once done, push the branch to the remote repository [hint: **`git push`**]
  4. (GitHub) Head to the remote repository webpage
  5. (GitHub) Create a Pull Request from `new-fix` to be merged in `main` branch
  6. (GitHub) Review the Pull Request and merge it if ok
- Benefits of this workflow
  - **`main` branch is protected from direct pushes**
    - Via feature branches we isolate new additions/fixes from production/working version in `main`
- Exercise
  - Based on the workflow above, merge a new change in `main` with a Pull Request
  - **Use the repository you have previously created from the local directory**

Git Branching - Merging

- When we click on `Merge Pull Request` button in GitHub, it runs **`git merge`** for us

```
# Create and switch to a new branch ‘feature-to-merge’
$ git checkout -b feature-to-merge

# Do whatever change (e.g. remove README file) and commit
$ git rm README
$ git commit -m “Remove unused README file”

# Checkout to main and merge
$ git checkout main
$ git merge feature-to-merge
Updating c858518..4b41eff
Fast-forward
 README | 1 -
 1 file changed, 1 deletion(-)
 delete mode 100644 README
```

Git Branching - Merge conflicts

- Occasionally, merging two branches (such as a feature branch into main) does not go smoothly
  - E.g. if we have **changed the same file differently within two branches**

```
# Create and switch to a new branch ‘generate-conflict’
$ git checkout -b generate-conflict

# Add a new line (3rd line) in fix.txt and commit
$ echo “line added in generate-conflict branch” >> fix.txt
$ git add fix.txt
$ git commit -a -m “Added important line”

# Switch to main branch
$ git checkout main

# Add a new line (3rd line) in fix.txt and commit
$ echo “line added in main branch” >> fix.txt
$ git add fix.txt
$ git commit -m “Added very important line”

# Merge branch ‘generate-conflict’ into ‘main’
$ git merge generate-conflict
Auto-merging fix.txt
CONFLICT (content): Merge conflict in fix.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Git Branching - Merge conflicts

To solve conflicts, we have to edit the affected file/s to select the right content and commit

```
$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

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

        both modified:   fix.txt

$ cat fix.txt
otra cosa
otra cosa 2
<<<<<<< HEAD
line added in main branch
=======
line added in generate-conflict branch
>>>>>>> generate-conflict
```

- Conflict markers:
  - `<<<<<` (start conflict)
  - `>>>>>` (ends conflict)
  - `=====` (conflict separation)

```
# Edit fix.txt with ‘nano’ editor and leave just the
# right content
$ cat fix.txt
otra cosa
otra cosa 2
line added in generate-conflict branch

# Mark resolution and commit
$ git add fix.txt
$ git commit -m “Merge branch 'generate-conflict'”
```

A last note on Fork maintenance..

- Since a fork is a copy, it needs to be up-to-date
  - You only have the versions of the files at the time the fork was created
  - If there are changes in the original repo, you may find that your fork is out of sync
- How to update your fork's main branch?

```
# Be sure you are in main branch
$ git checkout main
# 'upstream' remote has to be previously added
$ git fetch upstream
# Find the commit ID you want to revert to
$ git pull upstream main
# Push the updated main to GitHub
$ git push origin main
```

Exercise: after updating a main@upstream branch, try to update your fork's main branch

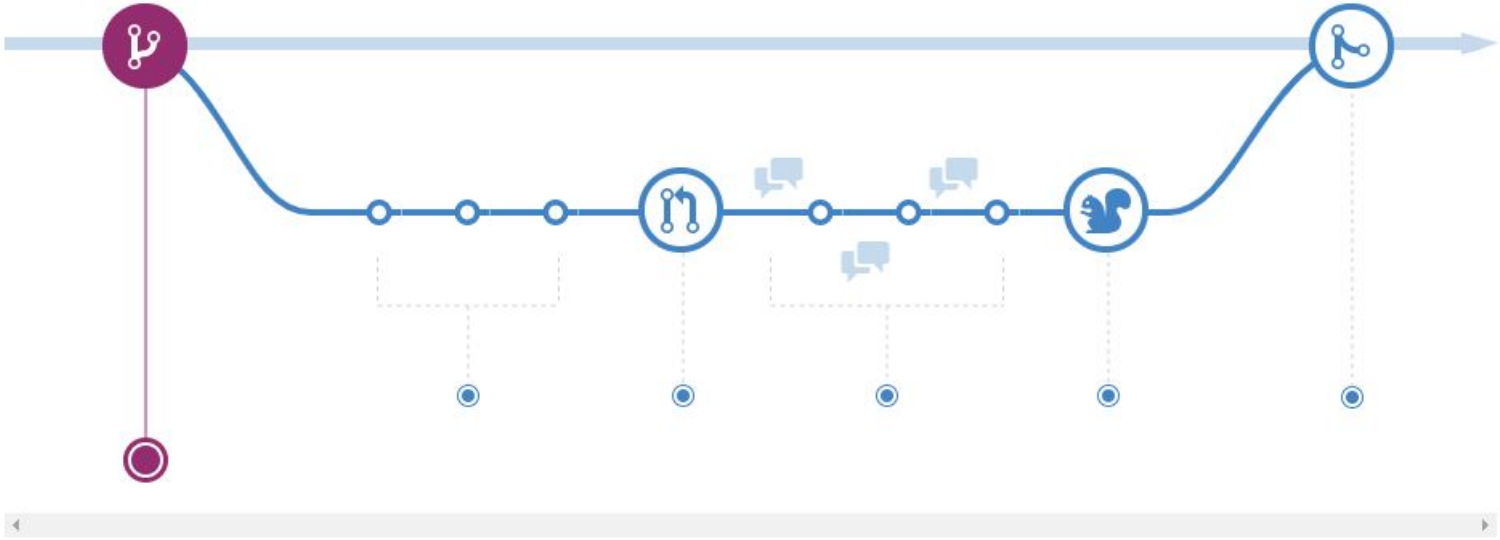
Exercise: resolve a conflict with main@upstream branch

Use your hellogitworld fork repository

Branching Workflows: **GitHub flow**

GitHub flow (step 1 of 6)

https://guides.github.com/introduction/flow/

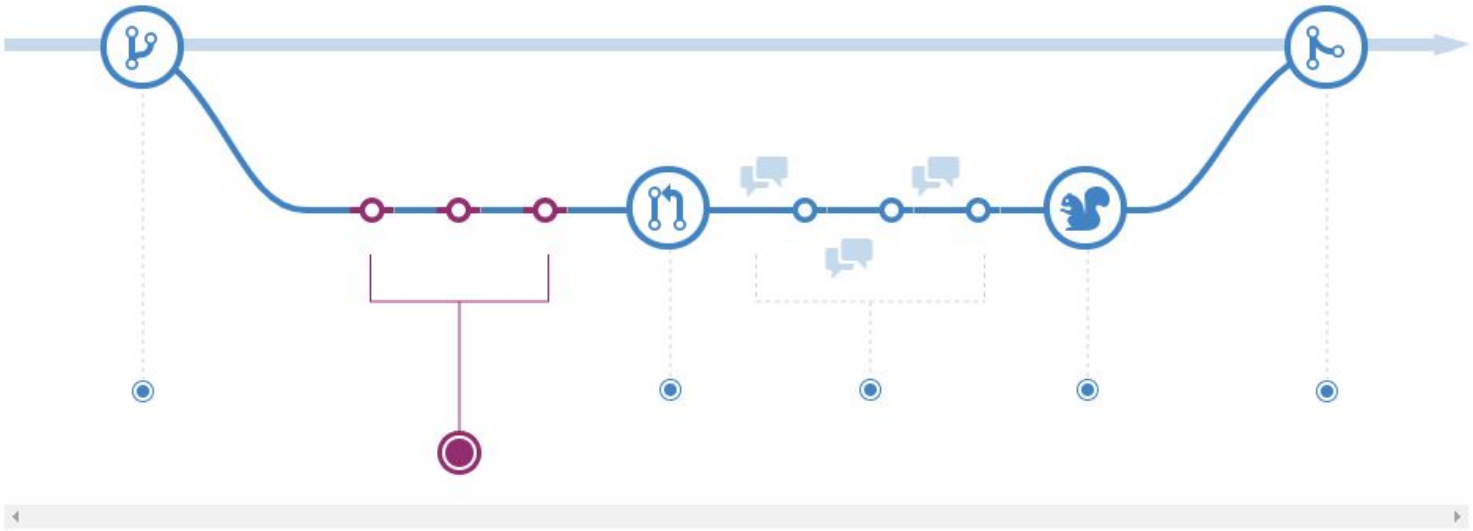


Create a branch

Branching Workflows: **GitHub flow**

GitHub flow (step 2 of 6)

https://guides.github.com/introduction/flow/

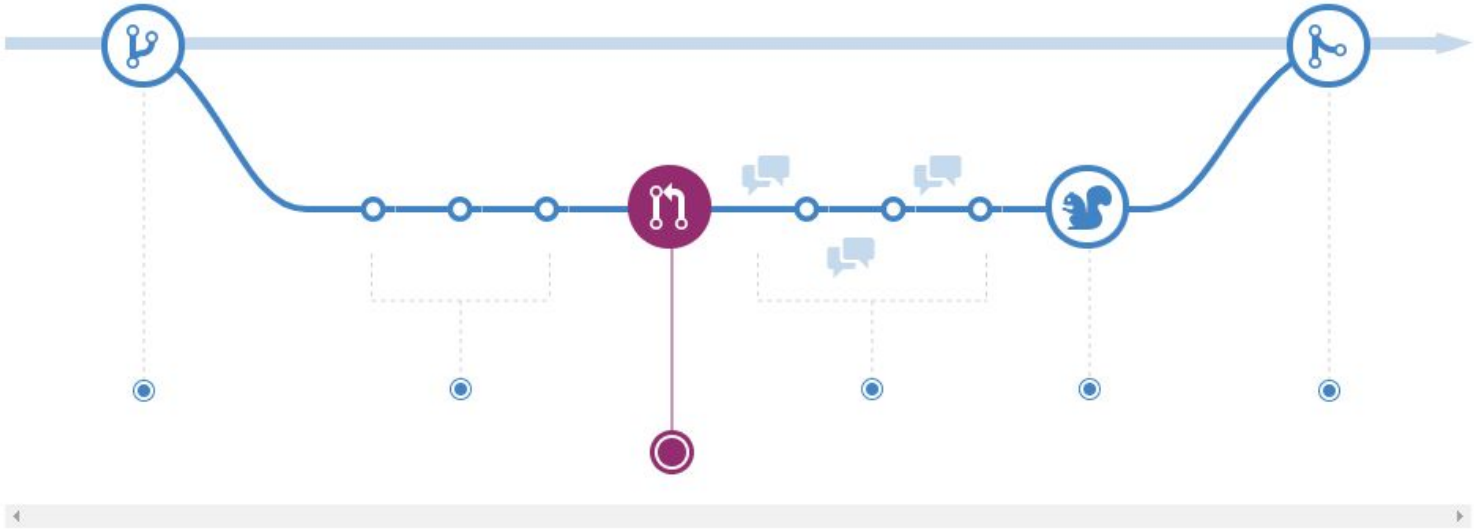


Add commits

Branching Workflows: **GitHub flow**

GitHub flow (step 3 of 6)

https://guides.github.com/introduction/flow/

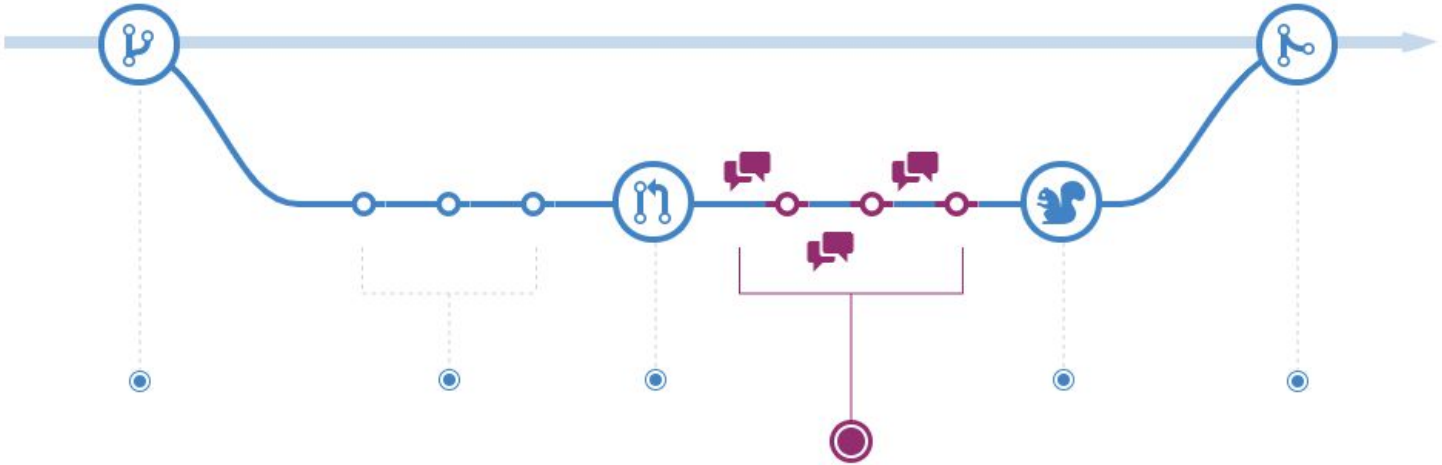


Open a Pull Request

Branching Workflows: **GitHub flow**

GitHub flow (step 4 of 6)

https://guides.github.com/introduction/flow/

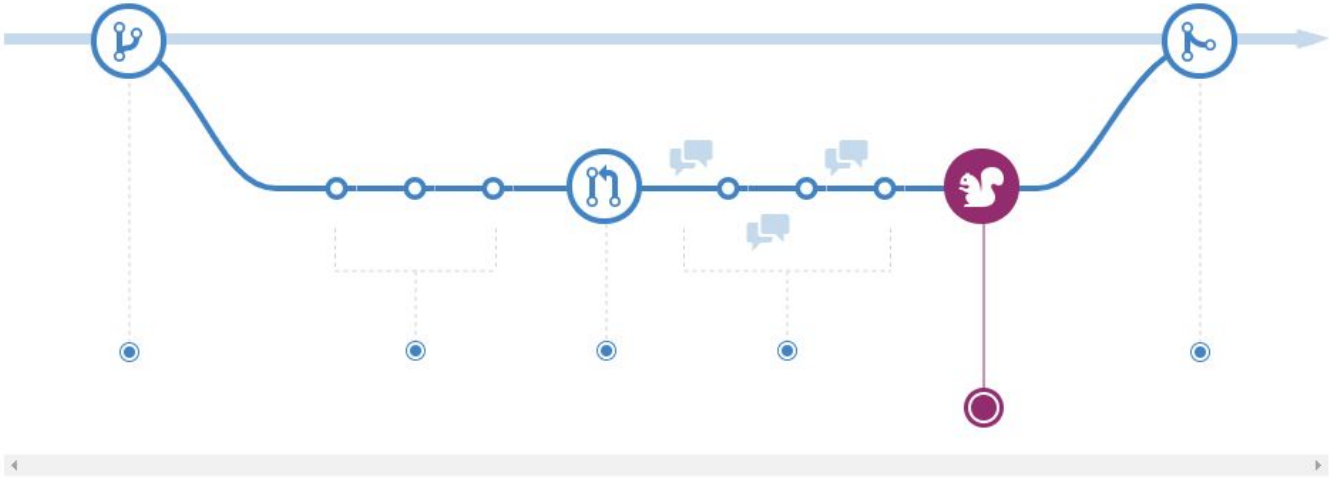


Discuss and review your code

Branching Workflows: **GitHub flow**

GitHub flow (step 5 of 6)

https://guides.github.com/introduction/flow/

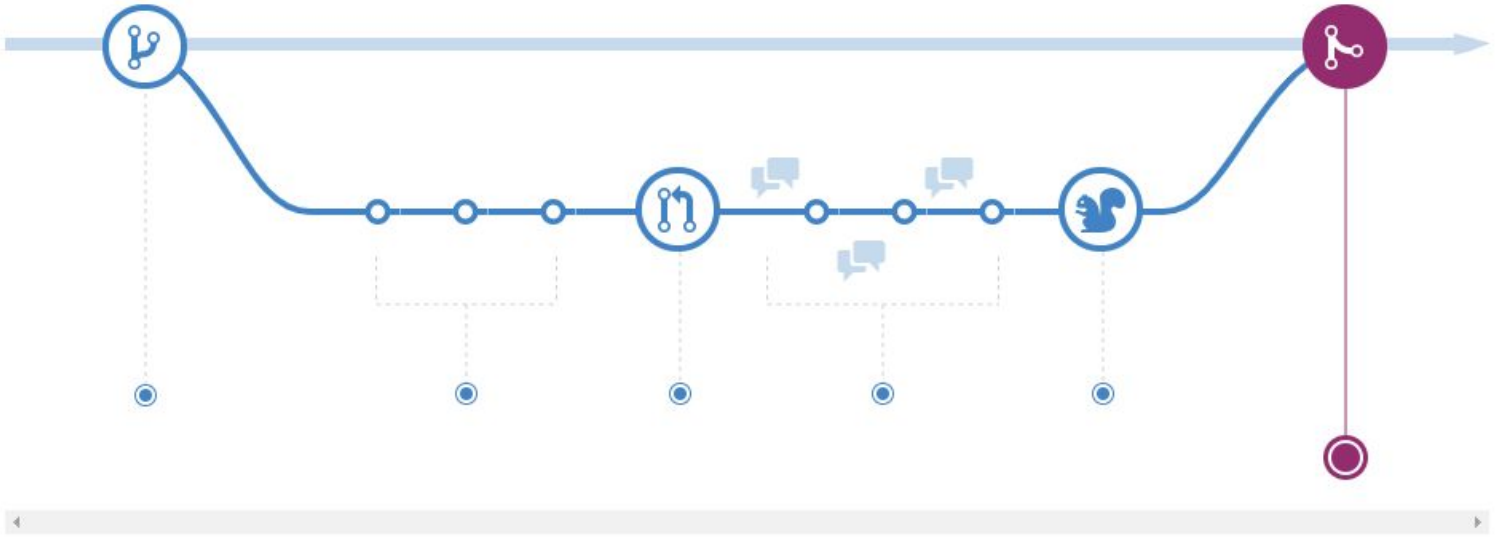


Deploy

Branching Workflows: **GitHub flow**

GitHub flow (step 6 of 6)

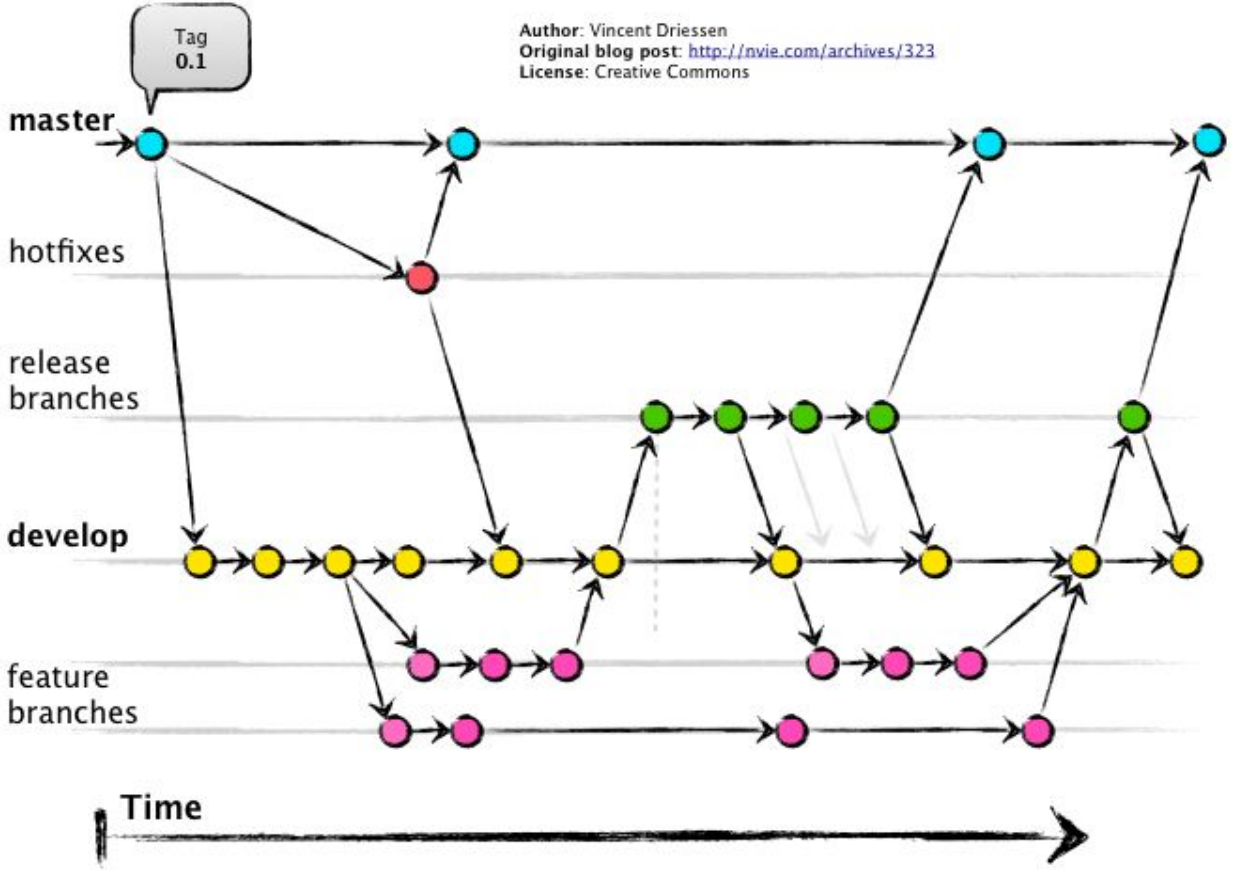
https://guides.github.com/introduction/flow/



Merge

Branching Workflows: **git-flow**

https://nvie.com/posts/a-successful-git-branching-model/



Hands-on

Undoing things

Basic Undoing
Undoing a commit message

- Example scenario:
  - You commit too early and forget to add some files or you want to modify the commit message

Basic Undoing
Undoing a commit message

- Example scenario:
  - You commit too early and forget to add some files or you want to modify the commit message
- Solution:
  - Use ***git commit --amend***
    - Right after having committed something it will allow you to change the commit message
    - Otherwise if you make some changes afterwards and stage them, they will take part of the last commit:

```
$ git commit -m “Initial commit”
$ echo “line added for the amendment” >> README.md
$ git add README.md
$ git commit --amend
```

Basic Undoing
Unstaging a Staged file

- Example scenario:
  - You added some file to the staged area that you later feel you don’t want it there

Basic Undoing
Unstaging a Staged file

- Example scenario:
  - You added some file to the staged area that you later feel you don’t want it there
- Solution:
  - Use ***git reset HEAD <file>***
    - The command is suggested by ***git status*** command after adding a file

```
$ echo “line added that was not for this commit” >> README.md
$ git add README.md
$ git status
On branch main
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:    README.md
# Let’s unstage README.md file
$ git reset HEAD README.md
# The commit won’t consider this change (BEWARE of using the ‘handy’ -a option when
committing)
$ git commit -m “Unstaging test”
```

Basic Undoing
Unmodifying a Modified file

- Example scenario:
  - You want to discard the changes done to a file

Basic Undoing

Unmodifying a Modified file

- Example scenario:
  - You want to discard the changes done to a file
- Solution:
  - Use ***git checkout -- <file>***
    - The command is (once again) suggested by ***git status*** command after modifying a file

```
$ echo “line added to test unmodifying” >> README.md
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   README.md
# Let’s unmodify README.md file
$ git checkout -- README.md
# Check with `git status` that the file is not longer listed in “Changes not staged for
commit”
$ git status
```

Basic Undoing

Reverting a file to a previous version

- Example scenario:
  - You want a file to get back to a previous version, N commits ago

Basic Undoing

Reverting a file to a previous version

- Example scenario:
  - You want a file to get back to a previous version, N commits ago
- Solution
  - Use ***git checkout <commit-ID> -- <file1> <file2>***

```
# Check your commit history
$ git log --patch --pretty=oneline
# Find the commit ID you want to revert to
$ git checkout 94d0778 -- README.md
# (or `nano README.md`) Check that the file has been reverted to the specific version
$ cat README.md
```

If all you remember is..

- Anything that is committed in Git can almost always be recovered
  - Commits overwritten with --amend, commits in deleted branches, etc.

If all you remember is..

- Anything that is committed in Git can almost always be recovered
  - Commits overwritten with --amend, commits in deleted branches, etc.
- Last commit (file/s included, message) can be modified with ***git commit --amend***

If all you remember is..

- Anything that is committed in Git can almost always be recovered
  - Commits overwritten with --amend, commits in deleted branches, etc.
- Last commit (file/s included, message) can be modified with ***git commit --amend***
- ***git checkout*** and ***git reset*** are powerful commands to revert things

Hands-on

Advanced topics on Git

Stashing..

git stash stores (ideally grouped) changes that can be popped later on

From Git help: “Stash the changes in a dirty working directory away”

- ***git stash list***
- ***git stash pop***
- ***git stash apply***
- ***git stash show***
  - ***git stash show stash@{0}***
    - **Shows the files** with modifications
  - ***git stash show stash@{0} -p***
    - **Shows the changes** from the files modified
- Partial stash
- Create a branch from a stash