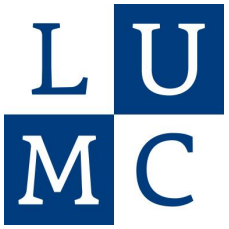

Git Basics

Wibowo Arindrarto

Git Course
6 September 2017

Starting a Project

Creating a repository is easy. You can either:

1. start from scratch on your own

```
$ cd ~/projects
$ git init
Initialized empty Git repository in {current-directory}/.git/
```

Starting a Project

Creating a repository is easy. You can either:

1. start from scratch on your own, or

```
$ cd ~/projects
$ git init
Initialized empty Git repository in {current-directory}/.git/
```

2. clone an existing remote (or local) repository

```
$ cd ~/projects
$ git clone {url-or-path-to-existing-repo}
Cloning into ‘{repo-name}’...
...
```

Starting a Project

You can see that git creates a hidden directory inside your working directory.

```
$ ls -a
. .. .git
```

This is where git stores the files necessary to track your progress. You rarely need to edit the contents of this directory.

Removing this directory means removing your repository!

Git Operations

Operations can be local or remote. Most importantly, they track files around these three areas:

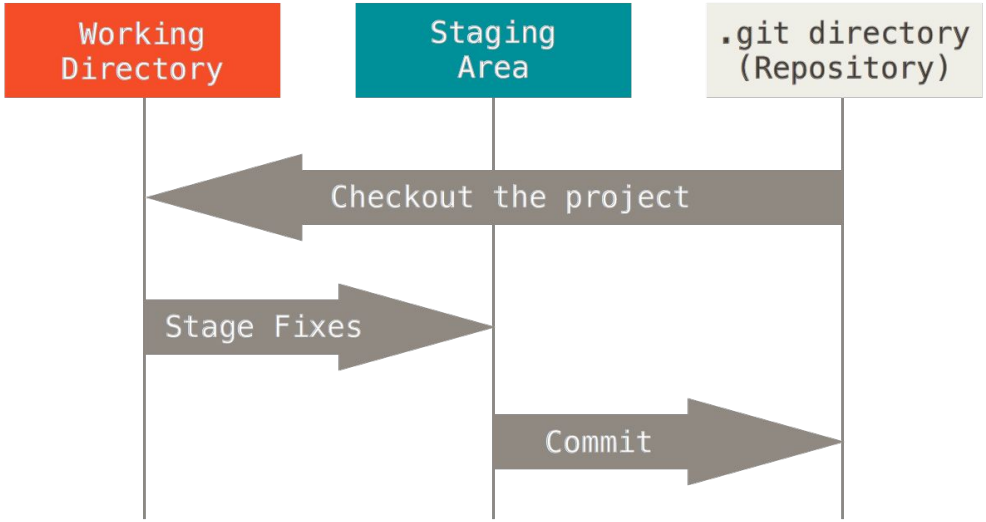

Image taken from the Pro Git Book

1. Working Directory - where you modify your files
2. Staging Area - where parts of files are marked for saving
3. Git Repository - where history is saved (mostly invisible to you)

Git Operations

Git itself will associate your files with various states:

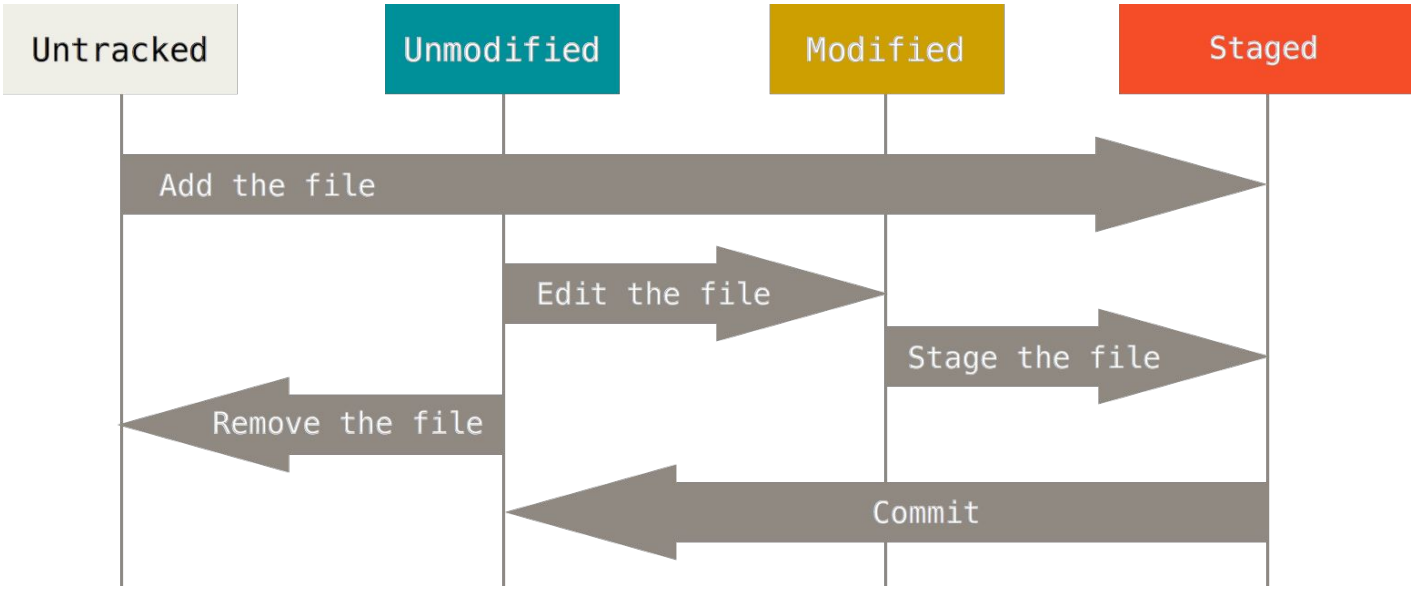

Image taken from the Pro Git Book

Git Operations

Git itself will associate your files with various states:

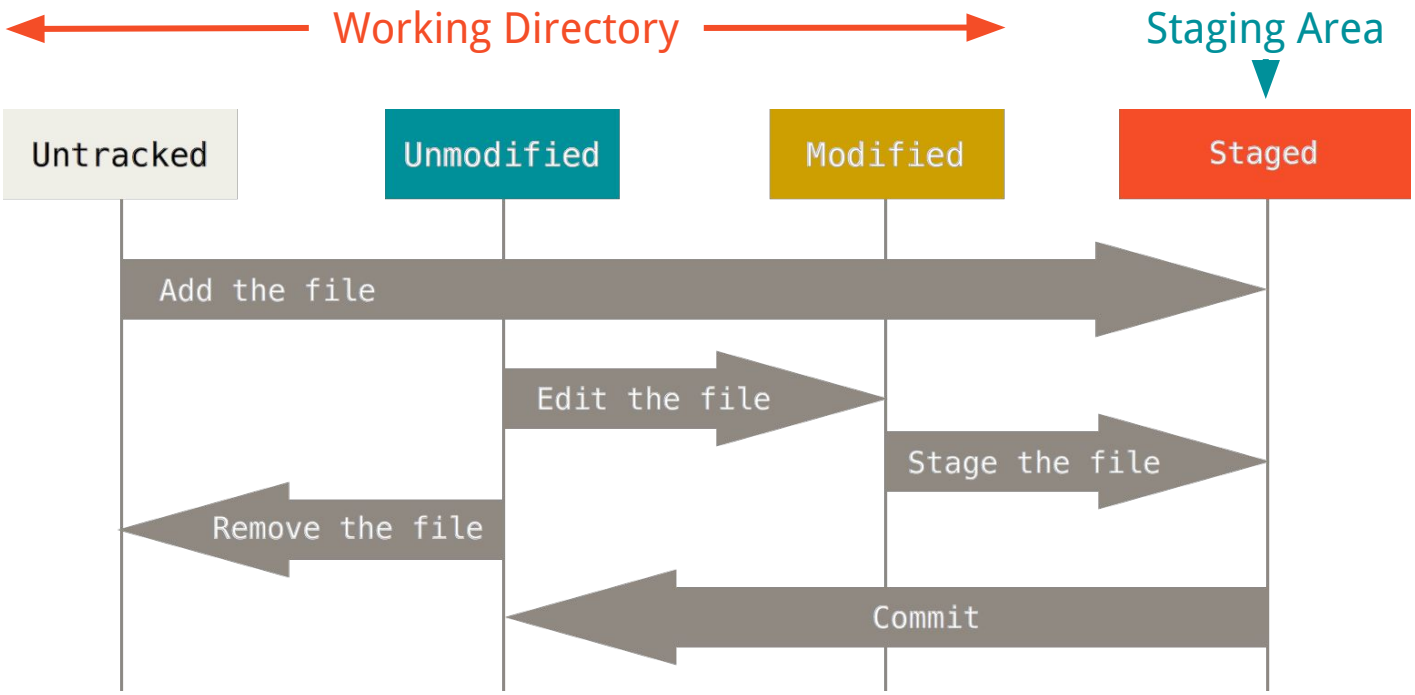

Image taken from the Pro Git Book

Prelude: Quick Setup

If you have never used git in your computer before, you need to tell it who you are. This information is used to mark each save point / commit.

```
$ git config --global user.name "{your-name-or-nick-name}"
$ git config --global user.email "{your-email}"
```

Let's also tell git that we want its output to be colored:

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

This configurations are all saved in the `~/.gitconfig` file. You can also change the file contents to change these values.

Git Operations

In a freshly git-initialised directory, create a `README` file:

```
$ echo "First version." > README
```

You can then check the state of the directory in git using the `git status` command:

```
$ git status
On branch master

Initial commit

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)
```

Git Operations

To start tracking the file, we need to stage it first. Use the `git add` command to do so:

```
$ git add README
```

We can see that the state of the file has changed:

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   README
```

Note also that the file is now in the staging area.

Git Operations

Finally, we can commit the file into the repository using the `git commit` that specifies a commit message:

```
$ git commit -m "First commit"
[master (root-commit) 5466170] First commit
 1 file changed, 1 insertion(+)
 create mode 100644 README
```

If the commit message is long, you can omit the `-m` flag and git will open a text editor in which you can write your longer message.

We can see that our working directory is clean of any changes. Also, the `README` file is now tracked by git.

```
$ git status
On branch master
nothing to commit, working directory clean
```

Git Operations

Let's update the file now:

```
$ echo "Second version." > README
```

Git knows that there are changes to the file:

```
$ git status
On branch master
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

no changes added to commit (use "git add" and/or "git commit -a")
```

Git Operations

It also knows where and what the changes are. You can see this using the `git diff` command:

```
$ git diff
diff --git i/README w/README
index efe6f7c..4fe6328 100644
--- i/README
+++ w/README
@@ -1 +1 @@
-First version.
+Second version.
```

This compares the working directory with the staging area. To compare the staging area with the repository, we use the `--cached` flag:

```
$ git diff --cached
```

Git Operations

Let's add and commit the file again. You can do `git add` followed by `git commit` as we did before:

```
$ git add README
$ git commit -m "Second commit"
```

Since we have tracked the file, you can also abbreviate this into one `git commit` command:

```
$ git commit -am "Second commit"
[master ef70f09] Second commit
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Notice the `-a` flag. This tells git to automatically stage all previously-tracked files.

Git Operations

We can see our history of commits using the `git log` command:

```
$ git log
commit 7a6e47cfbb38048b46937d9f8d2427a7e6e20936
Author: Wibowo Arindrarto <w.arindrarto@lumc.nl>
Date:   Tue Nov 24 16:13:59 2015 +0100

    Second commit

commit 54661709e859427358c97a94475643a7ccffa052
Author: Wibowo Arindrarto <w.arindrarto@lumc.nl>
Date:   Tue Nov 24 15:04:54 2015 +0100

    First commit
```

Each commit is identified by a unique hash ID. It is used in various command, for example: `git diff {ID1} {ID2}` compares two different commits.

Restoring Previous Versions

Having tracked our files, git then allows us to retrieve their previous versions back.

There are several variations of this, depending on which state your files are in.

Restoring Previous Versions

One that is used quite commonly is to discard working directory changes and restore the latest repository state:

```
$ git checkout -- {filename}
```

A variation of this command discards all changes in the working directory:

```
$ git checkout -- .
```

Another variation retrieves the file from a specific commit into the staging area:

```
$ git checkout {commit-id} {filename}
```

Restoring Previous Versions

One that is used quite commonly is to discard working directory changes and restore the latest repository state:

```
$ git checkout -- {filename}
```

A variation of this command discards all changes in the working directory:

```
$ git checkout -- .
```

Another variation retrieves the file from a specific commit into the staging area:

```
$ git checkout {commit-id} {filename}
```

To move the file out of the staging area back to the working directory you can use the `git reset` command:

```
$ git reset HEAD {filename}
```

Intermezzo: Referring to Commits

Git provides a shortcut to refer to commits based on the current state, so you do not need to always copy-paste `git log` output.

For example, to refer to the previous commit, you can either use `HEAD~1` or `HEAD^`, for example:

```
$ git diff HEAD~1
$ git diff HEAD^
```

To refer to the commit before the previous one, you can use `HEAD~2` or `HEAD^^`:

```
$ git diff HEAD~2
$ git diff HEAD^^
```

The `HEAD` reference can be used in most places you would use actual commit IDs.

Restoring Previous Versions

You can also restore the whole directory to a previous commit state using the `git revert` command:

```
$ git revert 7a6e47
[master a76f17d] Revert "Second commit"
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Notice that you need the commit ID of a previous commit. This can be obtained from the git log command and does not have to be the full ID. You can also use `HEAD`.

Restoring Previous Versions

Notice that `git revert` adds a new commit:

```
$ git log
commit a76f17dedf8d6a16d6ae910b18148272fead7282
Author: Wibowo Arindrarto <w.arindrarto@lumc.nl>
Date:   Tue Nov 24 16:50:38 2015 +0100

    Revert "Second commit"

    This reverts commit 7a6e47cfbb38048b46937d9f8d2427a7e6e20936.

commit 7a6e47cfbb38048b46937d9f8d2427a7e6e20936
Author: Wibowo Arindrarto <w.arindrarto@lumc.nl>
Date:   Tue Nov 24 16:13:59 2015 +0100

    Second commit

commit 54661709e859427358c97a94475643a7ccffa052
Author: Wibowo Arindrarto <w.arindrarto@lumc.nl>
Date:   Tue Nov 24 15:04:54 2015 +0100

    First commit
```

Who edited what?

`git blame` shows you the last author of each line

```
$ git blame
a76f17de (Wibowo Arindrarto 2015-11-24 16:13:59 +0100 1) Second version.
```

```
$ git blame
ef131a9f (bow 2016-07-11 09:52:57 +0200   1) # -*- coding: utf-8 -*-
ef131a9f (bow 2016-07-11 09:52:57 +0200   2) """
3c323a28 (bow 2016-07-14 00:30:54 +0200   3)     my.app
3c323a28 (bow 2016-07-14 00:30:54 +0200   4)     ~~~~~~
3c323a28 (bow 2016-07-14 00:30:54 +0200   5)
3c323a28 (bow 2016-07-14 00:30:54 +0200   6)     Functions and classes ...
3c323a28 (bow 2016-07-14 00:30:54 +0200   7)
ef131a9f (bow 2016-07-11 09:52:57 +0200   8) """
...
```

Cleaning untracked files

`git clean` is your 'janitor'

```
$ git status
On branch master

Initial commit

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)

$ git clean
fatal: clean.requireForce defaults to true and neither -i, -n, nor -f given;
refusing to clean

$ git clean -f
Removing README~
```

Extra: .gitignore

Certain files are not suitable for tracking by git:

- Binary files / executables
- PDF files
- Microsoft Office files

Other files are also not meant to be tracked:

- Password-containing files
- Large files

You can ignore these files by listing their names in a file called `.gitignore` in your directory root:

```
$ echo 'my_password.txt' >> .gitignore
$ echo '*.pdf' >> .gitignore
$ git add .gitignore
$ git commit -m "Add .gitignore file"
```

Acknowledgements

Jeroen Laros
Martijn Vermaat
Zuotian Tatum