Git better

Collaborative project management using *Git* and *Github*

Matteo Sostero

January 30, 2018

These slides: https://bit.ly/SA_git

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.

Find these slides at: https://bit.ly/SA_git

Let's Git it done!

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

Find these slides at: `https://bit.ly/SA_git`

References:

- Atlassian tutorial
- Github guides
- Git cheatsheet
- Git book

Troubleshooting:

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

Table of contents

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.

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.

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;

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:

Git concepts: What

- **repository** (**repo**): a collection of files and their history:
  - where the code is kept: under a single root directory;

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).

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:

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;

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;

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**);

Git concepts: What

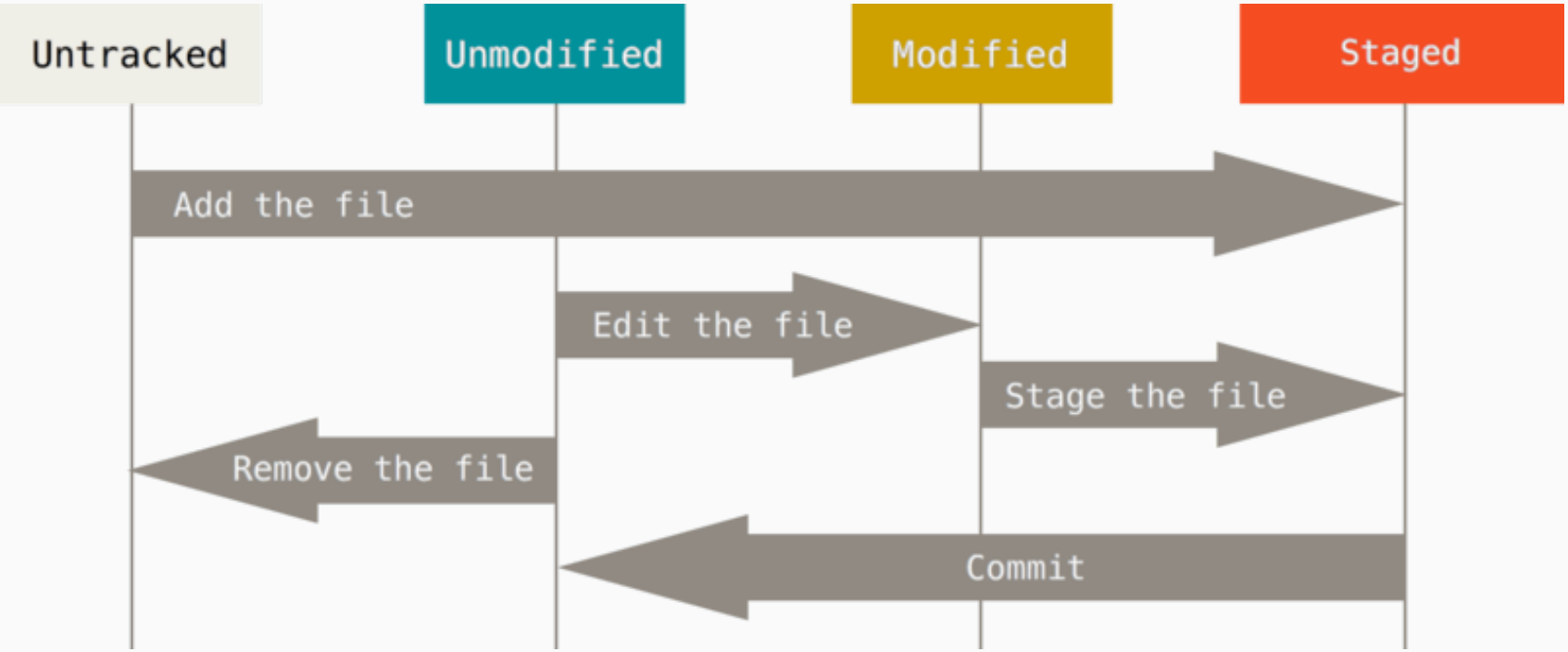
- **repository** (**repo**): a collection of files and their history:
  - where the code is kept: under a single root directory;
  - can be **local** (your machine), or distributed across your team or on a **remote** server (eg, Github).
- **commit**: a snapshot of your files at a given time:
  - how 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:

Key commands: repository actions

- `git status` (or interface) tells you the state of your repository:
  - which files are **untracked** and not indexed.

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;

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;

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.

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.

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.

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.

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;

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.

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;

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.

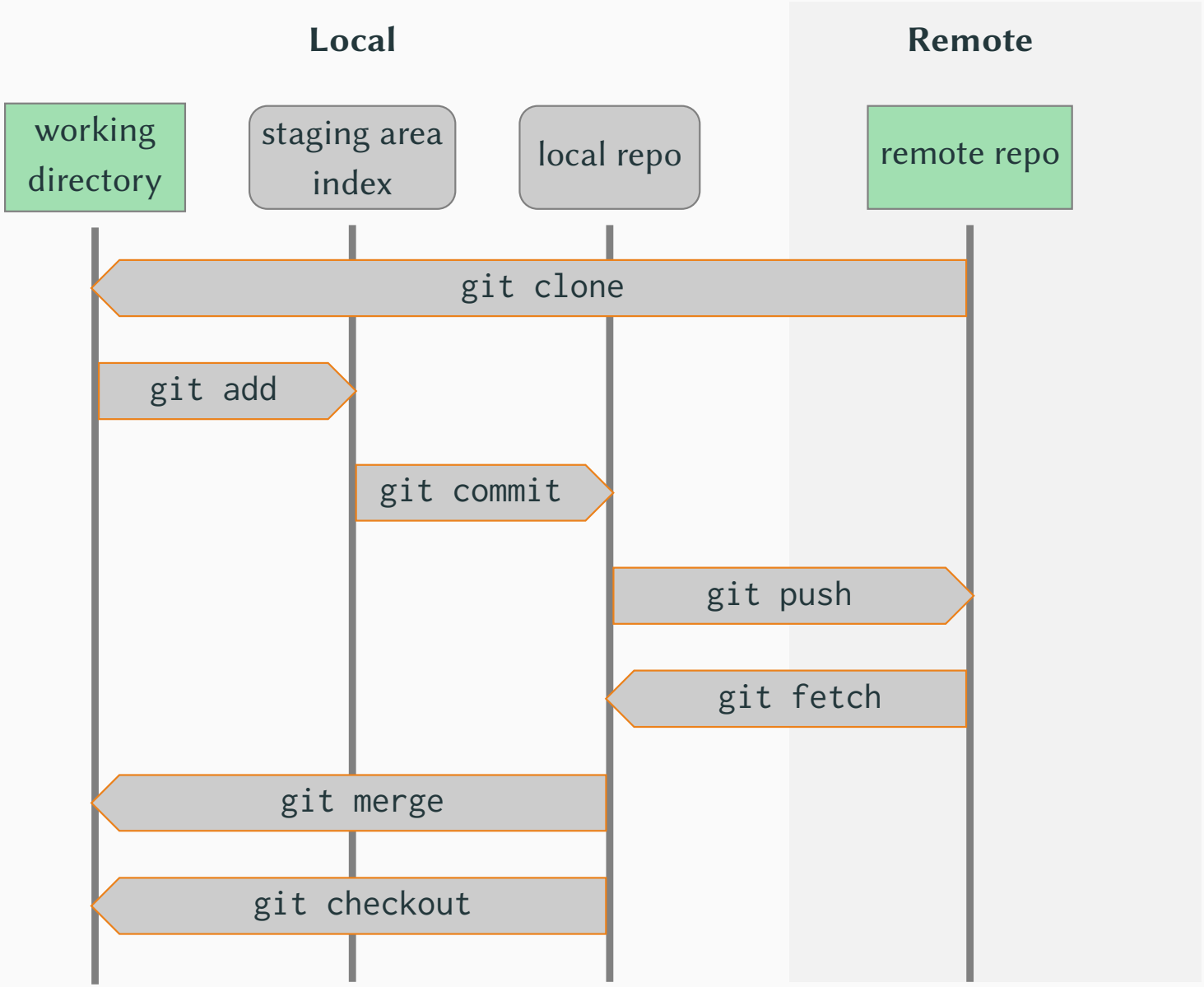
- `git clone`: copy a remote repository on your machine;
- `git init`: start a new repository in empty directory;

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:

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>
```

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.

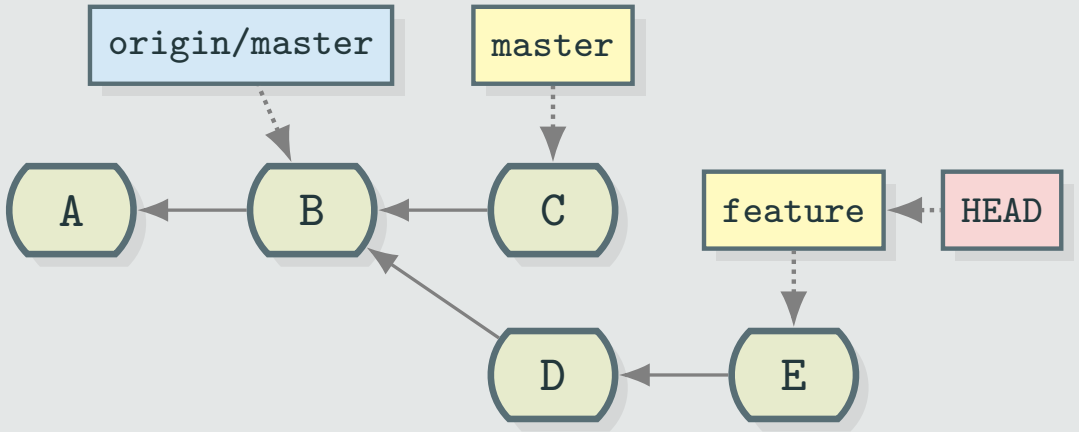
- There is always a `Master` branch, containing the *main*, *baseline* version of the code.
- Git makes creating and merging branches quick and easy: it's a great help to the worflow.
- Other branches can be created from any commit to *experiment*, *implement new features*, *keep diverging versions*.

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



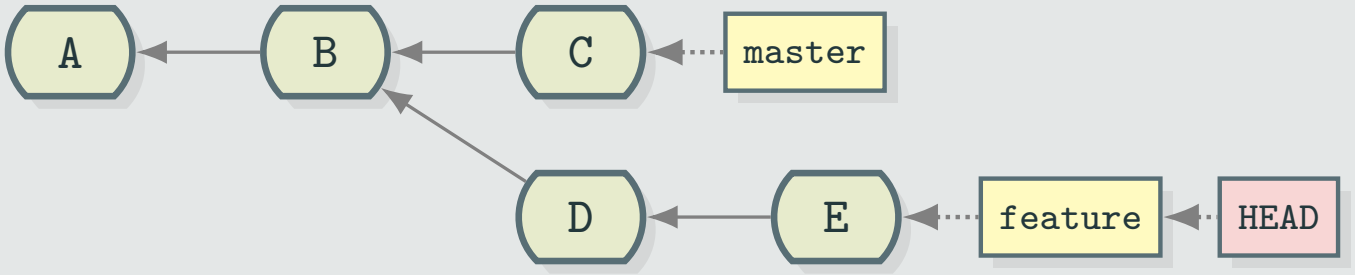
Moving across branches with `git checkout`

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

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.

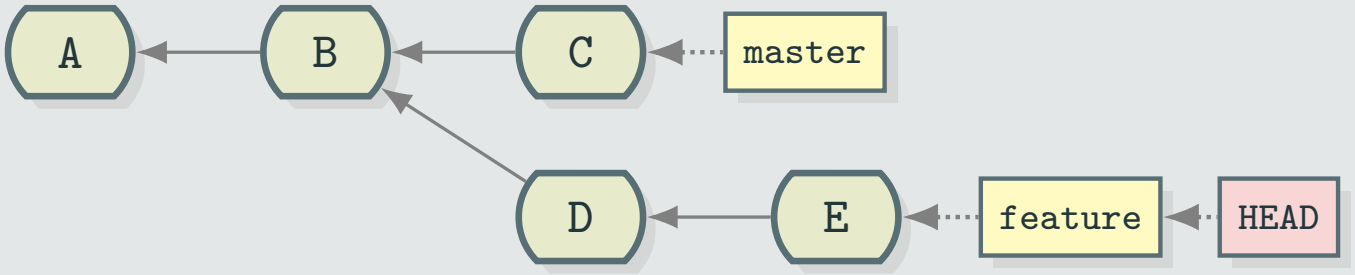
Example switching branches



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

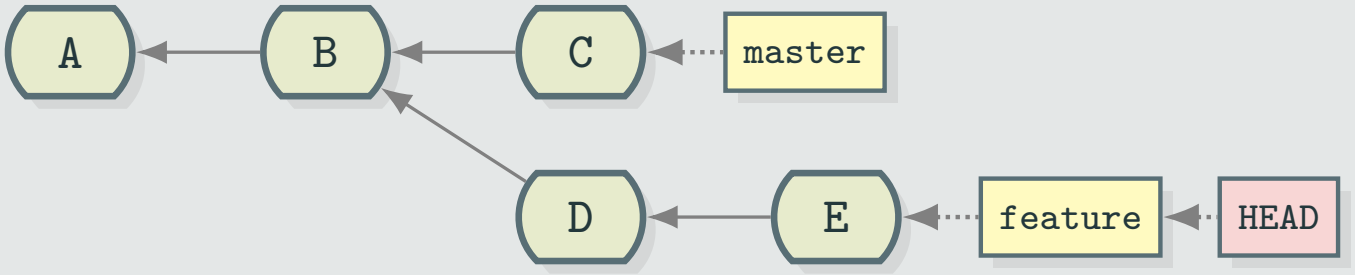



```
git checkout master
```

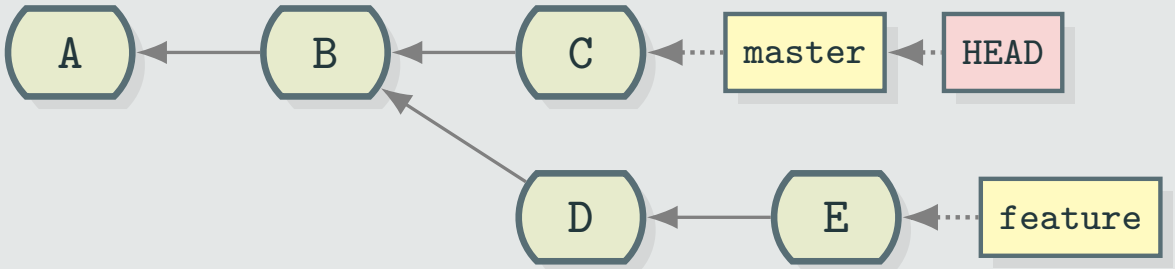
Moving across branches with `git checkout`

`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`

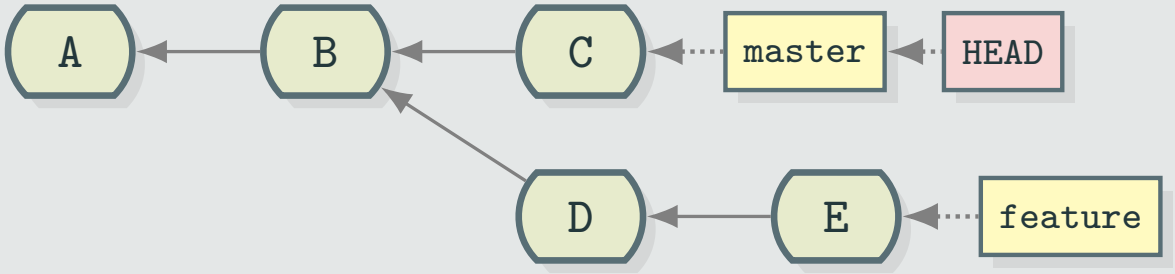


Merging branches with `git merge`

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

Merging branches

`git checkout master`

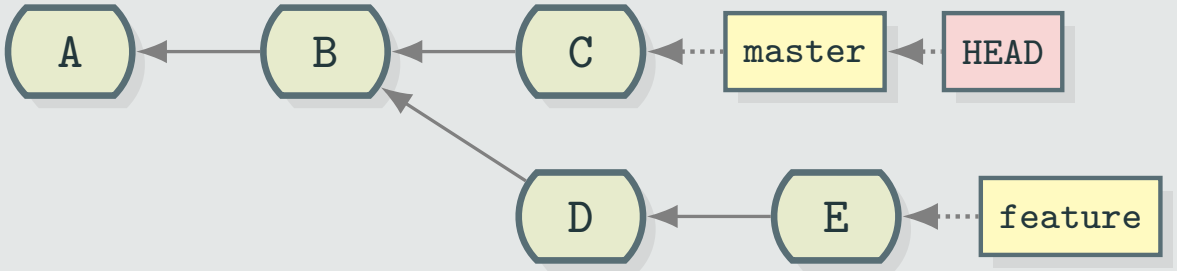


Merging branches with `git merge`

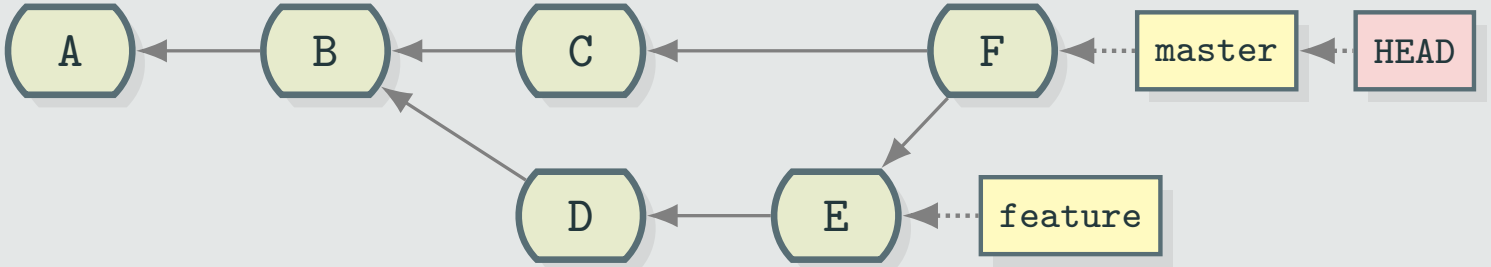
From the `Master` branch, `git merge feature master` creates a new commit combining the two branches.

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;

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;

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;

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;

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;

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.

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.

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.

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;

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.

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;

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;

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;

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;

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,...)

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.

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.

Style

Git with style

Some matters of style and workflow to collaborate with ease.

- repository as self-contained, dedicated directory of (mostly) code:

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, ...);

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`;

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).

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!

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.

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/);

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**;

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.

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;

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.

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`.

Pricing plans

Organization pricing plans

Git is free software, but extensive access to the cloud platform (GitHub, Bitbucket,...) is paying for non-open-source projects.

GitHub

cloud	*Education*	free for education, (max 10 repos)
cloud	*Team*	from $25/month (5 users) + $9/user/month
cloud	*Business*	$21 per user/month
self-hosted	*Enterprise*	$21 per user/month (by $\times$10 users, annual)

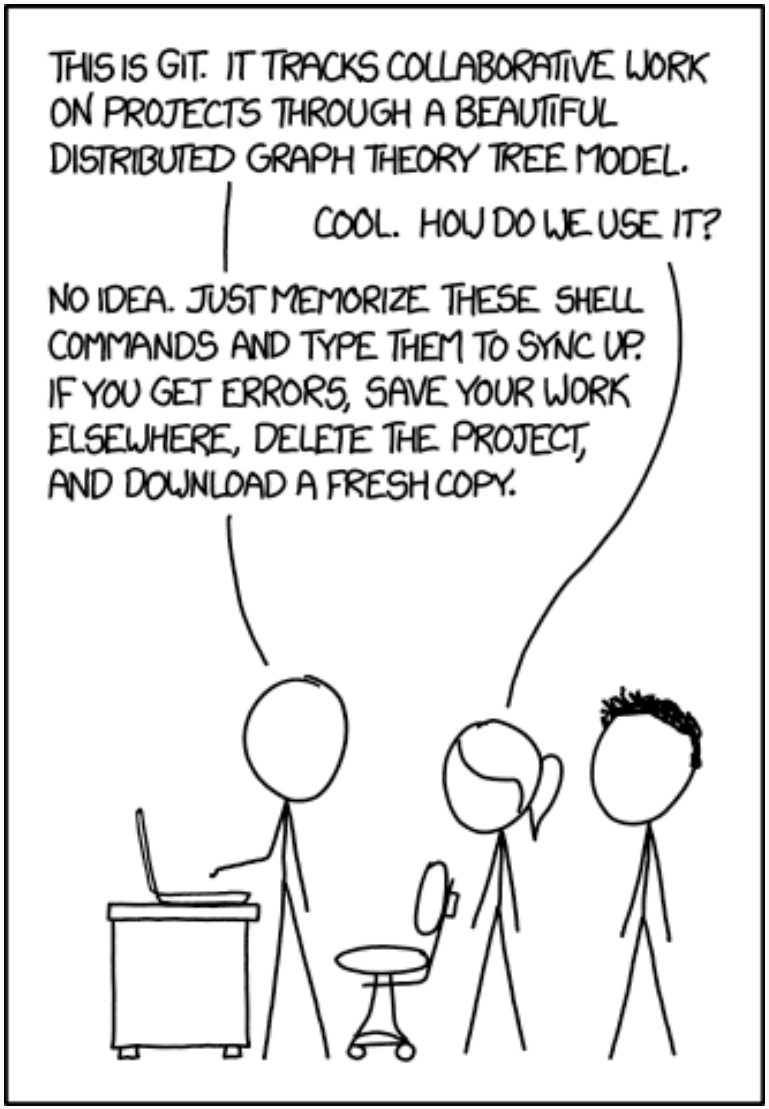
Bitbucket

cloud	*Standard*	$10/month + $2 per user/month
cloud	*Premium*	$25/month + $5 user/month
self-hosted		$1800/year (25 users)
self-hosted		$3300/year (50 users)

Thank you!

XKCD on git

back



If that doesn't fix it, git.txt contains the phone number of a friend of mine who understands git. Just wait through a few minutes of "It's really pretty simple, just think of branches as..." and eventually you'll learn the commands that will fix everything.