Leiden University
Medical Center

Code management with Git

Git Basics

Table of contents

Getting started

To start a project you can either:

1. Start from scratch on your own:

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

Getting started

To start a project you can either:

1. Start from scratch on your own:

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

2. Or you can `clone` an existing remote (or local) repository:

```
$ git clone {path-to-repository}
Cloning into {repo-name} ...
```

Getting started

Where does git store its repository information?

You can see a hidden directory in a Git repository.

```
$ 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!

Getting started

Quick setup

If you have never used `git` before you need to tell it who you are. This information is saved in `.gitconfig` and used to mark each commit.

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

Getting started

Quick setup

If you have never used git before you need to tell it who you are. This information is saved in .gitconfig and used to mark each commit.

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

Local configuration for each repository is possible as well.

```
$ git config --local user.name {your-name-or-nick-name}
$ git config --local user.email {your-email-address}
```

Getting started

Quick setup

If you have never used git before you need to tell it who you are. This information is saved in .gitconfig and used to mark each commit.

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

Local configuration for each repository is possible as well.

```
$ git config --local user.name {your-name-or-nick-name}
$ git config --local user.email {your-email-address}
```

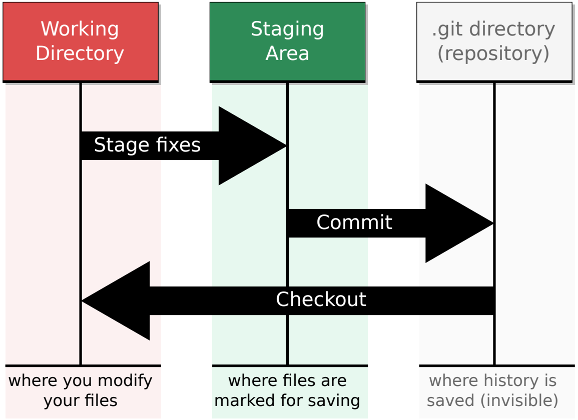
You would like colored output?

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

Getting started

Playing areas

Git operations track files around these three **areas** [a]:


Working Directory
Staging Area
.git directory (repository)
Stage fixes
Commit
Checkout
where you modify your files
where files are marked for saving
where history is saved (invisible)

[a] Adapted from the Pro Git Book.

Table of contents

Git Operations

Checking repository state

```
$ git status
nothing to commit (working directory clean)
```

Git Operations

Checking repository state

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

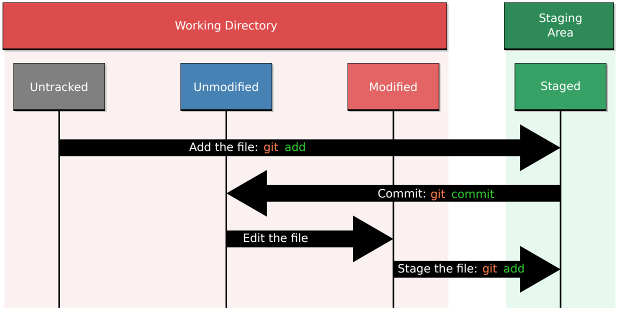
Git Operations

Checking repository state

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

Git itself will associate your files with various **states** [a]:

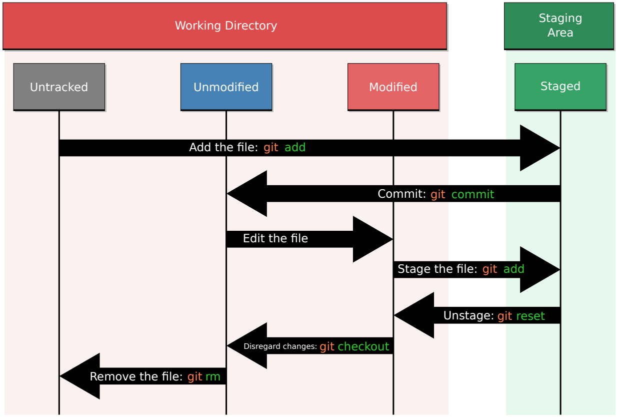


[a] Adapted from the Pro Git Book.

Git Operations

Git file states

Git itself will associate your files with various **states** [a]:



[a]Adapted from the Pro Git Book.

Git Operations

Adding/staging files

To start tracking the file, we need to stage it first:

```
$ git add README
```

Git Operations

Adding/staging files

To start tracking the file, we need to stage it first:

```
$ 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`.

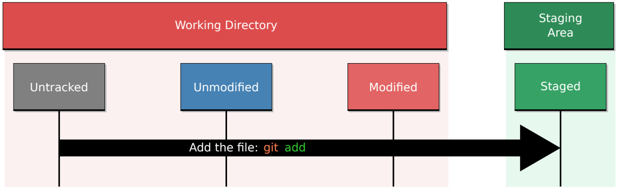
Git Operations

Adding/staging files

To start tracking the file, we need to stage it first:

```
$ git add README
```

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



Git Operations

Commit

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

Git Operations

Commit

```
$ git commit -m "First commit"
[master 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.

Git Operations

Commit

```
$ git commit -m "First commit"
[master 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.

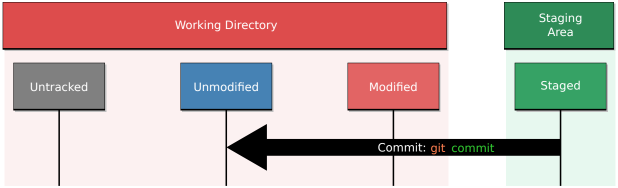
We can see that the file state has changed:

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

Git Operations

Commit

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



Git Operations

Let's update the file now:

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

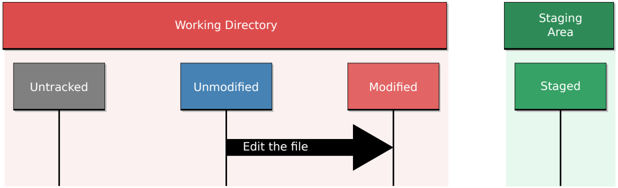
Git knows that there are changes to the file:

```
$ 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

Let's update the file now:

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



Git Operations

Check the differences

```
$ 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 for the next commit`. The differences are what you could tell Git to further add to the staging area but you still haven't.

Git Operations

Check the differences

```
$ git add README
$ git diff
$
```

Git Operations

Check the differences

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

This displays the changes you staged for the **next commit** relative to the **previous commit**.

Git Operations

Check the differences

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

This displays the changes you staged for the **next commit** relative to the **previous commit**.
You can also specify a specific **commit id** to which to compare the staged files with: `git diff --cached <commit-id>`

Git Operations

To stage and commit the file again we could use:

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

Git Operations

To stage and commit the file again we could use:

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

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

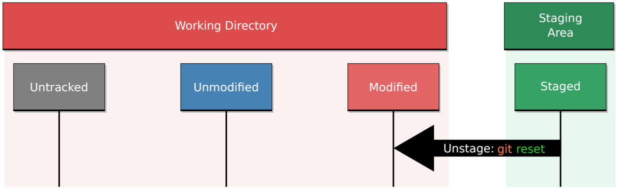
```
$ git commit -am "Second commit"
[master ef70f09] Second commit
1 file changed, 1 insertion(+), 1 deletion(-)
```

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

Git Operations

Unstage

```
$ echo "Some mistake" >> README
$ git add README
$ git reset HEAD README
```



Git Operations

Check commit history

```
$ git log
commit 7a6e47cfbb38048b46937d9f8d2427a7e6e20936
Author: Zorro <zorro@poor.es>
Date: Tue Nov 24 16:13:59 2015 +0100

   Second commit

commit 54661709e859427358c97a94475643a7ccffa052
Author: Zorro <zorro@poor.es>
Date: Tue Nov 24 15:04:54 2015 +0100

   First commit
```

Git Operations

Restoring Previous Versions

One that is used quite commonly is to discard a `working directory` file changes and **restore its latest repository state**:

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

To discard all changes in the `working directory`:

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

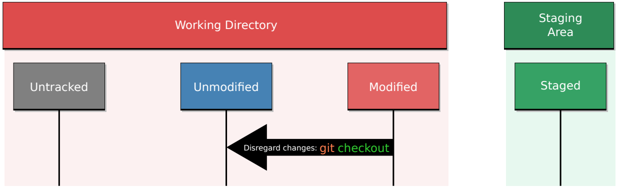
Restoring Previous Versions

One that is used quite commonly is to discard a `working directory` file changes and **restore its latest repository state**:

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

To discard all changes in the `working directory`:

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



Git Operations

Restoring Previous Versions

To retrieve a file from a specific commit into the `staging area`:

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

It's up to you to include this file into a new commit.

Table of contents

Explicit not tracking

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 - use git-annex for those.

Extras

Explicit not tracking

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 - use git-annex for those.

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

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

Extras

Who edited what?

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

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

Extras

Cleaning untracked files

```
$ git status
On branch master

Initial commit

Untracked files
(use "git add <file>..."to include in what will be committed)
   README.bkp

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.bkp
```

Acknowledgements

Martijn Vermaat
Wibowo Arindrarto
Szymon Kiełbasa
Jeroen Laros

http://git-scm.com/book