

A Brief Introduction to Git

Flavio Stanchi
(based on slides by Hautahi Kingi)

Introduction

Version control is better than mailing files back and forth because:

- ▶ Nothing that is committed to version control is ever lost. It's always possible to go back in time to see exactly who wrote what on a particular day, or what version of a program was used to generate a particular set of results.
- ▶ It keeps a record of who made what changes when, so that if people have questions later on, they know who to ask.
- ▶ It's hard (but not impossible) to accidentally overlook or overwrite someone's changes: the version control system automatically notifies users whenever there's a conflict between one person's work and another's.

Git is one of many version control systems. It is more complex than some alternatives, but it is widely used, both because it's easy to set up and because of a hosting site called GitHub, which we will get to later.

Set Up

After downloading Git, an easy way to check if it is installed and working properly is

```
$ git help
```

On our first time with Git on a new machine, we need to do a few things.

```
$ git config --global user.name "Flavio Stanchi"  
$ git config --global user.email "fs379@cornell.edu"  
$ git config --global color.ui "auto"  
$ git config --global core.editor "nano"
```

Navigate to the `my_thesis` folder that we created earlier (see command line slides).

```
$ cd Workspace/my_thesis
```

and tell git to make it a *repository*. A repository is a storage area where a version control system stores old revisions of files and information about who changed what, when.

```
$ git init  
Initialized empty Git repository in /Users/flaviostanchi/Workspace/my_thesis/.git/
```

The .git subdirectory

Let's take a look at our new directory

```
$ ls  
code.m  introduction.txt
```

It looks like nothing has changed. But *ls* only lists files which are not hidden. The *init* command created a hidden directory, which we can see by using the *-a* flag

```
$ ls -a  
code.m  .git  introduction.txt
```

Git stores information about the project in this special *.git* sub-directory. If deleted, we lose the project's entire history.

Git Status

```
$ git status
On branch master

Initial commit

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

        code.m
        introduction.txt

nothing added to commit but untracked files present (use "git add" to track)
```

The output states that we are located on the master branch (more on this later). The *Initial commit* message states that we are yet to make a commit. The *Untracked files* message means that there are files in the directory that Git is not keeping track of.

Git Add

We tell Git to stage untracked/modified files using *git add*:

```
$ git add code.m introduction.txt
```

and checking the status again...

```
$ git status
On branch master

Initial commit

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

        new file:   code.m
        new file:   introduction.txt
```

... tells us that Git knows that it needs to keep track of the files *code.m* and *introduction.txt*. But it has not yet recorded the changes.

Git Commit

Let's *commit* the added changes.

```
$ git commit -m "First draft introduction and code"
[master (root-commit) 1ec0eef] First draft introduction and code
2 files changed, 8 insertions(+)
create mode 100755 code.m
create mode 100644 introduction.txt
```

- ▶ Git takes everything we have told it to save by using *git add* and stores a copy permanently inside the special *.git* directory.
- ▶ Version has a short identifier 1ec0eef.
- ▶ -m flag to record a comment that will help us remember later on what we did and why.

Now let's check the status again

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

This is the nice clean message that tells us everything is up to date and committed.

Adding new Files

Let's create a latex file called template.tex.

```
$ ls
code.m          template.aux    template.pdf    template.tex
draft.txt       template.log    template.synctex.gz
```

My particular latex compiler created a number of auxiliary files. Checking the status of the repository gives

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    template.aux
    template.log
    template.pdf
    template.synctex.gz
    template.tex

nothing added to commit but untracked files present (use "git add" to track)
```

The *template* files are untracked because I have not added them.

Git ignore

I don't particularly care about the auxiliary latex files. They are a waste of disk space and distract us from changes that actually matter. We can tell Git to ignore them by creating a file in the root directory of our project called `.gitignore`.

```
$ nano .gitignore
```

A screenshot of a terminal window with a nano editor. The window title is "my_thesis -- nano". The editor's status bar at the top shows "GNU nano 2.0.6", "File: .gitignore", and "Modified". The main text area contains three lines of text: "*.aux", "*.log", and "*.gz". At the bottom, the nano editor's command palette is visible, showing various shortcuts like "Get Help", "WriteOut", "Read File", "Prev Page", "Cut Text", "Cur Pos", "Exit", "Justify", "Where Is", "Next Page", "UnCut Text", and "To Spell". A "New File" button is also present in the center of the command palette.

```
GNU nano 2.0.6 File: .gitignore Modified
*.aux
*.log
*.gz

[ New File ]
Get Help WriteOut Read File Prev Page Cut Text Cur Pos
Exit Justify Where Is Next Page UnCut Text To Spell
```

We've told Git to ignore any file whose name ends in `.aux`, `.log` and `.gz`.

Git ignore

Once we have created the .gitignore file, the output of git status is much cleaner:

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    .gitignore
    template.pdf
    template.tex

nothing added to commit but untracked files present (use "git add" to track)
```

Git no longer notices those pesky auxiliary latex files and we can focus on the changes that matter.

Modifying files

Now let's edit our introduction.txt file and check the status again.

```
$ nano introduction.txt
$ cat introduction.txt
Once upon a time, in a galaxy far far away...
EPISODE IV
$ 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:   introduction.txt

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

        .gitignore
        template.pdf
        template.tex

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

As well as the untracked files in the previous status check, Git also tells us that the introduction.txt file has been modified.

Completing the Git Cycle

Let's now add and commit these changes.

```
$ git add introduction.txt template.tex template.pdf .gitignore
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   .gitignore
        modified:   introduction.txt
        new file:   template.pdf
        new file:   template.tex
$ git commit -m "New latex file and added line to introduction.txt"
[master d796846] New latex file and added line to introduction.txt
4 files changed, 49 insertions(+)
create mode 100644 .gitignore
create mode 100644 template.pdf
create mode 100644 template.tex
```

and checking the status....

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

... we're back to a clean sheet.

An overview of the Git Cycle

We should now be able to see the basic structure of how Git works. Once a directory is set up as a Git repository using *git init*, any change that we make to that directory is noted by Git. The workflow is then as follows:

1. Modify/create new folders and files.
2. Stage these changes ready to commit by using *git add*.
3. Commit the changes using *git commit* which permanently saves the current version of that repository.

Undoing a “bad” commit

Suppose we have made some unwanted changes to a file, and we have committed that file. For example, we want to retrieve the file `introduction.txt` from the “First draft introduction and code” commit.

```
$ git log --oneline
d796846 New latex file and added line to introduction.txt
1ec0eef First draft introduction and code

$ git checkout 1ec0eef introduction.txt
$ cat introduction.txt
Once upon a time, in a galaxy far far away...
```

Note that this will produce changes in our working directory. Then, we can commit these changes to our repository.

```
$ git commit -m "Retrieved file introduction.txt from previous commit"
[master 8b8c6a3] Retrieved file introduction.txt from previous commit
1 file changed, 1 insertion(+)
```

Hard delete unpublished commits

Now suppose that we really want to eliminate all traces of a series of particularly bad commits. For example, suppose we have committed the following:

```
$ nano introduction.txt
$ cat introduction.txt
Once upon a time, in a galaxy far far away...
Star Trek is better than Star Wars
$ git add introduction.txt
$ git commit -m "Added a stupid line to introduction.txt"
[master 337d55e] Added a stupid line to introduction.txt
1 file changed, 1 insertion(+)
```

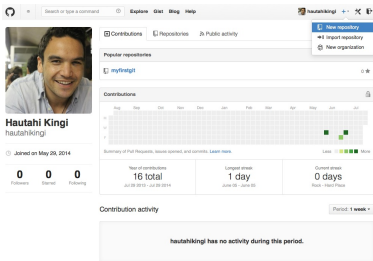
We want to go back to our previous commit, changing the history of the commits.

```
$ git log --oneline
337d55e Added a stupid line to introduction.txt
8b8c6a3 Retrieved file introduction.txt from previous commit
d796846 New latex file and added line to introduction.txt
1ec0eef First draft introduction and code
$ git reset --hard 8b8c6a3
HEAD is now at 8b8c6a3 Retrieved file introduction.txt from previous commit
```

GitHub: Taking Version Control Online

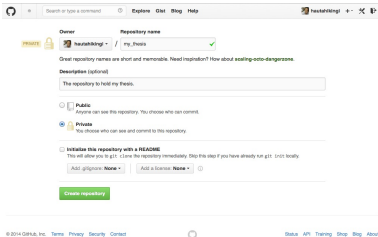
Version control really comes into its own when we begin to collaborate with other people. In practice, a remote repository stored on a central hub is preferable to one stored on someone's laptop. For this reason, websites like Github and BitBucket are becoming increasingly popular.

Login to your profile page on GitHub. Click on the icon in the top right corner to create a new repository called my_thesis.



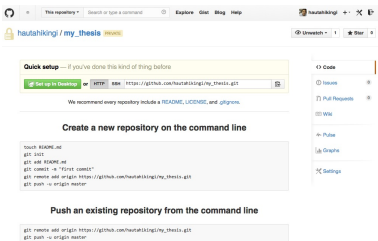
Public repositories allow any Github user to access your files. A student account receives 5 free private repositories.

Creating an Online Repository



The screenshot shows the GitHub 'Create repository' form. At the top, there's a search bar and navigation links for 'Explore', 'Git', 'Blog', and 'Help'. The user 'hautahiking!' is logged in. The form fields include: 'Owner' (hautahiking!), 'Repository name' (my_thesis), 'Description (optional)' (The repository to hold my thesis), 'Public' (selected) or 'Private' radio buttons, and a checkbox for 'Initialize this repository with a README'. Below these are dropdowns for 'Add a gitignore' and 'Add a license'. A green 'Create repository' button is at the bottom. The footer shows '© 2014 GitHub, Inc.' and links for 'Terms', 'Privacy', 'Security', 'Contact', 'Status', 'API', 'Training', 'Shop', 'Blog', and 'About'.

Clicking the green button creates a repository on the Github website. Once created, GitHub displays a page with a URL and some information on how to configure your local repository.



The screenshot shows the GitHub repository page for 'hautahiking! / my_thesis'. It includes a 'Quick setup' section with links to 'Set up to Develop' and 'HTTP', and a text box with the repository URL. Below this is a section titled 'Create a new repository on the command line' with a code block containing the following commands:

```
touch README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/hautahiking/my_thesis.git
git push -u origin master
```

Another section titled 'Push an existing repository from the command line' contains the following commands:

```
git remote add origin https://github.com/hautahiking/my_thesis.git
git push -u origin master
```

The right sidebar shows options for 'Code', 'Issues', 'Pull Requests', 'Wiki', 'Pulse', 'Graphs', and 'Settings'.

Connecting Local and Online Repositories

The home page of the GitHub repository includes instructions on how to link our newly created repository with our local repository. We are interested in the instructions in the second box.

Making sure we are in the `my_thesis` directory locally...

```
$ git remote add origin https://github.com/fs379/my_thesis.git
```

This command tells git to add a remote named *origin* at the url *https://github.com/fs379/my_thesis.git*.

We can check that the command worked by typing

```
$ git remote -v
origin  https://github.com/fs379/my_thesis.git (fetch)
origin  https://github.com/fs379/my_thesis.git (push)
```

which lists all remote repositories. The `-v` flag is short for verbose which tells git to show the remote url as well as the nickname.

Pushing

So far all we have done is set up a remote repository and linked it to our local repository. But if we take a look at our remote repository, it is still empty. We fix this by pushing the changes from our local repository to the GitHub repository

```
$ git push -u origin master
```

It will sometimes ask for your username and password

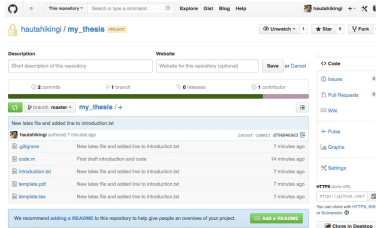
```
Username for 'https://github.com': fs379  
Password for 'https://fs379@github.com':
```

When you are typing in your password the cursor won't move. Don't panic just type it out and press enter.

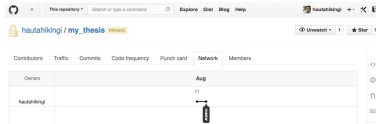
```
Counting objects: 10, done.  
Delta compression using up to 4 threads.  
Compressing objects: 100% (8/8), done.  
Writing objects: 100% (10/10), 42.65 KiB | 0 bytes/s, done.  
Total 10 (delta 1), reused 0 (delta 0)  
To https://github.com/fs379/my_thesis.git  
* [new branch]      master -> master  
Branch master set up to track remote branch master from origin.
```

GitHub - a visual repository

The homepage for our remote repository confirms that it is now linked to our local repository.



Selecting *Networks* from the *Graphs* tab on the right...



... gives a visual representation of the evolution of the repository. Each commit is represented by a node in the graph. There are two nodes because we have performed two commits.

Undo published commits with new commits

Note: you should never use *git reset* after the commits you want to undo have been pushed to a public repository. Instead, it is possible to use *git revert* to undo one or more target commits.

```
$ git log --oneline
8b8c6a3 Retrieved file introduction.txt from previous commit
d796846 New latex file and added line to introduction.txt
1ec0eef First draft introduction and code
$ git revert 8b8c6a3
[master f05a49c] Revert "Retrieved file introduction.txt from previous commit"
1 file changed, 1 deletion(-)
```

This doesn't change the history of the commits and is therefore preferable.

Note: you can revert multiple commits at the same time; this will in turn create multiple new commits. See this [blog post](#) about all the differences between undoing commits.

Centralized vs Distributed Version Control Systems

Version control systems are often divided into two groups: centralized and distributed.

Centralized version control systems, such as subversion, are based on the idea that there is a single central copy of your project on a server, and programmers *commit* their changes to this central copy.

The basic workflow is to pull down any changes other people have made from the central server. Then make your changes and commit these to the central server.

Git is a Distributed VCS. These do not necessarily rely on a central server to store all versions of a project's files. Instead, every developer **clones** a copy of a repository and has the full history of the project on their own hard drive. This copy has all of the metadata of the original.

So instead of a central repository being required, it is now optional and purely a matter of preference.

Centralized vs Distributed Version Control Systems

We have already made use of some of the features of a DVCS.

We were able to commit changes locally even before we created the central remote repository.

Once we pushed our local copy to the remote repository, GitHub still registered each separate commit (two in our case). This is one of the main advantages of distributed systems. Centralized systems only record commits that are made directly to the remote repository. For example, in subversion, there would only be one node in our network tree.

Being able to commit locally is advantageous for a number of reasons. First, performing actions is extremely fast because Git only needs to access the hard drive, not a remote server (except for pushing and pulling of course).

Everything but pushing and pulling can be done without an internet connection. So you can work on a plane, and you won't be forced to commit several bugfixes as one big changeset.

Pulling

We can pull changes from the remote repository to the local one as well:

```
$ git pull origin master
From https://github.com/fs379/my_thesis
 * branch      master      -> FETCH_HEAD
Already up-to-date.
```

Pulling has no effect in this case because the two repositories are already synchronized. If someone else had pushed some changes to the repository on GitHub, though, this command would download them to our local repository.

Cloning

We can simulate working with a collaborator using another copy of the repository on our local machine. To do this, *cd* to another directory on your computer. Instead of creating a new repository here with *git init*, we will **clone** the existing repository from GitHub.

```
$ cd /Users/flaviostanchi/Dropbox/temporary
$ git clone https://github.com/fs379/my_thesis.git
Cloning into 'my_thesis'...
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 10 (delta 1), reused 10 (delta 1)
Unpacking objects: 100% (10/10), done.
Checking connectivity... done.
```

I navigated to a new folder called *temporary* and then created a fresh local copy of a remote repository. Our computer now has two copies of the repository.

Working with a collaborator

Let's make a change to the copy in */temporary/my_thesis*:

```
$ cd my_thesis
$ nano introduction.txt
$ cat introduction.txt
Once upon a time, in a galaxy far far away...
EPISODE IV
A new hope
```

and now let's add and commit

```
$ git add introduction.txt
$ git commit -m "Added a third line to introduction.txt"
[master 2b1b130] Added a third line to introduction.txt
1 file changed, 1 insertion(+)
```

and checking the status...

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working directory clean
```

This message tells us that we have committed all changes, but that we have not yet pushed these changes to the remote repository on Github. Again, this is a feature of **distributed** version control systems.

Working with a collaborator

Let's now push the change to Github

```
$ git push origin master
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 345 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/fs379/my_thesis.git
170ae6f..3cafe3a master -> master
```

Checking the status...

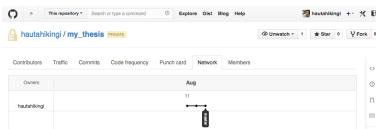
```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean
```

This tells us that we have committed all changes, and that this is also reflected in the remote repository.

Working with a collaborator

[label=coll] The network map also confirms this...



We can now download changes into the original repository on our machine:

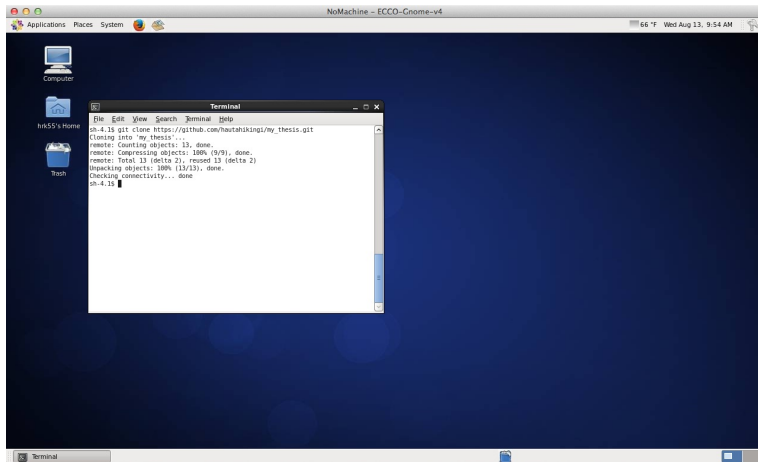
```
$ cd /Users/flaviostanchi/Dropbox/my_thesis
$ git pull origin master
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 2), reused 3 (delta 2)
Unpacking objects: 100% (3/3), done.
From https://github.com/fs379/my_thesis
 * branch          master      -> FETCH_HEAD
    d796846..2b1b130  master      -> origin/master
Updating d796846..2b1b130
Fast-forward
 introduction.txt | 1 +
 1 file changed, 1 insertion(+)
```

Both our local repositories are now in sync with the remote repository. In practice, we would never have two copies of the same remote repository on our laptop at once.

An ECCO Example

[label=ECCO] Pushing and pulling changes gives us a reliable way to share work between different people and machines.

Login to ECCO and clone the remote repository from the terminal.



An ECCO Example

Now create some changes, add and commit.

```
$ cd my_thesis
$ nano conclusion.txt
$ cat conclusion.txt
The franchise began in 1977 with the release of the film Star Wars
$ git add conclusion.txt
$ git commit -m "Began conclusion.txt"
[master ac717a9] Began conclusion.txt
Committer: fs379 <fs379@ecco.vrdc.cornell.edu>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:
```

```
git config --global user.name "Your Name"
git config --global user.email you@example.com
```

After doing this, you may fix the identity used for this commit with:

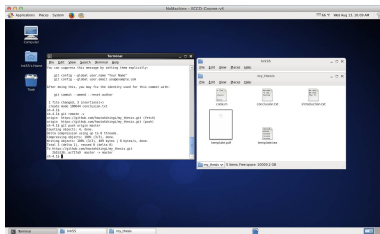
```
git commit --amend --reset-author
```

```
1 file changed, 1 insertion(+)
create mode 100644 conclusion.txt
```

Because I have not configured my Git settings on the ECCO server, Git has assigned me a different username.

An ECCO Example

Let's now push the changes to our remote repository.



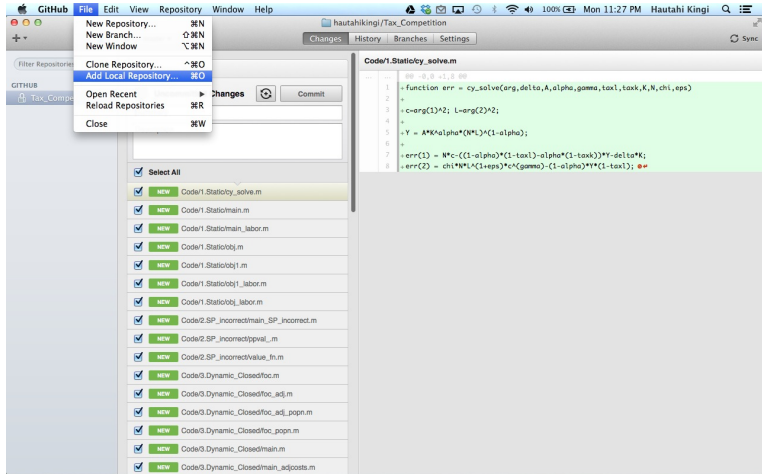
Switching back to your local copy we can then pull these changes.

```
$ git pull origin master
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1)
Unpacking objects: 100% (3/3), done.
From https://github.com/fs379/my_thesis
 * branch      master       -> FETCH_HEAD
 2b1b130..ac717a9 master    -> origin/master
Updating 2b1b130..ac717a9
Fast-forward
 conclusion.txt | 3 +++
 1 file changed, 3 insertions(+)
 create mode 100644 conclusion.txt
```

Used in this way, Git acts like a more robust Dropbox.

The GitHub GUI

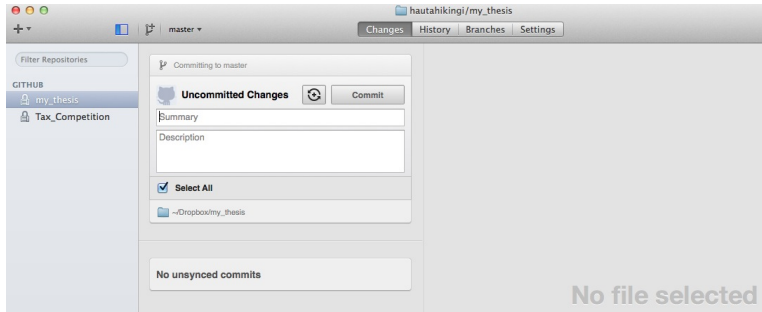
There are now a number of fantastic GitHub GUIs. Let's take a quick look at the GUI created by Github itself.



When we first launch the GUI we want to tell it to add our local repository from the menu.

The GitHub GUI

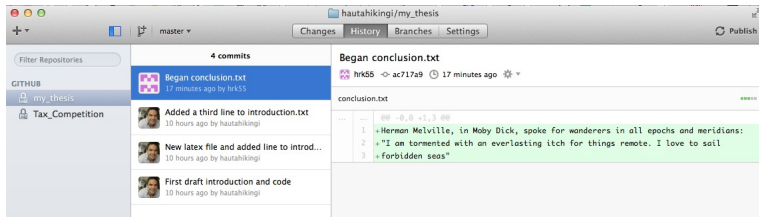
Select the repository in the left-hand menu.



The right hand pane shows a range of information about that repository contained in four separate panels. The *changes* panel shows the changes made within the repository which are yet to be committed. It effectively shows the same information as typing *git status* in the command line. As you can see, we have nothing untracked or changed since the last commit.

The GitHub GUI

The history panel allows us to select each commit, and investigate what changes were made since the last commit.



It provides the same information as the *git log* and *git diff* commands which we have not covered.

Most of this material was taken from the Software Carpentry website and their excellent bootcamp. More detailed explanations of what I discuss in these notes can be found at: <http://software-carpentry.org/lessons.html>

