# Git better

## Collaborative project management using *Git* and *GitHub*

Matteo Sostero

March 13, 2018

Sant'Anna School of Advanced Studies

# Let’s Git it done!

These slides are a brief primer to Git, and how it can help your workflow.

References:

- Atlassian tutorial
- GitHub guides
- Git cheatsheet
- Git book

Troubleshooting:

- How to undo (almost) anything with Git
- Git flight rules

# Table of contents

1. Version control with Git
2. Essential concepts
3. History and branching
4. Collaborative development
5. Coding with style
6. Managing a single-user project locally and remotely on GitHub
7. Collaborating on team projects on GitHub
8. Undo (almost) anything with Git

# Version control with Git

# Git

Git is a distributed version control system for tracking changes in files and coordinating work on those files among multiple people.

**Version control** (Git, Mercurial, CVS, Subversion, Bitkeeper):

- keep track of changes to text files line-by-line over time;
- easily track what changed between any two versions (text);
- revert any change if needed;
- back up and distribute copies of files;
- collaborate on projects.

**Distributed**:

- developers keep local copy of entire code and history;
- can make changes offline and asynchronously;
- changes (easily) reconciled later.

# Git pros

Advantages of Git:

- widely used, supported, documented;
- online platforms: GitHub, bitbucket, GitLab;
- dektop interfaces: shell, GitHub Desktop, SourceTree, GitKraken;
- integration in editors and IDEs: Emacs, Sublime Text, RStudio, XCode, Visual Studio, ...;
- distributed (asynchronous, offline) development;
- easy branching: eg, experimental branches for trying changes);
- easily make complex merges.

# Let’s Git on the same page

Challenges:

- steep learning curve;
- complex conceptual model;
- cryptic man pages (but good documentation!).
- trivial handling of binary files, images, Office documents;

Requires some workflow changes (see details):

- keep project under single directory (*outside* Dropbox, Google Drive, piCloud, ...!)
- consistency in personal and team coding style (eg, indentation, spacing, line-breaking);
- save and commit changes manually and frequently;
- requires to document code and explain changes.

# Essential concepts

# Git concepts: What

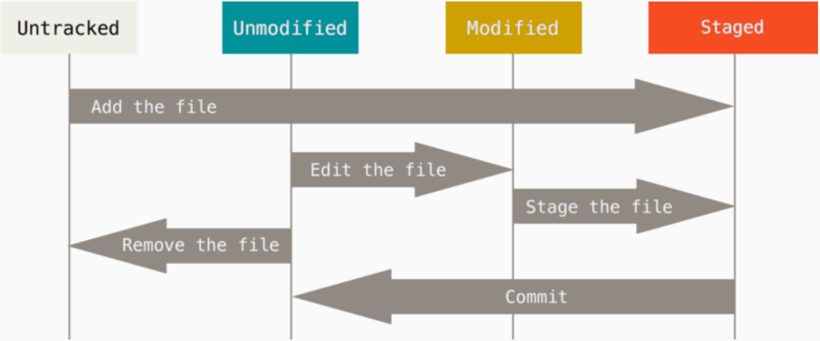
- **repository** (**repo**): a collection of files and their history:
  - where the code is kept: under a single root directory;
  - can be **local** (your machine), or distributed across your team or on a **remote** server (eg, GitHub).
- **commit**: a snapshot of your files at a given time:
  - how Git keeps track of changes in code over time and across team;
  - manually add one or more files to include to commit them and describe what changed in a message;
  - a permanent record of what changed (`diff`), when, by whom, with respect to what (**parent commit**);
  - a repository is "just a directed acyclic graph of commits".

# Git concepts: Where

Git has a sophisticated model of *where* things happen, based on how frequently you change things:

- **Working directory** (**working tree, workspace**): the files and sub-directories you can see and work on.
  *These are visible files stored on your disk.*
- **Staging area** (**index**): where you list files that will go into your next commit.
  *This is a file in the hidden* `.git/` *subdirectory on your disk.*
- **(local) Repository**: where the commits are stored; ie, it contains the full history of previous versions of the files in the repository, and relevant metadata.
  *Contained in the hidden* `.git/` *subdirectory on your disk.*
- **(remote) Repository**: a version of the repository hosted elsewhere.
  *On another computer, or online service like GitHub or Bitbucket.*

# Life cycle of files in a repository: from changes to staging area



# Key commands: repository actions

- `git status` (or interface) tells you the state of your repository:
  - which files are **untracked** and not indexed.
  - which files are **modified** with respect to the index/staging area;
  - which files are **staged** (**indexed**) and ready to be commited;
- `git add` new or modified files to the index.
- `git commit` commits changes in the index.

- `git diff` shows what has changed, wrt index or last commit.
- `git log` shows the history of commits up to this point.

- `git push`: send your local changes to the remote server;
- `git pull` (`git fetch`; `git merge`): get latest changes from remote.

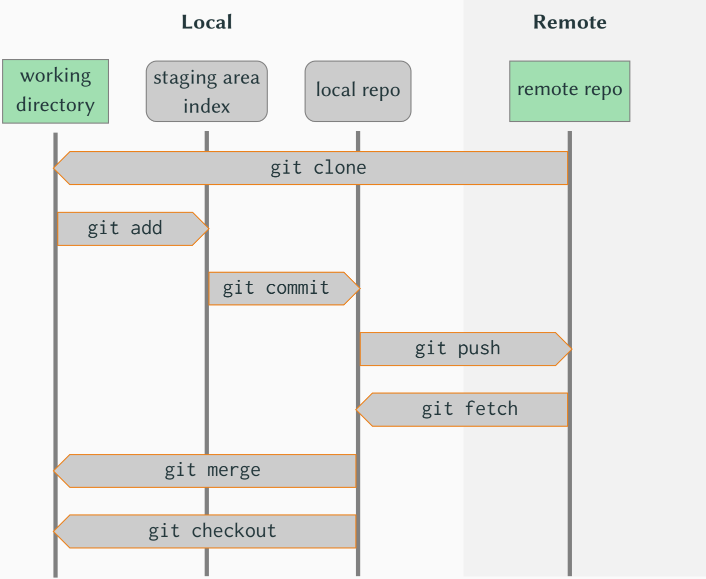
- `git clone`: copy a remote repository on your machine;
- `git init`: start a new repository in empty directory;

# Example workflow 1

## Solitary development, offline, from scratch

1. Create a new empty directory on your computer
2. `git init`: create a git repository in the directory
3. create, edit, and save some `file.txt` in the directory
4. `git status` shows the file is untracked
5. `git add file.txt` to stage the file
6. `git commit -m "First commit! added file.txt!"`
7. `git status` reports no changes
8. edit the file again, repeat from step (4).

# Working with repositories: from changes to commits



Adapted from Stackoverflow. See also ndpsoftware.com/git-cheatsheet.html

# History and branching

# History of the repo with `git log`

`git log` (or the interface) shows the history of commits of the repo:

### Log of a solitary development workflow

```
○ d764b48 HEAD -> Master - Polished model for release (1 hour ago) <Matteo>
│
○ 54ba4b2  - Fixed bug on credit constraint (4 hours ago) <Matteo>
│
○ c589395  - Improved consumer module (1 day ago) <Matteo>
│
○ b3bd158  - Added model files (2 days ago) <Matteo>
│
○ 63268c1  - Initial commit (4 days ago) <Matteo>
```

Note:

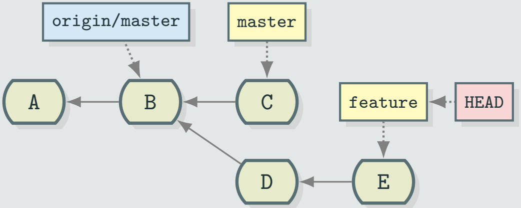
- Commits are uniquely identified by a SHA-1 hash (eg, a12b34...);
- Commit messages can have two parts:
  - short description ($\sim <$ 50 characters, above)
  - details after two line break (not shown);
- `HEAD` means “where your next commit would go” (pointer to branch);
- `Master` is the name of the main **branch**

# Branches

**Branches** are sequences of commits, which can trace an alternative history and versions of the code in yout repo.

- There is always a `Master` branch, containing the *main*, *baseline* version of the code.
- Git makes creating and merging branches quick and easy: it's a great help to the worflow.
- Other branches can be created from any commit to *experiment*, *implement new features*, *keep diverging versions*.

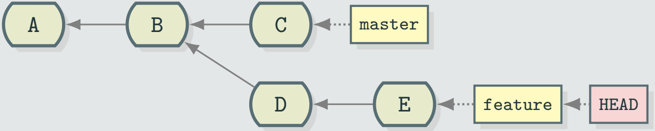
## Example history with branches



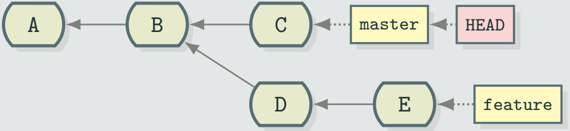
# Moving across branches with `git checkout`

`git checkout` moves `HEAD` on the tree (different branch or earlier commit on the same branch): it shows the files at different commit.

## Example switching branches



`git checkout master`

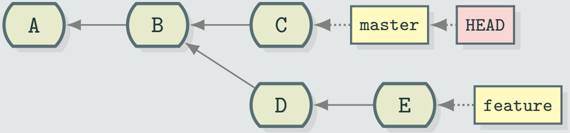


# Merging branches with `git merge`

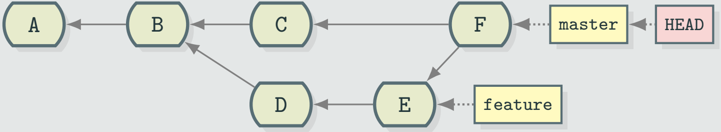
From the `Master` branch, `git merge feature master` creates a new commit combining the two branches.

## Merging branches

`git checkout master`



`git merge feature master`



# Collaborative development

# Collaborating on remote git repositories (GitHub, Bitbucket,...)

Online hosting services like GitHub and Bitbucket provide:

- **remote repository** to back up code;
- **issue tracking** interface: reporting bugs, suggesting improvements;
- **project and team management**: decide who can do what;
- managing contributions (**pull requests**) from team members and third parties;
- place to **host documentation** and **share know-how** on the code;
- venue to **make code public** and **invite collaboration** from anyone.

# Collaboration workflow on remote git repositories

The typical workflow within our GitHub *organization* or Bitbucket *team* would be:

1. create a (private/public) repository within the organization.
   - Invite collaborators among modelers;
   - Nominate administrators, maintainers, access rights, etc.
2. upload (or migrate) existing code on the repository;
3. other modelers clone repository on their machine, replicate and experiment;
4. document existing code; add a `README.md` explaining what the model does, reference to papers, how to collaborate;
5. document issues, suggest improvements;
6. create branches for developing new versions (features, bugfix,...)
7. merge branches in `Master` to integrate in the development version.
8. tag specific commits to refer to milestones of the project.

# Coding with style

# Git with style

Some matters of style and workflow to collaborate with ease.

- repository as self-contained, dedicated directory of (mostly) code:
  - include **all** relevant files in the repo (libraries, dependencies, ...);
  - in your code, refer to dependencies using relative paths: `./library/`, **not** `/home/user/project_name/library`;
  - exclude unnecessary files (eg, compilation artifacts, large data, useless binary files) using `.gitignore` (see github.com/github/gitignore).
  - keep it **outside** shared folders of Dropbox, Google Drive, piCloud!
- be **consistent** in personal and team coding style: indentation/tabs, spacing, character encoding (UTF-8), line-breaking.
- conform to a language **style guide** (eg, google.github.io/styleguide/);
- **commit early and often**;
- **never commit broken code**! use `git stash` to save it instead.
- commit related files together;
- write meaningful commit messages.
- tag release (baseline, published) versions of your code with `git tag`.

# Managing a single-user project locally and remotely on GitHub

# Existing project: local and remote repo

Use case: an existing local project in `/<project>/`, as yet untracked by git.

### Workflow

- Initialise a git repo:
  - `cd ~/<project>/`
  - `git init`
- Create `.gitignore` file, exclude irrelevant files.
- Stage all (relevant) files: `git add .`
- `git status` (is your friend)
- Commit changes

  `git commit -m "First commit (better late than never)"`

# Existing project: local and remote repo (continued)

## Workflow (continued)

- Create an **empty** remote repo called `<project>` in GitHub:
  - uncheck “Initialize this repository with a README”
  - no gitignore
  - no licence

  (You can always add those later; should not commit at this time)
- GitHub project now at `https://github.com/<user>/<project>`; name need not be globally unique (GUID: `<user>/<project>`), but make it **memorable**, **evocative** and **google-able**.
- Locally, add remote repo address called “origin”:
  ```
  git remote add origin https://github.com/<user>/<project>.git
  ```
- Push changes for the first time (creates remote `master` branch) :
  ```
  git push -u origin master
  ```

# Collaborating on team projects on GitHub

# Collaborating on a GitHub project

Use case: working with your team on `github.com/<user>/<project>`.

## Workflow

1. Add collaborators from `github.com/<user>/<project>/settings/collaboration`; they receive and accept invitations to work on the repo.
2. collaborators can access a private repository, clone repository locally and make changes;
3. collaborators can now push commits to repo, including `master`! Protect it from foolishness in `/settings/branches/` by requiring **branching** and **pull requests** in order to commit on `master`:
   - **collaborators**—branch → commit(s) → pull request;
   - **repo admin(s)**—evaluate pull requests.

# Undo (almost) anything with Git

# How to undo (almost) anything with Git

Git allows to undo most actions easily, depending on state of the repo.

### Golden rules

- *Don't Panic!* Your code is still there (somewhere).
- Working methodically will prevent (most) mishaps.
- Remote commits are there to stay—but you can revert their effects.

See also:

- GitHub's *How to undo (almost) anything with Git*;
- StackOverflow (pro tip: there is usually $>1$ way to do it).

# How to undo (almost) anything with Git: examples

Solutions to **local** (ie, not-yet-pushed) problems:

- Undo changes in `<file>` made since last commit: return files to last commit with `git checkout -- <file>`;
- Undo changes just committed (eg, typo in message or error in code): fix files, add them and commit again with `--amend` option;
- Stop tracking `<file>`: `git rm --cached <file>`; add it to `.gitignore`; commit;
- Undo changes in several recent commits: return to last good commit with `git log`, look for `<SHA>` of commit, then `git reset <SHA>`

Solutions to **public** (ie, pushed to remote) problems:

- Undo a public commit `<SHA>`: `git revert <SHA>` will create a new commit, that undoes (inverts) the changes in introduced by `<SHA>`; this new commit is added to existing (public) history.

# Contributing to third-party open-source projects on GitHub

# Contributing to third-party repositories on GitHub

GitHub hosts many great projects, mostly managed by volunteers.

Here's how you can help:

- **report issues (bugs)**:
  - **first, due diligence**: read documentation, search existing issues, repo Wiki, StackOverflow answers. Is your problem really new or general?
  - describe all the steps needed to reproduce; embed example code in Markdown (eg, R Minimal Reproducible Example);
  - give feedback on proposed solutions.
- **fix issues**: if you can solve an existing issue, fork the repo, work on it, and submit a pull request.
- **documentation**: writing good documentation takes time, and developers may be happy to delegate. Ask how you can help.
- **say thanks**: volunteer development is a thankless task; reach out to thank to the developers on social media.

# Further topics

# Sharing snippets of code easily with Github Gists

Github Gists (gist.github.com) are pages to store and share snippets of code. They are light-weight repositories, with an emphasis on showcasing the code itself, and making it easy to copy.

- great for sharing self-contained scripts or config files;
- automatically highlight code and can embed it in third-party websites;
- can be edited online by the author (automatic versioning);
- either public (listed on your profile page, searchable) or private (access with link only);
- can gather comments from everyone.

# References to advanced topics

Some useful topics covered in GitHub Guides (guides.github.com):

- Mastering Markdown: all about GitHub-flavoured Markdown, the easiest mark-up language around—used extensively in GitHub READMEs, issues, comments, and around the web;
- Documenting your projects on GitHub: writing great documentation (README, Wiki, pages) for your repo (in Markdown);
- Making Your Code Citable: assign a DOI (Digital Object Identifier) to cite your code in academic publication, using GitHub and Zenodo;
- Github Pages: easily build a website from a GitHub repo—for your project or for yourself!