Git Basics

7 February, 2020
This is Git. It tracks collaborative work on projects through a beautiful distributed graph theory tree model.

Cool. How do we use it?

No idea. Just memorize these shell commands and type them to sync up. If you get errors, save your work elsewhere, delete the project, and download a fresh copy.

http://www.xkcd.com
Git Basics
<table>
<thead>
<tr>
<th>Why git?</th>
<th>Local Git</th>
<th>Git branches</th>
<th>Remote Git</th>
<th>Best practices</th>
</tr>
</thead>
</table>

![Diagram](image)

**Git Basics**
Why git?

Local Git

Git branches

Remote Git

Best practices

Git Basics
Why git?

Local Git

Git branches

Remote Git

Best practices

Git Basics
Why git?

Local Git

Git branches

Remote Git

Best practices

Git Basics
Why git?

Local Git

Git branches

Remote Git

Best practices

Git Basics
Why git?

Local Git

Git branches

Remote Git

Best practices

Git Basics
Why git?

Local Git

Git branches

Remote Git

Best practices

Git Basics
Git Basics
<table>
<thead>
<tr>
<th>Why git?</th>
<th>Local Git</th>
<th>Git branches</th>
<th>Remote Git</th>
<th>Best practices</th>
</tr>
</thead>
</table>

![Diagram of Git flow](image)

**Git Basics**
<table>
<thead>
<tr>
<th>Why git?</th>
<th>Local Git</th>
<th>Git branches</th>
<th>Remote Git</th>
<th>Best practices</th>
</tr>
</thead>
</table>

**Git helps you to:**

- continuously keep backups of your work;
- restore the entire state of your project to a previous version;
- undo specific changes to some files – or even part of files – without affecting others;
- easily share your progress with other people;
- synchronise your work across multiple devices;
- experiment with a new feature without affecting existing work;
- maintain multiple versions of a product;
- and more!
Git helps you to:

- continuously keep backups of your work;
Git helps you to:

- continuously keep backups of your work;
- restore the entire state of your project to a previous version;
Git helps you to:

• continuously keep backups of your work;
• restore the entire state of your project to a previous version;
• undo specific changes to some files – or even part of files – without affecting others;
Git helps you to:

- continuously keep backups of your work;
- restore the entire state of your project to a previous version;
- undo specific changes to some files – or even part of files – without affecting others;
- easily share your progress with other people;
Git helps you to:

- continuously keep backups of your work;
- restore the entire state of your project to a previous version;
- undo specific changes to some files – or even part of files – without affecting others;
- easily share your progress with other people;
- synchronise your work across multiple devices;
Git helps you to:

- continuously keep backups of your work;
- restore the entire state of your project to a previous version;
- undo specific changes to some files – or even part of files – without affecting others;
- easily share your progress with other people;
- synchronise your work across multiple devices;
- experiment with a new feature without affecting existing work;
Git helps you to:

- continuously keep backups of your work;
- restore the entire state of your project to a previous version;
- undo specific changes to some files – or even part of files – without affecting others;
- easily share your progress with other people;
- synchronise your work across multiple devices;
- experiment with a new feature without affecting existing work;
- maintain multiple versions of a product;
Git helps you to:

- continuously keep backups of your work;
- restore the entire state of your project to a previous version;
- undo specific changes to some files – or even part of files – without affecting others;
- easily share your progress with other people;
- synchronise your work across multiple devices;
- experiment with a new feature without affecting existing work;
- maintain multiple versions of a product;
- and more!
<table>
<thead>
<tr>
<th>Why git?</th>
<th>Local Git</th>
<th>Git branches</th>
<th>Remote Git</th>
<th>Best practices</th>
</tr>
</thead>
</table>

**Actions:**

![Actions](image-url)
<table>
<thead>
<tr>
<th>Why git?</th>
<th>Local Git</th>
<th>Git branches</th>
<th>Remote Git</th>
<th>Best practices</th>
</tr>
</thead>
</table>

**Actions:** figure out which files should be on version control
Actions: `git add *.txt *.java images/ ; git commit`
Actions: replace all occurrences of “magenta” by “pink”
### Git Basics

<table>
<thead>
<tr>
<th>Why git?</th>
<th>Local Git</th>
<th>Git branches</th>
<th>Remote Git</th>
<th>Best practices</th>
</tr>
</thead>
</table>

**Actions:**

```
git commit -a
```
Actions: add a new source file and change some others
Actions: `git add newfile.java ; git commit`
Actions: start working on a new feature
Actions: realise it was a bad idea
**Actions:** `git reset --hard`
## Why git?

| Local Git | Git branches | Remote Git | Best practices |

### Git Basics

**Actions:**

```
Actions:
```

- master

```
```
Actions: update documentation
Actions: `git commit -a`
Actions: decide you want to see an earlier version
Actions: git checkout de337dc
Actions: `git checkout master`
Actions: realise that the colour change was a bad idea
Actions: `git revert de337dc`
Actions: realise that you were drunk during the last two commits
Actions: `git reset 252137e --hard` (be very careful!)
Why git? Local Git Git branches Remote Git Best practices

Actions:

Git Basics
You can also:

- view a graphic description of your commits (like given here)
- “stage” changes gradually
- view files in earlier versions (git show a62c16e:file1.txt)
- recover only a single file (git checkout a62c163 -- file1.txt)
- view differences between a current and prior version of a file
- stage changes gradually
Actions:
Actions: **start work on the network**
Actions: `git commit -a`
Actions: decide to work on a high-priority database change
Why git?  
Local Git  
Git branches  
Remote Git  
Best practices

master

foo

Actions: `git checkout -b foo de337dc`
Actions: make some changes to the database
Actions: `git commit -a`
Actions: complete changes to the database
Actions: `git commit -a`
Actions: `git checkout master`
Actions: **improve network negotiation**
Actions: `git commit -a`
Actions: fix last of the network bugs
Actions: `git commit -a`
Actions: `git merge foo`
Why git?

Local Git

Git branches

Remote Git

Best practices

master

foo

Actions:
Git branches should be seen as a directed acyclic graph!

- Every vertex has a unique label.
- Some vertices are also labeled with a branch name.
- Exclusively one vertex is active, this is the HEAD.
- Commands like show, diff, and checkout work with either.
- Commit and merge add a vertex and advance the branch.
Git branches should be seen as a directed acyclic graph!

- every vertex has a unique label
Git branches should be seen as a directed acyclic graph!

- every vertex has a unique label
- some vertices are also labeled with a branch name
Git branches should be seen as a directed acyclic graph!

- every vertex has a unique label
- some vertices are also labeled with a branch name
- exactly one vertex is active, this is the HEAD
Git branches should be seen as a directed acyclic graph!

- every vertex has a unique label
- some vertices are also labeled with a branch name
- exactly one vertex is active, this is the HEAD
- commands like `show`, `diff`, `checkout` work with either
Git branches should be seen as a directed acyclic graph!

- every vertex has a unique label
- some vertices are also labeled with a branch name
- exactly one vertex is active, this is the HEAD
- commands like `show`, `diff`, `checkout` work with either
- commit and merge add a vertex and advance the branch
Git branches should be seen as a directed acyclic graph!

- every vertex has a unique label
- some vertices are also labeled with a branch name
- exactly one vertex is active, this is the HEAD
- commands like `show`, `diff`, `checkout` work with either
- `commit` and `merge` add a vertex and advance the branch
Git branches should be seen as a directed acyclic graph!

- every vertex has a unique label
- some vertices are also labeled with a branch name
- exactly one vertex is active, this is the HEAD
- commands like `show`, `diff`, `checkout` work with either
- commit and `merge` add a vertex and advance the branch

Git Basics
Why git? Local Git Git branches Remote Git Best practices

Git rebase

Git Basics
Why git?

Local Git

Git branches

Remote Git

Best practices

Git rebase

Git Basics
Actions:
Actions: `git pull`
Actions: Martha updates the database handlers.
Actions: `git commit -a`
Actions: `git push`
Actions: git pull
Actions: Martha decides that the colour change was a bad idea.
Actions: `git revert de337dc`
Actions: Harry changes the visualisation module.
Actions: git push
Actions: `git commit -a`
Actions: git pull
Actions: Harry handles the merge conflicts (if necessary).
Why git? Local Git Git branches Remote Git Best practices

Actions: git push

Git Basics
Why git?

Local Git

Git branches

Remote Git

Best practices

Actions:

Git Basics
Actions:  Harry decides to start working on a new feature.
Why git?

Local Git

Git branches

Remote Git

Best practices

---

**Actions:**

```bash
git branch DB
```
Actions: Harry changes the table structure
Actions: git commit -a
Actions: Collaborators make their own changes to the project.
Actions: **Harry moves database interactions to a separate class**
Actions: `git add DBManager.java ; git commit -a`
Actions: Collaborators make more changes and ask Harry to look.
Why git? Local Git Git branches Remote Git Best practices

Actions: git checkout master
Actions: `git pull`
Actions: Harry makes a minor update.
Actions: `git commit -a`
Actions: git push
Actions: git checkout DB
Actions: Harry completes changes to the database.
Why git?

Local Git

Git branches

Remote Git

Best practices

Actions: `git commit -a`

Git Basics
Why git?  
Local Git  
Git branches  
Remote Git  
Best practices

Actions: git checkout master
Actions: git merge DB
Actions: git push
You can also...

- push branches onto the server (`git push -u origin <branchname>`)  
- view which branches you have, which is active (HEAD), etc.  
- keep a branch up-to-date with another without merging (`git rebase master`)  
- delete whole branches (`git branch -d <branchname>`)  
- change branch names; make another branch master (but be careful!)
## Bad workflow

<table>
<thead>
<tr>
<th>Commit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial commit</td>
<td></td>
</tr>
<tr>
<td>First release</td>
<td></td>
</tr>
<tr>
<td>New function with bug</td>
<td></td>
</tr>
<tr>
<td>Bug fixed</td>
<td></td>
</tr>
<tr>
<td>Minor improvement</td>
<td></td>
</tr>
<tr>
<td>Second release</td>
<td></td>
</tr>
</tbody>
</table>
Bad workflow

Initial commit
Bad workflow

Initial commit

First release
Bad workflow

Initial commit

First release

New function with bug
Bad workflow

Initial commit

First release

New function with bug

Bug fixed
Bad workflow

Initial commit
First release
New function with bug
Bug fixed
Minor improvement
Bad workflow

Initial commit

First release

New function with bug

Bug fixed

Minor improvement

Second release
Good workflow

- master
- develop
- feature
Good workflow

Initial commit

Workflow
Good workflow

- Initial commit
- Working version
Good workflow

Initial commit

Working version

First release
Good workflow

Initial commit

Working version

First release

New function with bug
Good workflow

Initial commit
Working version
First release
New function with bug
Bug fixed
**Good workflow**

- Initial commit
- Working version
- First release
- New function with bug
- Bug fixed
- Update working version
Good workflow

- Initial commit
- Working version
- First release
- New function with bug
- Bug fixed
- Update working version
- Minor improvement
Good workflow

- Initial commit
- Working version
- First release
- New function with bug
- Bug fixed
- Update working version
- Minor improvement
- Second release
Try it out!!

• make a local git repository (possibly: one of those you played with before)
• create an empty repository on github
• push your repository to the server
• make branches and push those to the server, too
• delete branches on the server
• (fork and) clone someone else’s repository
• generate and resolve merge conflicts
• any remaining questions... try it yourself :)
General advice

- do not mess with remote history!
- use a development branch next to a release branch
- use a separate branch feature/pick-the-name for each feature
- all files in .gitignore will be ignored (e.g., *.sw? for swap files and *~ for editor backup files)
- keep an eye on the repository structure through git gui (or an alias)